

Theoretical foundations of dynamic program slicing[☆]

Dave Binkley^a, Sebastian Danicic^b, Tibor Gyimóthy^c, Mark Harman^{d,*},
Ákos Kiss^c, Bogdan Korel^e

^aLoyola College in Maryland, Baltimore, Maryland 21210-2699, USA

^bGoldsmiths College, University of London, New Cross, London SE14 6NW, UK

^cInstitute of Informatics, University of Szeged, 6720 Szeged, Hungary

^dKing's College London, Strand, London WC2R 2LS, UK

^eIllinois Institute of Technology, Chicago, IL 60616-3793, USA

Received 18 February 2005; received in revised form 30 November 2005; accepted 4 January 2006

Communicated by G. Levi

Abstract

This paper presents a theory of dynamic slicing, which reveals that the relationship between static and dynamic slicing is more subtle than previously thought. The definitions of dynamic slicing are formulated in terms of the projection theory of slicing. This shows that existing forms of dynamic slicing contain three orthogonal dimensions in their slicing criteria and allows for a lattice-theoretic study of the subsumption relationship between these dimensions and their relationship to static slicing formulations. © 2006 Elsevier B.V. All rights reserved.

Keywords: Program slicing; Dynamic slicing

1. Introduction

Program slicing is a technique for extracting parts of a program which affect a chosen set of variables of interest. By focusing on the computation of only a few variables the slicing process can be used to eliminate parts of the program which cannot affect these variables, thereby reducing the size of the program. The reduced program is called a slice.

Slicing has many applications because it allows a program to be simplified by focusing attention on a sub-computation of interest for a chosen application. The user specifies the sub-computation of interest using a 'slicing criterion'. This paper is concerned with the relationships between slicing criteria for dynamic and static forms of slicing and sets of slices allowable according to the different slicing techniques which use these criteria.

Among other applications, slicing has been applied to reverse engineering [14,56], program comprehension [21,34], software maintenance [13,17,24,25], debugging [1,42,50,62], testing [7,29,31,37,38], component re-use [3,16], program integration [11,39], and software metrics [5,48,51]. There are several surveys of slicing techniques, applications and variations [9,10,20,33,57].

[☆] A preliminary version of this paper appeared at the Fourth IEEE Workshop on Source Code Analysis and Manipulation [8].

* Corresponding author. Tel.: +44 207848 2895; fax: +44 207848 2851.

E-mail addresses: binkley@cs.loyola.edu (D. Binkley), s.danicic@gold.ac.uk (S. Danicic), gyimi@inf.u-szeged.hu (T. Gyimóthy), mark@dcs.kcl.ac.uk (M. Harman), akiss@inf.u-szeged.hu (Á Kiss), korel@iit.edu (B. Korel).

Slices can be constructed statically [61,41] or dynamically [45,2]. In static slicing, the input to the program is unknown and the slice must therefore preserve meaning for all possible inputs. By contrast, in dynamic slicing, the input to the program is known, and so the slice needs only preserve meaning for the input under consideration. Dynamic slicing is particularly useful in applications like debugging, where the input to the program has a crucial bearing on the problem in hand.

This paper is concerned with the formal definitions and properties of dynamic slicing (rather than algorithms for computing them). It employs the projection theory of program slicing introduced by Harman et al. [30,32]. Using this theory, it is possible to study and explain similarities and differences between slicing definitions.

The theory was first used to examine the differences and similarities between amorphous and syntax-preserving forms of slicing [30,32]. This paper uses the projection theory to investigate the nature of dynamic slicing as originally formulated by Korel and Laski [45].

The results on dynamic slicing reveal that the nature of the dynamic slicing criterion is more subtle than previous authors have observed [12,23,58]. There is no simple two-element subsumption 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, path sensitivity, and the iteration count sensitivity. Previous authors have regarded the addition of program input as the only aspect separating static and dynamic slicing criteria.

Using the projection theory, these three concepts are isolated, allowing for a lattice-theoretic analysis of the ‘subsumption’ relationship between various formulations of static and dynamic slicing. The analysis reveals the existence of new, as yet unexplored slicing criteria, which may find applications in their own right.

The paper also proves that the ‘subsumption’ relationship for the semantic properties of slicing criteria are respected by all definitions of slicing which use the standard statement deletion ordering. This means that where technique t subsumes techniques t' , then all slices computed according to t' can also be computed by t .

Following Venkatesh [58], we use the term ‘executable’ to mean that a slice of program p is itself an executable program whose semantics are a projection of the semantics of p . This is to be contrasted with a non-executable slice, which is a set of statements identified as being relevant to the slicing criterion [41,6]. We prefer executable slicing definitions because they can be more easily 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 primary contributions of the paper are as follows [8]:

- (1) New slicing criteria and their possible applications are uncovered in the existing definitions of dynamic slicing.
- (2) A subsumption relationship is formally demonstrated between the semantic equivalence relations of eight forms of slicing (four dynamic and four static).
- (3) This semantic subsumption relationship is extended to cover the eight slicing techniques.

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 discusses the definition of dynamic slicing introduced by Korel and Laski and reveals its incomparability to static backward slicing. Sections 5–7 set up a unified framework to describe the semantic properties preserved by various slicing definitions preserve. These sections analyze the discussed slicing methods using the unified framework to reveal the subsumption relationship between them. Finally, Section 8 presents related work and Section 9 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 partial order on programs.

Definition 2 (*Semantic equivalence*). A semantic equivalence, denoted by \approx , is an equivalence relation on programs.

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. An example instantiation of this framework for static slicing is given using the programs shown in Figs. 1, 9, and 10.

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 [30], so all slicing definitions in this paper share the following syntactic ordering.

Definition 4 (Traditional 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 Fig. 1, $p'_{\text{ex1}} \sqsubseteq p_{\text{ex1}}$ because $F(p'_{\text{ex1}}) \subseteq F(p_{\text{ex1}})$. Written out, $F(p'_{\text{ex1}}) = \{(1, \text{scanf}(\text{"\%d"}, \&n)), (3, p = 1), (4, \text{while } (n > 1)), (7, p = p * n), (8, n = n - 1)\}$ and $F(p_{\text{ex1}}) = \{(1, \text{scanf}(\text{"\%d"}, \&n)), (2, s = 0), (3, p = 1), (4, \text{while } (n > 1)), (6, s = s + n), (7, p = p * n), (8, n = n - 1)\}$. Or, repeated with just the line numbers, $F(p'_{\text{ex1}})$ includes $\{1, 3, 4, 7, 8\}$ while $F(p_{\text{ex1}})$ includes $\{1, 2, 3, 4, 6, 7, 8\}$.

3. Weiser's 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 slice semantics [61]. This definition is more general than some subsequent definitions, which required the slice to be taken with respect to a variable defined or used at the point of interest and always include this point in the slice [41]. The difference has few practical implications, but the reader's attention is drawn to this difference since the present paper is concerned with definitions and semantics. This definition considers slices of terminating programs only.

Definition 5 (State trajectory). A *state trajectory* is a finite sequence of (line-number, state) pairs:

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

where a state is a partial function mapping a variable to a value, and entry i is (l_i, σ_i) if after i statement executions the state is σ_i , and the next statement 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 :

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

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

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

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_{ex1} : Original Program		p'_{ex1} : Slice for $(\{p\}, 9)$	

Fig. 1. A program and one of its slices.

Definition 8 (*Proj*). For slicing criterion (V, n) , which specifies a slice taken with respect to a set of variables V at line number n , and state trajectory $T = (l_1, \sigma_1)(l_2, \sigma_2) \dots (l_k, \sigma_k)$,

$$Proj_{(V,n)}(T) = Proj'_{(V,n)}((l_1, \sigma_1)) \dots Proj'_{(V,n)}((l_k, \sigma_k)).$$

In the following we will use the symbol \oplus to denote concatenation of sequences. This allows us to write trajectories as follows:

$$Proj_{(V,n)}(T) = \bigoplus_{i=1}^k Proj'_{(V,n)}((l_i, \sigma_i)).$$

For the *slicing criterion* (V, 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 $p \mathcal{S}(V, n) q$, iff for all input states σ , when the execution of p in σ gives rise to a state trajectory T_p^σ and the execution of q in σ gives rise to a state trajectory T_q^σ , then $Proj_{(V,n)}(T_p^\sigma) = Proj_{(V,n)}(T_q^\sigma)$.

Example. For the slice shown in Fig. 1, p_{ex1} is static backward equivalent to p'_{ex1} , written $p_{\text{ex1}} \mathcal{S}(\{p\}, 9) p'_{\text{ex1}}$ because for every input state σ $Proj_{(\{p\},9)}(T_{p_{\text{ex1}}}^\sigma) = Proj_{(\{p\},9)}(T_{p'_{\text{ex1}}}^\sigma)$.

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 slices that respect a different projection of the semantics of the program from which they are constructed. Instantiating Definitions 4 and 9 into Definition 3, yields the following:

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

Example. Program p'_{ex1} shown in Fig. 1 is a static backward slice of program p_{ex1} with respect to the slicing criterion $(\{p\}, 9)$ because p'_{ex1} is a $(\sqsubseteq, \mathcal{S}(\{p\}, 9))$ projection of p_{ex1} .

4. Korel and Laski's 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 [45] first introduced the dynamic slice, which need only preserve the effect of the original 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;
$p_{\text{ex}2}$: Original Program		$p'_{\text{ex}2}$: Slice with respect to $(\{y\}, 7)$	

Fig. 2. A static slice, which is not a KL-slice.

upon the slicing criterion for a single 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.

The literature on dynamic slicing includes many different algorithms [2,4,27,43,45,47]. Many of these algorithms do not necessarily output executable programs [2,4,27,43]. Rather, they regard a dynamic slice as the collection of statements that have an effect upon the slicing criterion given the chosen input. This paper is concerned solely with executable forms of slicing. Each of the remaining techniques provides a slightly different definition of what makes one program a dynamic slice of another.

As defined by Korel and Laski an executable dynamic slicing criterion $c = (x, I^q, V)$, which, like static slicing criterion (V, n) includes a set of variables V . Unlike the static slicing criterion, it also includes the program's input x and replaces the location of interest n with I^q which is the q th instruction in the execution trajectory,¹ which is I . Thus, a slice can be taken with respect to a particular instance rather than all instances of a statement (instruction) from the program.

The definition uses two auxiliary functions on sequences, *Front* and *DEL* [45]. $\text{Front}(T, i)$ is the 'front' i elements of sequence T from 1 to i inclusive. $\text{DEL}(T, \pi)$ is a filtering operation, which takes a predicate π and returns the sequence obtained by deleting elements of T that satisfy π . The following definition is taken verbatim from Korel and Laski's work on dynamic slicing [45].

Definition 11 (Korel and Laski's dynamic slice). Let $c = (x, I^q, V)$ be a slicing criterion of a program p and T the trajectory of p on input x . A *dynamic slice* of p on c is any executable program p' that is obtained from p by deleting zero or more statements such that when executed on input x , produces a trajectory T' for which there exists an execution position q' such that

(KL1) $\text{Front}(T', q') = \text{DEL}(\text{Front}(T, q), T(i) \notin N' \wedge 1 \leq i \leq q)$,

(KL2) 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' ,

(KL3) $T'(q') = T(q) = I$,

where N' is a set of instructions in p' .

A common belief is that a static slice is (an overly large) Korel and Laski style dynamic slice. One intuitively expects that a dynamic slicing criterion is looser than a static one, since it preserves the semantics of a program for only one fixed input instead of all possible ones. Moreover, a dynamic slicing criterion selects only one occurrence of an instruction from the trajectory, as opposed to the static slicing, where all occurrences of the point of interest are taken into account.

However, as Fig. 2 reveals, Korel and Laski's definition of dynamic slicing is incomparable with the definition of static slicing, since not all static slices are appropriate KL-slices (and not all KL-slices are static slices, which is trivial). In Fig. 2, program $p'_{\text{ex}2}$ is a valid static slice with respect to $(\{y\}, 7)$ since at Line 7 the value of y is 1 for all inputs, just like in $p_{\text{ex}2}$. However, $p'_{\text{ex}2}$ is not a Korel and Laski style dynamic slice of $p_{\text{ex}2}$ with respect to $(\langle \rangle, 7^5, \{y\})$, because the trajectory of $p_{\text{ex}2}$ is (1 2 3 4 7), but the trajectory of $p'_{\text{ex}2}$ is (1 3 6 7). Having different execution paths violates KL1, since the truncated and filtered trajectories differ, i.e., (1 3 4 7) \neq (1 3 6 7).

¹ In Korel and Laski's interpretation, a trajectory is simply a finite sequence of line numbers, as opposed to Weiser's definition of *state* trajectory (see Definition 5). However, in the following, we will refer to both kinds of trajectories as *trajectory* and the context will make clear which meaning is used.

Notice that the cause of incomparability between KL-dynamic-slicing and static slicing is that KL-dynamic-slicing is “looser” as it must preserve behavior for only a single input (a desired effect), while, because of KL1, it is also more strict. Thus, restriction KL1 can prevent us from choosing an otherwise acceptable program from several semantically equivalent programs.

5. The unified equivalence

Static and dynamic slicing criteria differ in three mutually independent (i.e., orthogonal) aspects

- whether only one fixed or all possible inputs are taken into account,
- whether one or all occurrences of an instruction in the trajectory are considered, and
- whether the execution path is important or not.

To analyze the relation of different slicing methods we need to give a unified description of the above-mentioned aspects. However, Definitions 7 and 8 are not sufficient for this purpose as they cannot capture the execution path requirement.

To set up a unified framework we extend these definitions by introducing counter parts to *Proj* and *Proj'* named *Proj** and *Proj'**, respectively. The extension splits the “statement” parameter *n* into *P* and *I*: *P*, an instruction-natural number pair, identifies those instruction occurrences from the trajectory whose semantics must be preserved. Parameter *I* captures the trajectory requirement of KL1 by keeping only the line number, in the form of (n, \perp) , for those instructions that are not in the slicing criterion but get executed.

Notice that, in the following definitions, the notation is different from that used by Korel and Laski. While, in Definition 11 I^q represents the q th instruction in the trajectory, which is I , $n^{(k)}$ is used to denote the k th occurrence of instruction n in the trajectory. The change helps to capture the iteration count component of the Korel and Laski slicing criterion.

Definition 12 (*Proj**). *Proj** is defined in terms of five parameters: a set of variables V , a set of (line-number, natural number) pairs P , a set of line numbers I , a (line-number, natural number) pair $n^{(k)}$, and a 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 (*Proj'**). For a set of variables V , set of (line-number, natural number) pairs P , set of line numbers I and state trajectory T :

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

where k_i is the number of occurrences of n_i in the first i elements of T (i.e., $n_i^{(k_i)}$ is the most recent occurrence of n_i in $T[0] \dots T[i]$), and l is the highest index in T such that $n_l^{(k_l)} \in P$.

Observe that if $P = \{n\} \times \mathbf{N}$ and $I = \emptyset$ then $Proj_{(V,P,I)}^*(T) = Proj_{(V,n)}(T)$, since the middle case of *Proj** can be dropped. This leaves Weiser’s definition of *Proj*. However, by choosing different values for P and I , *Proj** can capture Korel and Laski’s requirements as well. Consider, for example, program p_{ex2} from Fig. 2. If $V = \{y\}$, $P = \{7^{(1)}\}$, and $I = \{1, 3, 4, 5, 6, 7\}$ then $Proj_{(V,P,I)}^*(T_{p_{\text{ex2}}}) = (1, \perp)(3, \perp)(4, \perp)(7, \{y = 1\})$, thus it keeps not only the value of variable y at Line 7 but the path of execution as well. Note that the result of $Proj_{(V,P,I)}^*(T_{p'_{\text{ex2}}})$ is different because of the different path of execution taken in p'_{ex2} .

Using the above functions we can define a unified semantic equivalence relation \mathcal{U} capable of expressing multiple slicing techniques. In the following definition, the roles of the parameters are as follows: S denotes the set of initial states for which the equivalence must hold. This captures the ‘input’ part of the slicing criteria. The set of variables of interest V is common to all slicing criteria. Parameter P , just as in Definitions 12 and 13, contains the points of interest in the trajectory and also captures the ‘iteration count’ component of the criteria. Finally, X captures the ‘trajectory

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;
Program p_{ex3}		Program q_{ex3}	

Fig. 3. Example to capture the difference between KL and non-KL equivalence relations.

requirement'. It is a function that determines which statements must be preserved in the trajectory (even though they have not affected on the variables of the slicing criterion). The domain of X is a pair of sets of statement numbers from two programs. For program p , the set of statement numbers is denoted as \bar{p} .

Definition 14 (Unified equivalence). Given programs p and q , a set of states S , a set of variables V , a set of (line-number, natural number) pairs P , and a set of line-numbers \times set of line-numbers \rightarrow set of line-numbers function X , the unified equivalence, \mathcal{U} , is defined as follows:

$$\begin{aligned}
 & p \mathcal{U}(S, V, P, X) q \\
 & \text{iff} \\
 & \forall \sigma \in S : Proj_{(V, P, X(\bar{p}, \bar{q}))}^*(T_p^\sigma) = Proj_{(V, P, X(\bar{p}, \bar{q}))}^*(T_q^\sigma).
 \end{aligned}$$

In the sequel, we shall adopt the notational convention that \mathcal{S} and \mathcal{D} denote static and dynamic slicing, respectively, a *KL* subscript indicates that a slicing respects the KL1 requirement and that an i subscript indicates that only one occurrence of an instruction in the trajectory is of interest as described below. Figs. 3, 4, and 6 illustrate the differences between these three components of the equivalence relations. (Fig. 3 uses the same example program as given in Fig. 2.)

By instantiating this definition with appropriate parameters we can get two equivalence relations which describe the semantics of the static and KL-dynamic-slicing:

$$\begin{aligned}
 \mathcal{S}(V, n) &= \mathcal{U}(\Sigma, V, \{n\} \times \mathbf{N}, \varepsilon), \\
 \mathcal{D}_{KLi}(\sigma, V, n^{(k)}) &= \mathcal{U}(\{\sigma\}, V, \{n^{(k)}\}, \cap),
 \end{aligned}$$

where Σ is the set of all possible input states, and \mathbf{N} is the set of natural numbers (thus $\{n\} \times \mathbf{N}$ represents all occurrences of instruction n). For every set of line numbers, x and y , $\varepsilon(x, y) = \emptyset$, and \cap denotes the set intersection operation.

That is, in the traditional static formulation for slicing, the set of states of interest is the set of all possible states, Σ . The set of variables, V and the point in the program n are those of the traditional static slicing criterion. For traditional static slicing, the slicing process must preserve the behavior of the program at the point of interest n , and for each possible execution of n (hence $n \times \mathbf{N}$ above). However, the traditional definition of static slicing makes no requirement on the way in which the slice must be computed (hence ε above). On the other hand, KL-slicing does not require a slice to behave for all possible inputs the same way as the original program does, but only for a specific one, σ . Moreover, the point of interest is only one occurrence of a statement, $n^{(k)}$. Contrary to the traditional static slicing, KL-slicing does care about the path of execution in the slice, thus parameter I of $Proj^*$ is $\bar{p} \cap \bar{q}$.

In Fig. 3, the program performs no input, so the relation \mathcal{U} for this program will not be affected by different choices of the first parameter. The set of variables, V_{ex3} is $\{Y\}$, n_{ex3} is set to 7 and k_{ex3} to 1. So for all states σ , $p_{ex3} \mathcal{U}(\sigma, V_{ex3}, \{n_{ex3}^{(k_{ex3})}\}, \varepsilon) q_{ex3}$ but $\neg(p_{ex3} \mathcal{U}(\sigma, V_{ex3}, \{n_{ex3}^{(k_{ex3})}\}, \cap) q_{ex3})$. That is, the fourth parameter of \mathcal{U} , which captures the presence or absence of the KL requirement, is sensitive to the difference in the two programs p_{ex3} and q_{ex3} in Fig. 3. Observe that for both programs, the final value of y is 1, regardless of how the program is executed. However, the trajectory followed by the program q_{ex3} differs from that followed by p_{ex3} even when the two trajectories are restricted to those nodes which occur in both programs; it seems that q_{ex3} arrives at the same answer as p_{ex3} but in a different way.

1 x=1;	1 x=1;
2 while (x<=2) {	2 while (x<=2) {
3 y=1;	3 y=1;
4 if (x==1)	
5 y=2;	
6 z=y;	6 z=y;
7 x++;	7 x++;
8 }	8 }
Program p_{ex4}	Program q_{ex4}
$V_{ex4} = \{y\}, n_{ex4} = 6, k_{ex4} = 2$	

Fig. 4. Example to capture the difference between iteration count aware and iteration count unaware equivalence relations.

1 prev=1;	2 curr=1;	1 prev=1;
2 curr=1;		2 curr=1;
3 i=1;		3 i=1;
4 while (i<n) {		4 while (i<n) {
5 oldc=curr;	5 oldc=curr;	5 oldc=curr;
6 curr=curr+prev;		6 curr=curr+prev;
7 prev=oldc;	7 prev=oldc;	7 prev=oldc;
8 i++;		8 i++;
9 }		9 }
Program p_{ex5}	$V = \{\text{prev}\}, n = 7, k = 1$	$V = \{\text{prev}\}, n = 7, k > 1$

Fig. 5. Example where iteration count is interesting in a static computation.

The requirement that a slice observes this (stringent) requirement for equivalence is similar to the path equivalence studied in the context of program restructuring [44,53]. It is useful in the context of debugging however. When slicing is applied to debugging, it is important that the sliced program faithfully reproduces the behavior that causes a fault to manifest itself as an error. For this reason, program q_{ex3} would not be a useful slice of program p_{ex3} in Fig. 3. In this regard, the KL requirement is important for debugging applications of slicing [50,42]. It may also be important in applications to program comprehension [21,46], because, in these applications, the programmer typically tries to understand the behavior of the original program in terms of the behavior of the slice. However, for other applications, such as testing, reuse, and restructuring [3,13,35], the KL requirement is unimportant because program modification is inherent to these application areas.

To see how the iteration count can affect the meaning of the equivalence preserved by slicing, consider the program in the left-hand column of Fig. 4. In this program, the conditional at line numbers 4 and 5 can only affect the value of y at Line 6 on the first time it is executed. Therefore, choosing the second iteration of this statement in the slicing criterion, will allow the conditional to be deleted. That is, in terms of equivalence, for all states σ , $p_{ex4} \mathcal{U}(\sigma, \{y\}, \{6^{(2)}\}, \cap) q_{ex4}$ and $p_{ex4} \mathcal{U}(\sigma, \{y\}, \{6^{(2)}\}, \varepsilon) q_{ex4}$.

When slicing is applied to debugging, the iteration count will be of interest, but in other applications it is unlikely to be of interest. This is because debugging typically starts when the program fails due to a fault. To locate the fault, a slice can be constructed. Of course, it would be sensible to take into account the iteration count for the statement which reveals the error when constructing the slice; this may reduce the size of the slice, thereby reducing debugging effort.

Although it was (implicitly) introduced as part of Korel and Laski's dynamic slicing criterion, the iteration count concept is independent of whether a slice is to be static or dynamic. The same is true of the KL requirement. This can be seen from the fact that no input was necessary in the two examples used to illustrate the difference in equivalence relations produced by including or excluding these two requirements.

Furthermore, it is possible to find static computations in which the iteration count is an interesting and useful concept. For example, in loop carried dependence, it may take several iterations of a loop in order to propagate a dependence from one point to another. An example of this is the program which computes values in the Fibonacci sequence in Fig. 5. This program performs no input. In the example, the ability to focus upon different iteration counts allows the

1	y=1;	1	y=1;
2	scanf ("%d", &x);		
3	if (x>1)		
4	y=2;		
5	z=y;	5	z=y;
Program p_{ex6}		Program q_{ex6}	
$\sigma_{\text{ex6}} = \langle 1 \rangle, V_{\text{ex6}} = \{y\}, n_{\text{ex6}} = 5, k_{\text{ex6}} = 1$			

Fig. 6. Example to capture the difference between static and dynamic equivalence relations.

dependence structure to be examined in more detail; it becomes possible to see how dependence grows with each loop iteration. In this example, on the first iteration the value of the variable `prev` does not depend on the assignment to `curr` at Line 6, but it does on the second (and subsequent iterations). As this example shows, the concept of an iteration count may be a useful slicing criterion in its own right.

Finally, consider the example in Fig. 6, this illustrates the traditional difference between static and dynamic slicing. That is, for dynamic slicing the input affects the outcome of slicing, while for static slicing, the slice must be correct for all possible inputs. This is the difference between static and dynamic slicing to which most authors [2,12] refer. However, as the preceding discussion shows, there are two other aspects to a dynamic slice: path equivalence (or otherwise) and iteration count sensitivity (or otherwise).

6. Subsumes relation between semantic equivalence relations

The two semantic equivalence relations $\mathcal{S}(V, n)$ and $\mathcal{D}_{KL}(\sigma, V, n^{(k)})$ represent extremes in a space of eight possible equivalence relations. The space has three orthogonal criteria, corresponding to choices of whether or not to include i or KL and whether a slice is static (\mathcal{S}) or dynamic (\mathcal{D}). This means that there are six new intervening equivalence relations (and thus, three additional pairs of extremes) resulting from the other possible parameterizations of the unified equivalence:

$$\begin{aligned}
\mathcal{S}_i(V, n^{(k)}) &= \mathcal{U}(\Sigma, V, \{n^{(k)}\}, \varepsilon), \\
\mathcal{D}(\sigma, V, n) &= \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \varepsilon), \\
\mathcal{D}_i(\sigma, V, n^{(k)}) &= \mathcal{U}(\{\sigma\}, V, \{n^{(k)}\}, \varepsilon), \\
\mathcal{S}_{KL}(V, n) &= \mathcal{U}(\Sigma, V, \{n\} \times \mathbf{N}, \cap), \\
\mathcal{S}_{KLi}(V, n^{(k)}) &= \mathcal{U}(\Sigma, V, \{n^{(k)}\}, \cap), \\
\mathcal{D}_{KL}(\sigma, V, n) &= \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \cap).
\end{aligned}$$

These six equivalence relations capture the semantic property of six new, hitherto undiscussed slicing methods. For example, $\mathcal{S}_i(V, n^{(k)})$ slicing takes as a slicing criterion a set of variables V and a single occurrence of statement n from the execution trajectory.

The eight equivalence relations \mathcal{S} , \mathcal{S}_i , \mathcal{D} , \mathcal{D}_i , \mathcal{S}_{KL} , \mathcal{S}_{KLi} , \mathcal{D}_{KL} and \mathcal{D}_{KLi} represent, in fact, classes of equivalence relations, since they are parameterized with σ, V, n and k (even though not all of the relations make use of all four). Denoting an equivalence relation \approx , it is possible to define a subsumption relationship, $\approx_A \subseteq \approx_B$ between these classes. Formally, as in the following definition, these equivalence relations are parameterized by σ, V, n and k . However, in the sequel, the superscript (σ, V, n, k) is omitted when it is clear from context.

Definition 15 (Subsumes relation). Equivalence relation \approx_A subsumes equivalence relation \approx_B iff

$$\forall \sigma, V, n, k : \approx_B^{(\sigma, V, n, k)} \subseteq \approx_A^{(\sigma, V, n, k)}$$

or equivalently,

$$\forall p, q, \sigma, V, n, k : (p, q) \in \approx_B^{(\sigma, V, n, k)} \Rightarrow (p, q) \in \approx_A^{(\sigma, V, n, k)}.$$

This subsumes relation is a partial ordering of equivalence relations, since it is defined with the help of the subset relation, which is itself a partial ordering (i.e., reflexive, transitive and antisymmetric). Fig. 7 presents the lattice of the

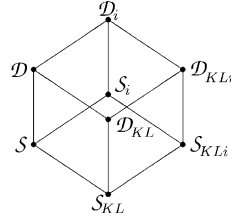


Fig. 7. Subsumes relationship between equivalence relations.

subsumes relation for $S, S_i, D, D_i, S_{KL}, S_{KL_i}, D_{KL}$ and D_{KL_i} (e.g., S is subsumed by D). The following theorem proves the correctness of the diagram in Fig. 7: in other words, there are no superfluous and no missing edges in the lattice.

Theorem 16. *The lattice shown in Fig. 7 is correct: two equivalence relations are connected in the diagram iff they are in subsumes relation.*

The proof makes use of four lemmas and their corollaries. The first lemma is used to prove the “if” direction and the latter three the “only if” direction.

Lemma 17. *Given sets of initial states S_1 and S_2 , sets of variables V_1 and V_2 , sets of points of interests P_1 and P_2 and functions of pairs of line number sets X_1 and X_2 such that*

$$S_1 \subseteq S_2, \quad V_1 \subseteq V_2, \quad P_1 \subseteq P_2 \quad \text{and} \quad \forall p, q : X_1(\bar{p}, \bar{q}) \subseteq X_2(\bar{p}, \bar{q})$$

then

$$\mathcal{U}(S_2, V_2, P_2, X_2) \subseteq \mathcal{U}(S_1, V_1, P_1, X_1).$$

Proof. Let $(p, q) \in \mathcal{U}(S_2, V_2, P_2, X_2)$. By definition, this is equivalent to $\forall \sigma \in S_2$:

$$Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(T_p^\sigma) = Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(T_q^\sigma).$$

As $S_1 \subseteq S_2$, it follows that $\forall \sigma \in S_1$:

$$Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(T_p^\sigma) = Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(T_q^\sigma).$$

By inlining the definition of $Proj^*$, this is equivalent to $\forall \sigma \in S_1$:

$$\bigoplus_{i=1}^{l_2^p} Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(n_i^{p(k_i^p)}, \sigma_i^p) = \bigoplus_{i=1}^{l_2^q} Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(n_i^{q(k_i^q)}, \sigma_i^q).$$

Corresponding prefixes from the above equality are also equal. As points of interest P_1 is a subset of P_2 , the location of the last occurrence in the trajectory of a point from P_1 must occur before the last occurrence of a point from P_2 . More formally, $l_1^p \leq l_2^p$ and $l_1^q \leq l_2^q$; thus, the above equality is maintained when $\bigoplus_{i=1}^{l_2^p}$ and $\bigoplus_{i=1}^{l_2^q}$ are replaced with $\bigoplus_{i=1}^{l_1^p}$ and $\bigoplus_{i=1}^{l_1^q}$. Consequently, $\forall \sigma \in S_1$:

$$\bigoplus_{i=1}^{l_1^p} Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(n_i^{p(k_i^p)}, \sigma_i^p) = \bigoplus_{i=1}^{l_1^q} Proj_{(V_2, P_2, X_2(\bar{p}, \bar{q}))}^*(n_i^{q(k_i^q)}, \sigma_i^q).$$

This means that the projections of those state trajectory elements, which are not projected to λ , are pairwise equal in the two trajectories. Thus, any corresponding subsequence of these elements must be pairwise equal. In particular, as $V_1 \subseteq V_2, P_1 \subseteq P_2$, and $\forall p, q : X_1(\bar{p}, \bar{q}) \subseteq X_2(\bar{p}, \bar{q})$, restricting the sequences to variables in V_1 at points in P_1

where the instruction from $X_1(\bar{p}, \bar{q})$ are preserved must also be equivalent. Thus, $\forall \sigma \in S_1$:

$$\bigoplus_{i=1}^{l_1^p} \text{Proj}_{(V_1, P_1, X_1(\bar{p}, \bar{q}))}^*(n_i^{p(k_i^p)}, \sigma_i^p) = \bigoplus_{i=1}^{l_1^q} \text{Proj}_{(V_1, P_1, X_1(\bar{p}, \bar{q}))}^*(n_i^{q(k_i^q)}, \sigma_i^q),$$

which, by definition, is equivalent to $\forall \sigma \in S_1$:

$$\text{Proj}_{(V_1, P_1, X_1(\bar{p}, \bar{q}))}^*(T_p^\sigma) = \text{Proj}_{(V_1, P_1, X_1(\bar{p}, \bar{q}))}^*(T_q^\sigma),$$

which, again by definition, is equivalent to $(p, q) \in \mathcal{U}(S_1, V_1, P_1, X_1)$, as required. \square

The existence of each of the 12 subsumption relationships between the equivalence relations shown in Fig. 7 follows from Lemma 17, as proven by the corollary below.

Corollary 18. *The equivalence relations connected in the diagram are in subsumes relation.*

Proof. The proof of each case considers each of the four attributes of the unified equivalence operator $\mathcal{U}(S, V, P, X)$ independently. The relevant relationships are as follows: For S , $\{\sigma\} \subseteq \Sigma$, for V , all 12 use the same argument (which is thus ignored below), for P , $n^{(k)} \subseteq (\{n\} \times \mathbf{N})$, and for X , $\forall p, q : \varepsilon(\bar{p}, \bar{q}) = \emptyset \subseteq \cap(\bar{p}, \bar{q})$. The table below shows how Lemma 17 implies all of the cases ($\varepsilon(\bar{p}, \bar{q})$ and $\cap(\bar{p}, \bar{q})$ are abbreviated as ε and \cap).

Subsumption	Lemma 17 requirement		
	S	P	X
$\mathcal{D}_{KLi} \subseteq \mathcal{D}_i$	$\{\sigma\} \subseteq \{\sigma\}$	$n^{(k)} \subseteq n^{(k)}$	$\varepsilon \subseteq \cap$
$\mathcal{D}_{KL} \subseteq \mathcal{D}_{KLi}$	$\{\sigma\} \subseteq \{\sigma\}$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\cap \subseteq \cap$
$\mathcal{D}_{KL} \subseteq \mathcal{D}$	$\{\sigma\} \subseteq \{\sigma\}$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \cap$
$\mathcal{D} \subseteq \mathcal{D}_i$	$\{\sigma\} \subseteq \{\sigma\}$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \varepsilon$
$\mathcal{S}_{KLi} \subseteq \mathcal{D}_{KLi}$	$\{\sigma\} \subseteq \Sigma$	$n^{(k)} \subseteq n^{(k)}$	$\cap \subseteq \cap$
$\mathcal{S}_{KLi} \subseteq \mathcal{S}_i$	$\Sigma \subseteq \Sigma$	$n^{(k)} \subseteq n^{(k)}$	$\varepsilon \subseteq \cap$
$\mathcal{S}_{KL} \subseteq \mathcal{D}_{KL}$	$\{\sigma\} \subseteq \Sigma$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\cap \subseteq \cap$
$\mathcal{S}_{KL} \subseteq \mathcal{S}_{KLi}$	$\Sigma \subseteq \Sigma$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\cap \subseteq \cap$
$\mathcal{S}_{KL} \subseteq \mathcal{S}$	$\Sigma \subseteq \Sigma$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \cap$
$\mathcal{S}_i \subseteq \mathcal{D}_i$	$\{\sigma\} \subseteq \Sigma$	$n^{(k)} \subseteq n^{(k)}$	$\varepsilon \subseteq \varepsilon$
$\mathcal{S} \subseteq \mathcal{D}$	$\{\sigma\} \subseteq \Sigma$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \varepsilon$
$\mathcal{S} \subseteq \mathcal{S}_i$	$\Sigma \subseteq \Sigma$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \varepsilon$

\square

The proof of the “only if” direction involves showing that there are no “missing” edges in Fig. 7. To be more specific, the following nine pairs of slicing equivalence relations are shown incomparable (denoted by “ $\approx_A \not\subseteq \approx_B$ ”): $(\mathcal{D} \not\subseteq \not\subseteq \mathcal{D}_{KLi})$, $(\mathcal{D} \not\subseteq \not\subseteq \mathcal{S}_i)$, $(\mathcal{D}_{KLi} \not\subseteq \not\subseteq \mathcal{S}_i)$, $(\mathcal{S}_i \not\subseteq \not\subseteq \mathcal{D}_{KL})$, $(\mathcal{D}_{KL} \not\subseteq \not\subseteq \mathcal{S})$, $(\mathcal{D}_{KL} \not\subseteq \not\subseteq \mathcal{S}_{KLi})$, $(\mathcal{S} \not\subseteq \not\subseteq \mathcal{S}_{KLi})$, $(\mathcal{D} \not\subseteq \not\subseteq \mathcal{S}_{KLi})$, and $(\mathcal{D}_{KLi} \not\subseteq \not\subseteq \mathcal{S})$.

Proving the incomparability of two equivalence relations, \approx_A and \approx_B , requires showing that (for some σ, V, n and k that parameterize the relation) neither relation subsumes the other. This is done by showing two things. First, that there exist programs p and q such that $(p, q) \in \approx_A$ but $(p, q) \notin \approx_B$ and then by showing that there exist programs p' and q' such that $(p', q') \in \approx_B$ but $(p', q') \notin \approx_A$.

The following three lemmas introduce examples used to show the necessary incomparabilities.

Lemma 19. $\mathcal{S} \not\subseteq \mathcal{D}_{KLi}$.

Proof. For $p_{\text{ex3}}, q_{\text{ex3}}, V_{\text{ex3}}, n_{\text{ex3}}, k_{\text{ex3}}$ as given in Fig. 3, and for any input state σ the following hold:

$$(p_{\text{ex3}}, q_{\text{ex3}}) \in \mathcal{S}(V_{\text{ex3}}, n_{\text{ex3}}) \text{ and } (p_{\text{ex3}}, q_{\text{ex3}}) \notin \mathcal{D}_{KLi}(\sigma, V_{\text{ex3}}, n_{\text{ex3}}^{(k_{\text{ex3}})}).$$

First, as the program is unaffected by its input, $(p_{\text{ex}3}, q_{\text{ex}3}) \in \mathcal{S}(V_{\text{ex}3}, n_{\text{ex}3})$ for all input states σ because $\forall \sigma \in \Sigma$:

$$\text{Proj}_{(V_{\text{ex}3}, \{n_{\text{ex}3}\} \times \mathbf{N}, \emptyset)}^*(T_{p_{\text{ex}3}}^\sigma) = \text{Proj}_{(V_{\text{ex}3}, \{n_{\text{ex}3}\} \times \mathbf{N}, \emptyset)}^*(T_{q_{\text{ex}3}}^\sigma) = (7, \{\mathcal{Y} = 1\}).$$

Second $(p_{\text{ex}3}, q_{\text{ex}3}) \notin \mathcal{D}_{KLi}(\sigma, V_{\text{ex}3}, n_{\text{ex}3}^{(k_{\text{ex}3})})$ because, as shown in Section 4, KL1 is violated. Thus, combined $\mathcal{S}(V_{\text{ex}3}, n_{\text{ex}3}) \not\subseteq \mathcal{D}_{KLi}(\sigma, V_{\text{ex}3}, n_{\text{ex}3}^{(k_{\text{ex}3})})$ and, in general, $\mathcal{S} \not\subseteq \mathcal{D}_{KLi}$. \square

Five corollaries to Lemma 19 are used in the proof of Theorem 16. They are given as Eqs. (2)–(6) in Fig. 8. A detailed proof of the first is given below. The other proofs are similar. Note that Eq. (6) follows from Eqs. (2) and (4).

Corollary 20. $\mathcal{S}_i \not\subseteq \mathcal{D}_{KLi}$ (Eq. (2) of Fig. 8).

Proof. From Lemma 19 we know that $(p_{\text{ex}3}, q_{\text{ex}3}) \in \mathcal{S}(V_{\text{ex}3}, n_{\text{ex}3})$ and $(p_{\text{ex}3}, q_{\text{ex}3}) \notin \mathcal{D}_{KLi}(\sigma, V_{\text{ex}3}, n_{\text{ex}3}^{(k_{\text{ex}3})})$. Additionally, Lemma 17 implies $\mathcal{S}(V_{\text{ex}3}, n_{\text{ex}3}) \subseteq \mathcal{S}_i(V_{\text{ex}3}, n_{\text{ex}3}^{(k_{\text{ex}3})})$, from which follows $(p_{\text{ex}3}, q_{\text{ex}3}) \in \mathcal{S}_i(V_{\text{ex}3}, n_{\text{ex}3}^{(k_{\text{ex}3})})$. Thus, it must be the case that $\mathcal{S}_i(V_{\text{ex}3}, n_{\text{ex}3}) \not\subseteq \mathcal{D}_{KLi}(\sigma, V_{\text{ex}3}, n_{\text{ex}3}^{(k_{\text{ex}3})})$ or simply $\mathcal{S}_i \not\subseteq \mathcal{D}_{KLi}$. \square

Lemma 21. $\mathcal{S}_{KLi} \not\subseteq \mathcal{D}$.

Proof. For $p_{\text{ex}4}, q_{\text{ex}4}, V_{\text{ex}4}, n_{\text{ex}4}, k_{\text{ex}4}$ as given in Fig. 4, and for any input state σ the following hold:

$$(p_{\text{ex}4}, q_{\text{ex}4}) \in \mathcal{S}_{KLi}(V_{\text{ex}4}, n_{\text{ex}4}^{(k_{\text{ex}4})}) \quad \text{and} \quad (p_{\text{ex}4}, q_{\text{ex}4}) \notin \mathcal{D}(\sigma, V_{\text{ex}4}, n_{\text{ex}4}).$$

First, $(p_{\text{ex}4}, q_{\text{ex}4}) \in \mathcal{S}_{KLi}(V_{\text{ex}4}, n_{\text{ex}4}^{(k_{\text{ex}4})})$ since $\forall \sigma \in \Sigma$:

$$\begin{aligned} & \text{Proj}_{(V_{\text{ex}4}, \{n_{\text{ex}4}^{(k_{\text{ex}4})}\}, \overline{p_{\text{ex}4} \cap q_{\text{ex}4}})}^*(T_{p_{\text{ex}4}}^\sigma) \\ &= \text{Proj}_{(V_{\text{ex}4}, \{n_{\text{ex}4}^{(k_{\text{ex}4})}\}, \overline{p_{\text{ex}4} \cap q_{\text{ex}4}})}^*(T_{q_{\text{ex}4}}^\sigma) \\ &= (1, \perp)(2, \perp)(3, \perp)(6, \perp)(7, \perp)(2, \perp)(3, \perp)(6, \{\mathcal{Y} = 1\}). \end{aligned}$$

Second $(p_{\text{ex}4}, q_{\text{ex}4}) \notin \mathcal{D}(\sigma, V_{\text{ex}4}, n_{\text{ex}4})$ because

$$\text{Proj}_{(V_{\text{ex}4}, \{n_{\text{ex}4}\} \times \mathbf{N}, \emptyset)}^*(T_{p_{\text{ex}4}}^\sigma) = (6, \{\mathcal{Y} = 2\})(6, \{\mathcal{Y} = 1\}),$$

but

$$\text{Proj}_{(V_{\text{ex}4}, \{n_{\text{ex}4}\} \times \mathbf{N}, \emptyset)}^*(T_{q_{\text{ex}4}}^\sigma) = (6, \{\mathcal{Y} = 1\})(6, \{\mathcal{Y} = 1\}).$$

Thus, combined $\mathcal{S}_{KLi}(V_{\text{ex}4}, n_{\text{ex}4}^{(k_{\text{ex}4})}) \not\subseteq \mathcal{D}(\sigma, V_{\text{ex}4}, n_{\text{ex}4})$ and, in general, $\mathcal{S}_{KLi} \not\subseteq \mathcal{D}$. \square

As with Lemma 19, five corollaries to Lemma 21 are given as Eqs. (8)–(12) in Fig. 8. Note that Eq. (12) follows from Eqs. (9) and (11).

Lemma 22. $\mathcal{D}_{KL} \not\subseteq \mathcal{S}_i$.

Proof. For $p_{\text{ex}6}, q_{\text{ex}6}, \sigma_{\text{ex}6}, V_{\text{ex}6}, n_{\text{ex}6}, k_{\text{ex}6}$ as given in Fig. 6, the following hold:

$$(p_{\text{ex}6}, q_{\text{ex}6}) \in \mathcal{D}_{KL}(\sigma_{\text{ex}6}, V_{\text{ex}6}, n_{\text{ex}6}) \quad \text{and} \quad (p_{\text{ex}6}, q_{\text{ex}6}) \notin \mathcal{S}_i(V_{\text{ex}6}, n_{\text{ex}6}^{(k_{\text{ex}6})}).$$

First, $(p_{\text{ex}6}, q_{\text{ex}6}) \in \mathcal{D}_{KL}(\sigma_{\text{ex}6}, V_{\text{ex}6}, n_{\text{ex}6})$ since

$$\begin{aligned} & \text{Proj}_{(V_{\text{ex}6}, \{n_{\text{ex}6}\} \times \mathbf{N}, \overline{p_{\text{ex}6} \cap q_{\text{ex}6}})}^*(T_{p_{\text{ex}6}}^{\sigma_{\text{ex}6}}) \\ &= \text{Proj}_{(V_{\text{ex}6}, \{n_{\text{ex}6}\} \times \mathbf{N}, \overline{p_{\text{ex}6} \cap q_{\text{ex}6}})}^*(T_{q_{\text{ex}6}}^{\sigma_{\text{ex}6}}) \\ &= (1, \perp)(5, \{\mathcal{Y} = 1\}). \end{aligned}$$

Second $(p_{\text{ex}6}, q_{\text{ex}6}) \notin \mathcal{S}_i(V_{\text{ex}6}, n_{\text{ex}6}^{(k_{\text{ex}6})})$ because $\text{Proj}_{(V_{\text{ex}6}, \{n_{\text{ex}6}^{(k_{\text{ex}6})}\}, \emptyset)}^*(T_{p_{\text{ex}6}}^{\sigma^*}) = (5, \{\mathcal{Y} = 2\})$ but $\text{Proj}_{(V_{\text{ex}6}, \{n_{\text{ex}6}^{(k_{\text{ex}6})}\}, \emptyset)}^*(T_{q_{\text{ex}6}}^{\sigma^*}) = (5, \{\mathcal{Y} = 1\})$, where $\sigma^* = \langle 2 \rangle$. Thus, combined $\mathcal{S}_i(V_{\text{ex}6}, n_{\text{ex}6}^{(k_{\text{ex}6})}) \not\subseteq \mathcal{D}_{KL}(\sigma_{\text{ex}6}, V_{\text{ex}6}, n_{\text{ex}6})$, and in general, $\mathcal{S}_i \not\subseteq \mathcal{D}_{KL}$. \square

Eq.	Result/Corollaries	Justification
1	$S \not\subseteq \mathcal{D}_{KLi}$ by Lemma 19	
2	$S_i \not\subseteq \mathcal{D}_{KLi}$ as $S \subseteq S_i$	implies $(p_{ex3}, q_{ex3}) \in S_i(V_{ex3}, n_{ex3}^{(k_{ex3})})$
3	$\mathcal{D} \not\subseteq \mathcal{D}_{KLi}$ as $S \subseteq \mathcal{D}$	implies $(p_{ex3}, q_{ex3}) \in \mathcal{D}(\sigma, V_{ex3}, n_{ex3})$
4	$S \not\subseteq \mathcal{D}_{KL}$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}_{KLi}$	implies $(p_{ex3}, q_{ex3}) \notin \mathcal{D}_{KL}(\sigma, V_{ex3}, n_{ex3})$
5	$S \not\subseteq S_{KLi}$ as $S_{KLi} \subseteq \mathcal{D}_{KLi}$	implies $(p_{ex3}, q_{ex3}) \notin S_{KLi}(V_{ex3}, n_{ex3}^{(k_{ex3})})$
6	$S_i \not\subseteq \mathcal{D}_{KL}$ as $(p_{ex3}, q_{ex3}) \in S_i(V_{ex3}, n_{ex3}^{(k_{ex3})})$ and $(p_{ex3}, q_{ex3}) \notin \mathcal{D}_{KL}(\sigma, V_{ex3}, n_{ex3})$	
7	$S_{KLi} \not\subseteq \mathcal{D}$ by Lemma 21	
8	$S_i \not\subseteq \mathcal{D}$ as $S_{KLi} \subseteq S_i$	implies $(p_{ex4}, q_{ex4}) \in S_i(V_{ex4}, n_{ex4}^{(k_{ex4})})$
9	$\mathcal{D}_{KLi} \not\subseteq \mathcal{D}$ as $S_{KLi} \subseteq \mathcal{D}_{KLi}$	implies $(p_{ex4}, q_{ex4}) \in \mathcal{D}_{KLi}(\sigma, V_{ex4}, n_{ex4}^{(k_{ex4})})$
10	$S_{KLi} \not\subseteq \mathcal{D}_{KL}$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}$	implies $(p_{ex4}, q_{ex4}) \notin \mathcal{D}_{KL}(\sigma, V_{ex4}, n_{ex4})$
11	$S_{KLi} \not\subseteq S$ as $S \subseteq \mathcal{D}$	implies $(p_{ex4}, q_{ex4}) \notin S(V_{ex4}, n_{ex4})$
12	$\mathcal{D}_{KLi} \not\subseteq S$ as $(p_{ex4}, q_{ex4}) \in \mathcal{D}_{KLi}(\sigma, V_{ex4}, n_{ex4}^{(k_{ex4})})$ and $(p_{ex4}, q_{ex4}) \notin S(V_{ex4}, n_{ex4})$	
13	$\mathcal{D}_{KL} \not\subseteq S_i$ by Lemma 22	
14	$\mathcal{D} \not\subseteq S_i$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}$	implies $(p_{ex6}, q_{ex6}) \in \mathcal{D}(\sigma_{ex6}, V_{ex6}, n_{ex6})$
15	$\mathcal{D}_{KLi} \not\subseteq S_i$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}_{KLi}$	implies $(p_{ex6}, q_{ex6}) \in \mathcal{D}_{KLi}(\sigma_{ex6}, V_{ex6}, n_{ex6}^{(k_{ex6})})$
16	$\mathcal{D}_{KL} \not\subseteq S$ as $S \subseteq S_i$	implies $(p_{ex6}, q_{ex6}) \notin S(V_{ex6}, n_{ex6})$
17	$\mathcal{D}_{KL} \not\subseteq S_{KLi}$ as $S_{KLi} \subseteq S_i$	implies $(p_{ex6}, q_{ex6}) \notin S_{KLi}(V_{ex6}, n_{ex6}^{(k_{ex6})})$
18	$\mathcal{D} \not\subseteq S_{KLi}$ as $(p_{ex6}, q_{ex6}) \in \mathcal{D}(\sigma_{ex6}, V_{ex6}, n_{ex6})$ and $(p_{ex6}, q_{ex6}) \notin S_{KLi}(V_{ex6}, n_{ex6}^{(k_{ex6})})$	

Fig. 8. Corollaries to Lemmas 19, 21, and 22.

Incompatibility	Follows from
$\mathcal{D} \not\subseteq \mathcal{D}_{KLi}$	(3) and (9)
$\mathcal{D} \not\subseteq S_i$	(14) and (8)
$\mathcal{D}_{KLi} \not\subseteq S_i$	(15) and (2)
$S_i \not\subseteq \mathcal{D}_{KL}$	(6) and (13)
$\mathcal{D}_{KL} \not\subseteq S$	(16) and (4)
$\mathcal{D}_{KL} \not\subseteq S_{KLi}$	(17) and (10)
$S \not\subseteq S_{KLi}$	(5) and (11)
$\mathcal{D} \not\subseteq S_{KLi}$	(18) and (7)
$\mathcal{D}_{KLi} \not\subseteq S$	(12) and (1)

As with Lemmas 19 and 21, five corollaries to Lemma 22 are given as Eqs. (14)–(18) in Fig. 8. Note that Eq. (18) follows from Eqs. (14) and (17). Using Lemma 17 and Eqs. (1)–(18) from Fig. 8, it is now possible to prove Theorem 16, which is restated.

Theorem 16. *The lattice shown in Fig. 7 is correct: two equivalence relations are connected in the diagram iff they are in subsumes relation.*

Proof. The “if” direction is proven in Corollary 18, while the relations given in Fig. 8 are sufficient to prove all of the cases in the “only if” direction as summarized in the following table:

Theorem 16 formalizes the relationship between the eight slicing equivalence relations depicted in Fig. 7.

The importance of this result is that it shows that the dynamic slicing criterion contains two, previously un-studied criteria: path sensitivity and iteration count sensitivity. The presence of these criteria make the subsumption relationship between forms and static and dynamic slicing more involved than previously thought. As discussed in this section, the

new criteria may also find useful application in their own right. For example, they allow those working on building slicers to better understand the tradeoffs between slicing precision and computation time. They also allow slice users to understand and then choose the slicing definition that is more appropriate for a given problem.

7. Allowable slices according to a slicing technique

The previous section established a unified equivalence relation which allowed the study of the relationships between the semantic properties of the eight forms of slicing considered in this paper. However, in addition to studying the semantic properties of slicing, we are also interested in the relation between the sets of allowable slices, not merely the relationships between the semantic equivalence relations.

In order to achieve this we will need to take account of the syntactic ordering relation as well as the semantic equivalence relation. We call the combination of the semantic equivalence and the syntactic ordering relations a ‘slicing technique’, since it captures the set of slices which can be produced according to a particular form of slicing. That is, those slices which can be produced by a particular slicing technique.

Informally, a slicing technique s_1 subsumes another slicing technique s_2 iff all slices of an arbitrary program with respect to any given slicing criterion according to s_2 are valid slices with respect to the same slicing criterion according to s_1 . This informal definition is formalized below.

Definition 23 (*Subsumes relation of slicing techniques*). Given syntactic ordering \lesssim and semantic equivalence relations \approx_A and \approx_B , both parameterized by σ, V, n and k , (\lesssim, \approx_A) -slicing subsumes (\lesssim, \approx_B) -slicing iff

$$\forall p, \sigma, V, n, k : \mathbb{S}_p(\lesssim, \approx_B^{(\sigma, V, n, k)}) \subseteq \mathbb{S}_p(\lesssim, \approx_A^{(\sigma, V, n, k)}),$$

where $\mathbb{S}_p(\lesssim, \approx) = \{q \mid q \approx p \text{ and } q \lesssim p\}$ is the set of all possible slices of program p for given projection (\lesssim, \approx) .

Example. $(\sqsubseteq, \mathcal{S}_i)$ -slicing subsumes $(\sqsubseteq, \mathcal{S})$ -slicing as every $(\sqsubseteq, \mathcal{S}(V, n))$ projection of a given program p is a $(\sqsubseteq, \mathcal{S}_i(V, n^{(k)}))$ projection of p as well, i.e. $\mathbb{S}_p(\sqsubseteq, \mathcal{S}(V, n)) \subseteq \mathbb{S}_p(\sqsubseteq, \mathcal{S}_i(V, n^{(k)}))$, for any given V, n and k . On the contrary, $(\sqsubseteq, \mathcal{S})$ -slicing does not subsume $(\sqsubseteq, \mathcal{S}_i)$ -slicing. This is illustrated in Fig. 4 where q_{ex4} is a $(\sqsubseteq, \mathcal{S}_i(\{Y\}, 6^{(2)}))$ projection of p_{ex4} but is not a $(\sqsubseteq, \mathcal{S}(\{Y\}, 6))$ projection.

This definition of subsumption relationship between slicing methods is closely related to the subsumption relationship defined for semantic equivalence relations. Namely, if \approx_A subsumes \approx_B then (\lesssim, \approx_A) -slicing subsumes (\lesssim, \approx_B) -slicing as well. This is stated and proven in the following lemma:

Lemma 24. *Given semantic equivalence relations \approx_A and \approx_B , both parameterized with σ, V, n and k , if \approx_A subsumes \approx_B then (\lesssim, \approx_A) -slicing subsumes (\lesssim, \approx_B) -slicing, for any syntactic ordering \lesssim .*

Proof. Let p be a program, \lesssim a syntactic ordering and $q \in \mathbb{S}_p(\lesssim, \approx_B^{(\sigma, V, n, k)})$ (for any given σ, V, n and k). Then, by definition, $q \approx_B^{(\sigma, V, n, k)} p$ and $q \lesssim p$. Since \approx_A subsumes \approx_B , implies $q \approx_A^{(\sigma, V, n, k)} p$ holds as well, which means that $q \in \mathbb{S}_p(\lesssim, \approx_A^{(\sigma, V, n, k)})$ as required. \square

The above lemma can be used to prove the correctness of the diagram depicted in Fig. 9, which mirrors the diagram from Fig. 7. Fig. 9 shows the relations between slicing techniques (as opposed to equivalence relations) that all use the traditional syntactic ordering \sqsubseteq and the equivalence relations $\mathcal{S}, \mathcal{S}_i, \mathcal{D}, \mathcal{D}_i, \mathcal{S}_{KL}, \mathcal{S}_{KL_i}, \mathcal{D}_{KL}$ and \mathcal{D}_{KL_i} . The correctness of this diagram is shown in the following theorem.

Theorem 25. *The lattice shown in Fig. 9 is correct: two slicing techniques are connected in the diagram iff they are in subsumes relation.*

Proof. “If”: The correctness of each of the subsumption relations shown in Fig. 9 follows from Theorem 16 and Lemma 24 where \lesssim is \sqsubseteq .

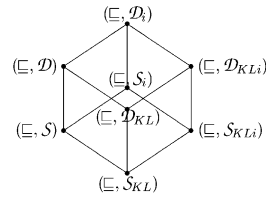


Fig. 9. Subsumes relation between slicing techniques.

“Only if”: The argument that no edges are missing from the diagram follows from the “only if” argument of Theorem 16 and the observation that the examples in Figs. 3, 4, and 6 are constructed so that $q_{\text{ex}3} \sqsubseteq p_{\text{ex}3}$, $q_{\text{ex}4} \sqsubseteq p_{\text{ex}4}$ and $q_{\text{ex}6} \sqsubseteq p_{\text{ex}6}$. \square

These results reveal the intricate structure which underlies the definitions of dynamic slicing. They also highlight the presence of, previously implicit, criteria for slicing, buried in the accepted definitions of dynamic slice. This is both theoretically interesting and practically important. For example, the practical value of this result lies in the potential for new applications embodied in the new slicing criteria.

8. Related work

Program slicing was introduced by Mark Weiser in 1979 as a static program analysis and extraction technique [59]. Weiser originally had many applications of slicing in mind. Most of these and many others have been developed in the literature which followed. One of the primary initial goals of slicing was to assist with debugging. Weiser noticed [60] that programmers naturally form program slices, mentally, when they debug and understand programs. Therefore, it seemed natural to attempt to automate this process to improve the efficiency of the debugging process. Lyle and Weiser [50] further developed the theme of slicing as an aid to debugging and this remained a primary application of slicing for some time.

In this initial work on slicing, the algorithms used for slicing were based upon data flow equations [61]. However, in 1984 Ottenstein and Ottenstein [52] showed how the program dependence graph (PDG) could be used to turn slicing into a graph reachability question. Ottenstein and Ottenstein’s formulation was an intraprocedural one, however, and it was not clear how to cater for the calling context problem using the PDG. In 1998, Horwitz et al. [40,41] introduced the system dependence graph (SDG), an extension of the PDG which could allow for efficient computation of interprocedural slices, while respecting calling context. Since then, the majority of slicing algorithms have been SDG-based including those implemented in Grammatech’s commercial program slicing tool, CodeSurfer [28].

Slices produced by the algorithm of Horwitz et al. are not necessarily executable programs [40]. The problem arises when different calling context require different subsets of a procedure’s input parameters. Horwitz et al. propose two methods to transform non-executable SDG slices into executable programs. The first creates a copy of a procedure for each calling context requiring different subsets of the input parameters. Unfortunately, such a ‘slice’ does not satisfy the syntactic ordering requirement of Definition 4. The second option, later refined by Binkley [6], iteratively includes intraprocedural slices taken with respect to actual parameters until all calls to a procedure include the same parameters. This approach yields static (S) slices in the projection framework that also satisfy the KL requirement of being execution path preserving.

In 1988 Korel and Laski [45] observed that slices might be more useful as a debugging aid, if they could be constructed dynamically, taking into account the execution characteristics which led to the observation of erroneous behavior. If slices are constructed dynamically then they are guaranteed to be no larger than their static counter parts and may be smaller. Korel and Laski’s algorithm for constructing dynamic slices was a modified version of Weiser’s data flow equations.

In 1990 Agrawal and Horgan [2] introduced two algorithms for constructing dynamic slices based on the PDG. (They actually propose four algorithms, but two only impact performance and not the slices computed.) These two algorithms differ in ways made clear with the benefit of the theory introduced herein. In terms of the equivalence relations from Section 5, Agrawal and Horgan’s first algorithm preserves $\mathcal{D}(\sigma, V, n)$ while their second algorithm and that of Korel and Laski preserves $\mathcal{D}_{KL_i}(\sigma, V, n^{(k)})$.

De Lucia et al. [21,12] introduced a concept called conditioned slicing. The conditioned slicing criterion augments the traditional static criterion with a condition. The slicing process needs only to preserve the effect of the original program on the variables of interest if the condition is satisfied. By choosing this condition to be simply the constant predicate ‘true’, the definition of conditioned slicing becomes that of static slicing and by making it a conjunction of equalities, it is possible to mimic the effect of an input sequence.

These observations have led several authors to observe that conditioned slicing “subsumes” static and dynamic slicing [12,19,23]. However, this use of the term “subsumes” differs from the one used herein. It is based on the expressive power of the slicing criterion (a slicing method $S1$ subsumes a slicing method $S2$ if any $S2$ slicing criterion can be expressed as an $S1$ slicing criterion). The subsumption relations introduced herein are based on the semantics preserved by slicing and set of programs that qualify as slices. Informally, it appears that conditioned slicing subsumes static slicing and is subsumed by dynamic slicing. The same is true for the iteration aware and execution path preserving variants.

Finally, the two additional criteria identified for dynamic slicing could be used to augment any other form of slicing criterion, including the conditioned slicing formulation. Therefore, it would be possible to speak of an iteration count aware conditioned slicing criterion, for example. Showing that these possibilities exist is one of the theoretical contributions of the present paper.

Other theoretical work has attempted to lay the foundations of slicing. However, this previous work has been primarily concerned with static slicing. Reps and Yang [55] show that the PDG is adequate as a representation of program semantics, allowing it to be used in slicing and related program analyses without loss of semantic information. Reps [54] shows how interprocedural-slicing can be formulated as a graph reachability problem. Cartwright and Felleisen [15] show that the PDG semantics is a lazy semantics, because of the demand driven nature of the representation, while Giacobazzi and Mastroeni [26] present a transfinite semantics to attempt to capture the behavior of static slicing. Harman et al. show the slicing is lazy in the presence of errors [36]. Weiser [59] observed that his slicing algorithm was not dataflow minimal and speculated on the question of whether dataflow minimal slices were computable. Danicic showed how this problem could be reformulated as a theorem about unfolding [18] while Laurence et al. [49] show how the problem can be expressed in terms of program schematology.

However, all this work has been concerned with static slicing. There has been very little formal theoretical analysis of the properties of dynamic slicing. The closest prior work to that in the present paper is the previous work of Venkatesh [58] and that by Harman et al. [30]. 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). Harman et al. introduced the projection theory used in this paper to analyze dynamic slicing. However, they used this theory to explain the difference between syntax-preserving and amorphous slicing, and did not address the issue of dynamic slicing.

Venkatesh also 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. Contamination propagates through the semantic description of a program in much the same way as data dependence and control dependence propagation is represented by the edges in a PDG [22,41].

Venkatesh’s approach does allow for a formal statement of the way in which dynamic and static slicing are related. However, Venkatesh was concerned with the three broad parameters of slicing and not the details of dynamic slicing. As a result, he did not take account of the additional components of the dynamic slicing criterion: the path preservation and the iteration count sensitivity. Rather, Venkatesh’s work was only cornered with Agrawal and Horgan version of dynamic slicing and so avoided a lot of the subtlety found in the present paper.

There are several surveys of slicing: Tip [57], and Binkley and Gallagher [9] provide surveys of program slicing techniques and applications. De Lucia [20] presents a shorter, but more up-to-date survey of slicing paradigms. Binkley and Harman [10] present a survey of empirical results on program slicing. These papers provide a broad picture of slicing technology, tools, applications, definitions, and theory. By contrast, the present paper is solely concerned with the formalization and analysis of dynamic slicing.

9. 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. It had previously been thought that there were only two nodes in the subsumption relationship between static and dynamic slicing. That is, it was thought that the dynamic slicing criterion merely adds the input sequence to the static criterion and this is all that there is to the difference between the two.

However, the results of the study presented here show that the original dynamic slicing criterion introduced by Korel and Laski contains two additional aspects over-and-above the input sequence. These are the iteration count and the requirement to maintain a form of projected path equivalence to the original program.

These two additional criteria are shown to be orthogonal components of the original dynamic slicing definition, thereby yielding an eight-element lattice rather than a two-element lattice. These two new dimensions can be treated as separate criteria in their own right and may find applications which have yet to be fully exploited by the program slicing community.

The paper considered two forms of subsumption relationship. The first is the relationship between semantic properties of a slice, as captured in the equivalence maintained by slicing. The second relationship concerns the relationship between the slices which may be constructed by the equivalence relations. Thus, the first subsumption relationship is purely about the semantic projections denoted by different forms of slicing criteria, while the second concerns the slices which may be produced when this semantic requirement is combined with a syntactic ordering. The paper shows that the two lattices so-constructed are isomorphic.

Future work will consider the relationships among minimal formulations of slice and operations on slicing criteria and will attempt to encompass additional forms of slicing within the theoretical framework established by this paper.

Acknowledgements

The authors wish to thank the anonymous referees. Their detailed, constructive and thoughtful comments greatly improved the presentation of the results in this paper. Mark Harman is supported, in part, by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two development grants from DaimlerChrysler. Dave Binkley is supported by National Science Foundation grant CCR0305330. Tibor Gyimóthy and Ákos Kiss are supported by The Péter Pázmány Program of the Hungarian National Office of Research and Technology (no. RET-07/2005).

References

- [1] H. Agrawal, R.A. DeMillo, E.H. Spafford, Debugging with dynamic slicing and backtracking, *Software Practice Experience* 23 (6) (1993) 589–616.
- [2] H. Agrawal, J.R. Horgan, Dynamic program slicing, in: *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, New York, 1990, pp. 246–256.
- [3] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: *IEEE/ACM 15th Conf. on Software Engineering (ICSE'93)*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1993, pp. 509–518.
- [4] A. Beszédes, T. Gergely, Z.M. Szabó, J. Csirik, T. Gyimóthy, Dynamic slicing method for maintenance of large C programs, in: *Proc. Fifth European Conf. on Software Maintenance and Reengineering (CSMR 2001)*, IEEE Computer Society, Silver Spring, MA, 2001, pp. 105–113.
- [5] J.M. Bieman, L.M. Ott, Measuring functional cohesion, *IEEE Trans. Software Eng.* 20 (8) (1994) 644–657.
- [6] D.W. Binkley, Precise executable interprocedural slices, *ACM Lett. Programming Languages Systems* 3 (1–4) (1993) 31–45.
- [7] D.W. Binkley, The application of program slicing to regression testing, in: M. Harman, K. Gallagher (Eds.), *Information and Software Technology Special Issue on Program Slicing*, Vol. 40, Elsevier, Amsterdam, 1998, pp. 583–594.
- [8] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, L. Ouarbya, Formalizing executable dynamic and forward slicing, in: *Fourth Internat. Workshop on Source Code Analysis and Manipulation (SCAM 04)*, IEEE Computer Society Press, Los Alamitos, CA, USA, Chicago, IL, USA, 2004, pp. 43–52.
- [9] D.W. Binkley, K.B. Gallagher, Program slicing, in: M. Zelkowitz (Ed.), *Advances in Computing*, Vol. 43, Academic Press, New York, 1996, pp. 1–50.
- [10] D.W. Binkley, M. Harman, A survey of empirical results on program slicing, *Adv. Comput.* 62 (2004) 105–178.
- [11] D.W. Binkley, S. Horwitz, T. Reps, Program integration for languages with procedure calls, *ACM Trans. Software Eng. Methodology* 4 (1) (1995) 3–35.
- [12] G. Canfora, A. Cimitile, A. De Lucia, Conditioned program slicing, in: M. Harman, K. Gallagher (Eds.), *Information and Software Technology Special Issue on Program Slicing*, Vol. 40, Elsevier Science B.V., Amsterdam, 1998, pp. 595–607.

- [13] G. Canfora, A. Cimitile, A. De Lucia, G.A.D. Lucca, Software salvaging based on conditions, in: *Internat. Conf. on Software Maintenance (ICSM'96)*, IEEE Computer Society Press, Los Alamitos, CA, USA, Victoria, Canada, 1994, pp. 424–433.
- [14] G. Canfora, A. Cimitile, M. Munro, RE²: reverse engineering and reuse re-engineering, *J. Software Maintenance: Res. Practice* 6 (2) (1994) 53–72.
- [15] R. Cartwright, M. Felleisen, The semantics of program dependence, in: *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1989, pp. 13–27.
- [16] A. Cimitile, A. De Lucia, M. Munro, Identifying reusable functions using specification driven program slicing: a case study, in: *Proc. IEEE Internat. Conf. on Software Maintenance (ICSM'95)*, IEEE Computer Society Press, Los Alamitos, CA, USA, Nice, France, 1995, pp. 124–133.
- [17] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, *Software Maintenance: Res. Practice* 8 (1996) 145–178.
- [18] S. Danicic, Dataflow minimal slicing, Ph.D. Thesis, School of Informatics, University of North London, UK, April 1999.
- [19] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, L. Ouarbya, M. Ward, Consus: a lightweight program conditioner, *J. Systems Software* (2004), accepted for publication.
- [20] A. De Lucia, Program slicing: methods and applications, in: *First IEEE Internat. Workshop on Source Code Analysis and Manipulation*, IEEE Computer Society Press, Los Alamitos, CA, USA, Florence, Italy, 2001, pp. 142–149.
- [21] A. De Lucia, A.R. Fasolino, M. Munro, Understanding function behaviors through program slicing, in: *Fourth IEEE Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos, CA, USA, Berlin, Germany, 1996, pp. 9–18.
- [22] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, *ACM Trans. Programming Languages Systems* 9 (3) (1987) 319–349.
- [23] C. Fox, S. Danicic, M. Harman, R.M. Hierons, ConSIT: a fully automated conditioned program slicer, *Software Practice Experience* 43 (2004) 15–46; published online 26th November 2003.
- [24] K.B. Gallagher, Evaluating the surgeon's assistant: results of a pilot study, in: *Proc. Internat. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, pp. 236–244.
- [25] K.B. Gallagher, J.R. Lyle, Using program slicing in software maintenance, *IEEE Trans. Software Eng.* 17 (8) (1991) 751–761.
- [26] R. Giacobazzi, I. Mastroeni, Non-standard semantics for program slicing, *Higher-Order and Symbolic Comput.* 16 (4) (2003) 297–339 (special issue on Partial Evaluation and Semantics-Based Program Manipulation).
- [27] R. Gopal, Dynamic program slicing based on dependence graphs, in: *IEEE Conf. on Software Maintenance*, 1991, pp. 191–200.
- [28] Grammatech Inc., The codesurfer slicing system, 2002 (URL www.grammatech.com).
- [29] R. Gupta, M.J. Harrold, M.L. Soffa, An approach to regression testing using slicing, in: *Proc. IEEE Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, USA, Orlando, FL, USA, 1992, pp. 299–308.
- [30] M. Harman, D.W. Binkley, S. Danicic, Amorphous program slicing, *J. Systems Software* 68 (1) (2003) 45–64.
- [31] M. Harman, S. Danicic, Using program slicing to simplify testing, *Software Testing, Verification Reliability* 5 (3) (1995) 143–162.
- [32] M. Harman, S. Danicic, Amorphous program slicing, in: *Fifth IEEE Internat. Workshop on Program Comprehension (IWPC'97)*, IEEE Computer Society Press, Los Alamitos, CA, USA, Dearborn, Michigan, USA, 1997, pp. 70–79.
- [33] M. Harman, R.M. Hierons, An overview of program slicing, *Software Focus* 2 (3) (2001) 85–92.
- [34] M. Harman, R.M. Hierons, S. Danicic, J. Howroyd, C. Fox, Pre/post conditioned slicing, in: *IEEE Internat. Conf. on Software Maintenance (ICSM'01)*, IEEE Computer Society Press, Los Alamitos, CA, USA, Florence, Italy, 2001, pp. 138–147.
- [35] M. Harman, L. Hu, R.M. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, *IEEE Trans. Software Eng.* 30 (1) (2004) 3–16.
- [36] M. Harman, D. Simpson, S. Danicic, Slicing programs in the presence of errors, *Formal Aspects Comput.* 8 (4) (1996) 490–497.
- [37] R.M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, *Software Testing, Verification Reliability* 9 (4) (1999) 233–262.
- [38] R.M. Hierons, M. Harman, C. Fox, L. Ouarbya, M. Daoudi, Conditioned slicing supports partition testing, *Software Testing, Verification Reliability* 12 (2002) 23–28.
- [39] S. Horwitz, J. Prins, T. Reps, Integrating non-interfering versions of programs, *ACM Trans. Programming Languages Systems* 11 (3) (1989) 345–387.
- [40] S. Horwitz, T. Reps, D.W. Binkley, Interprocedural slicing using dependence graphs, in: *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Atlanta, Georgia, 1988, pp. 25–46; *Proc. in SIGPLAN Notices*, Vol. 23(7), 1988, pp. 35–46.
- [41] S. Horwitz, T. Reps, D.W. Binkley, Interprocedural slicing using dependence graphs, *ACM Trans. Programming Languages Systems* 12 (1) (1990) 26–61.
- [42] M. Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, Ph.D. Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, available as Linköping Studies in Science and Technology, Dissertations, Number 297, 1993.
- [43] M. Kamkar, N. Shahmehri, P. Fritzson, Interprocedural dynamic slicing, in: *Proc. Fourth Conf. on Programming Language Implementation and Logic Programming*, 1992, pp. 370–384.
- [44] D.E. Knuth, R.W. Floyd, Notes on avoiding “go to” statements, *Inform. Process. Lett.* 1 (1) (1971) 23–31.
- [45] B. Korel, J. Laski, Dynamic program slicing, *Inform. Process. Lett.* 29 (3) (1988) 155–163.
- [46] B. Korel, J. Rilling, Dynamic program slicing in understanding of program execution, in: *Fifth IEEE Internat. Workshop on Program Comprehension (IWPC'97)*, IEEE Computer Society Press, Los Alamitos, CA, USA, Dearborn, MI, USA, 1997, pp. 80–89.
- [47] B. Korel, J. Rilling, Dynamic program slicing methods, in: M. Harman, K. Gallagher (Eds.), *Information and Software Technology Special Issue on Program Slicing*, Vol. 40, Elsevier, Amsterdam, 1998, pp. 647–659.
- [48] A. Lakhota, Rule-based approach to computing module cohesion, in: *Proc. 15th Conf. on Software Engineering (ICSE-15)*, 1993, pp. 34–44.

- [49] M.R. Laurence, S. Danicic, M. Harman, R. Hierons, J. Howroyd, Equivalence of conservative, free, linear program schemas is decidable, *Theoret. Comput. Sci.* 290 (2003) 831–862.
- [50] J.R. Lyle, M. Weiser, Automatic program bug location by program slicing, in: *Second Internat. Conf. on Computers and Applications*, IEEE Computer Society Press, Los Alamitos, CA, USA, Peking, 1987, pp. 877–882.
- [51] L.M. Ott, J.J. Thuss, Slice based metrics for estimating cohesion, in: *Proc. IEEE-CS Internat. Metrics Symp.*, IEEE Computer Society Press, Los Alamitos, CA, USA, Baltimore, MD, USA, 1993, pp. 71–81.
- [52] K.J. Ottenstein, L.M. Ottenstein, The program dependence graph in software development environments, *SIGPLAN Notices* 19 (5) (1984) 177–184.
- [53] L. Ramshaw, Eliminating goto's while preserving program structure, *J. ACM* 35 (4) (1988) 893–920.
- [54] T. Reps, Program analysis via graph reachability, in: M. Harman, K. Gallagher (Eds.), *Information and Software Technology Special Issue on Program Slicing*, Vol. 40, Elsevier Science B.V., Amsterdam, 1998, pp. 701–726.
- [55] T. Reps, W. Yang, The semantics of program slicing, Technical Report 777, University of Wisconsin, 1988.
- [56] D. Simpson, S.H. Valentine, R. Mitchell, L. Liu, R. Ellis, Recoup—maintaining Fortran, *ACM Fortran Forum* 12 (3) (1993) 26–32.
- [57] F. Tip, A survey of program slicing techniques, *J. Programming Languages* 3 (3) (1995) 121–189.
- [58] G.A. Venkatesh, The semantic approach to program slicing, in: *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Toronto, Canada, 1991, pp. 26–28; *Proc. SIGPLAN Notices*, Vol. 26(6), 1991, pp. 107–119.
- [59] M. Weiser, Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, Ph.D. Thesis, University of Michigan, Ann Arbor, MI, 1979.
- [60] M. Weiser, Programmers use slicing when debugging, *Comm. ACM* 25 (7) (1982) 446–452.
- [61] M. Weiser, Program slicing, *IEEE Trans. Software Eng.* 10 (4) (1984) 352–357.
- [62] M. Weiser, J. R. Lyle, Experiments on slicing-based debugging aids, in: E. Soloway, S. Iyengar (Eds.), *Empirical Studies of Programmers*, Molex, 1985, pp. 187–197 (Chap. 12).