

Static Program Slicing Algorithms are Minimal for Free Liberal Program Schemas

SEBASTIAN DANICIC¹, CHRIS FOX², MARK HARMAN³, ROB HIERONS⁴,
JOHN HOWROYD⁵ AND MICHAEL R. LAURENCE⁶

¹*Department of Computing, Goldsmiths College, University of London, New Cross,
London SE14 6NW, UK*

²*Department of Computer Science, University of Essex, Colchester CO4 3SQ, UK*

³*Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK*

⁴*School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge,
Middlesex UB8 3PH, UK*

⁵*@UK PLC, 5 Jupiter House, Calleva Park, Aldermaston, Berkshire RG7 8NN, UK*

⁶*Department of Computer Science, University of Liverpool, Peach Street, Liverpool L69 3BX, UK
Email: S.Danicic@gold.ac.uk*

Program slicing is an automated source code extraction technique that has been applied to a number of problems including testing, debugging, maintenance, reverse engineering, program comprehension, reuse and program integration. In all these applications the size of the slice is crucial; the smaller the better. It is known that statement minimal slices are not computable, but the question of dataflow minimal slicing has remained open since Weiser posed it in 1979. This paper proves that static slicing algorithms produce dataflow minimal end slices for programs which can be represented as schemas which are free and liberal.

Received 21 January 2005; revised 31 May 2005

1. INTRODUCTION

In program slicing [1], statements are deleted from a program, leaving a resulting program called a *slice*. The slice must preserve the effect of the original program on a set of variables of interest at a particular *program point*. The set of variables and the program point are known as the 'slicing criterion'. Slicing has applications in many areas of computing including reverse engineering [2, 3], program comprehension [4, 5], software maintenance [6, 7, 8, 9], debugging [10, 11, 12, 13], testing [14, 15, 16, 17, 18], component reuse [19, 20], program integration [21, 22] and software metrics [23, 24, 25]. There are several surveys of slicing techniques, applications and variations [26, 27, 28, 29, 30].

In all applications of slicing, the size of the slice is crucial. More the code removed by the slicing algorithm, the better. It is known, however, that statement minimal slices are not, in general, computable [1]. However, since the time Weiser posed the question in 1979, the question of dataflow minimality has remained open [1]. This paper reformulates the dataflow minimal slicing question in terms of program schematology and proves that slicing algorithms do produce minimal slices for free liberal program schemas.

Program schemas [31] are 'programs' where all expressions in the program have been replaced by *symbolic expressions* of the form $n(v_1, \dots, v_m)$ where n is a function or predicate name and v_1, \dots, v_m are variables (Figure 1). A program can be transformed into a program schema, simply by replacing all expressions by symbolic expressions, each with an uninterpreted function or predicate name. A schema where the function and predicate names are all unique is called a *linear* schema [32]. The variables mentioned in the expression correspond to the referenced variables in the node of the annotated CFG. Figure 1 shows a program, its corresponding annotated CFG and its corresponding program schema (program schemas are defined more formally in the sequel). An assignment in the program is represented by an assignment to a symbolic expression in the schema.

Weiser's dataflow-based approach [1, 33] and the program dependence graph approach [34], produce the same slices for the same slicing criterion, so the results of this paper apply equally to both. Furthermore, both approaches operate at a level of abstraction where, in a program, the only information that can be utilized about each expression, e , is the set of variables referenced by e . Weiser termed this approach

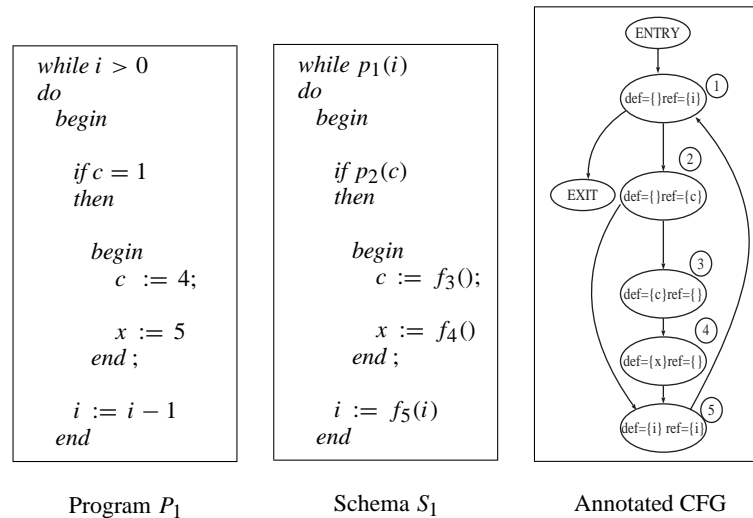


FIGURE 1. A program, its schema and its annotated CFG.

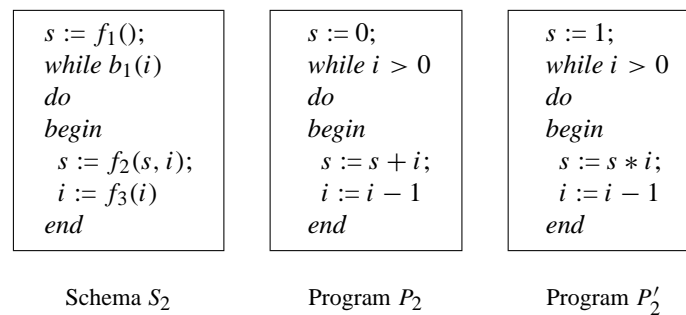


FIGURE 2. A schema and two programs in its equivalence class.

dataflow analysis, but we call it *DefRef abstraction*¹, as the term *dataflow analysis* now has more general connotations.

Figure 2 shows two distinct programs which are identical to each other under DefRef abstraction. Algorithms that use DefRef abstraction are limited in the sense that they cannot take advantage of situations where expressions in the program are equal, nor can any form of expression simplification be used. All the information required to do such things has been ‘abstracted away’. For example, after DefRef abstraction of $\{y := x + 1; z := x + 1\}$ the only remaining information is that the variable y is assigned an expression which references x and the variable z is assigned an expression which references x , and the assignments happen in that order.

Analysing a program, P , after DefRef abstraction, is identical to first converting P into a corresponding linear schema, S , and then analysing S . In this paper, this connection between program schemas and the level of abstraction for slicing is used to define dataflow minimal slicing in terms of program schemas and to prove the primary results of the paper: slicing algorithms are minimal for free, liberal schemas. In this paper, only intraprocedural slicing

¹Other approaches to program slicing exist [6, 7, 35, 36, 37] which do not use DefRef abstraction, but we are concerned with the theoretical properties of traditional static slicing which do.

is considered. Throughout the paper references to ‘Weiser’s algorithm’ may be interpreted as references also to the system dependence graph slicing algorithm of Horwitz *et al.* [34], since this algorithm produces the same intraprocedural slices as Weiser’s algorithm.

In this paper, we shall be concerned only with end slicing [24], so the point of interest will always be the end of the program and a slicing criterion will, therefore, simply be a set of variables.

As an example of the dataflow minimal slicing problem, consider slicing P_1 , in Figure 1, at the end of the program, with respect to variable x . Weiser’s algorithm fails to delete any statement at all. However, the assignment $c := f_3()$ can be deleted to produce a valid slice. To see this, observe that the assignment $c := f_3()$ is executed if and only if the constant assignment $x := f_4()$ is executed. Having been assigned a constant value, the value of x cannot be further changed by the body of the loop. The initial value of c is important, but the later assignment to c cannot affect the final value of x . The assignment $c := f_3()$, therefore, need not be included in the slice. The reason that Weiser’s algorithm includes $c := f_3()$ is that the assignment $x := f_4()$ is controlled by the predicate $p_2(c)$, which, in turn, is data dependent on $c := f_3()$. Therefore, since Weiser’s algorithm computes

the transitive closure, it infers that $x := f_4()$ depends on $c := f_3()$.

Importantly, the reason that the assignment, $c := f_3()$, can be left out of a slice can be justified purely by analysing the CFG in Figure 1 and not the program, i.e. the reasoning that allows it to be removed can be conducted at the DefRef level of abstraction. The example shows that Weiser's algorithm is not dataflow minimal. However, as this paper shows, Weiser's algorithm is dataflow minimal for an important class of program schemas.

The rest of this paper is organized as follows: In Section 2, program schemas and their properties are formally defined. In Section 3, a schema-based theory of slicing is introduced and the main result of the paper is proved in terms of this theory. Section 4 discusses related work and finally, Section 5 presents conclusions and future work.

2. PROGRAM SCHEMAS

2.1. Basic terminology

In this section, for completeness, the basic definitions and terminology used in the theory of program schemas are introduced. Similar definitions can be found in Manna [38]. For a complete and excellent survey of the theory of program schemas see Greibach [39].

Schemas. A schema S is possibly an empty sequence of statements. Each statement in S is called a *substatement* of S .

Statements. A statement, S , satisfies

$$\begin{aligned} S &\rightarrow skip \\ S &\rightarrow V := E \\ S &\rightarrow \text{if } E \text{ then } S \text{ else } S \\ S &\rightarrow \text{while } E \text{ do } S, \end{aligned}$$

where V is a variable name and E is an *expression* of the form $n(v_1, \dots, v_k)$ where n is a (function or predicate) name and v_1, \dots, v_k is possibly an empty list of variable names. The name n is called the *head* of the expression $n(v_1, \dots, v_k)$.

The schemas S_1 and S_2 are called *subschemas* of the statement *if e then S_1 else S_2* . The expression e is called the *guard* of *if e then S_1 else S_2* . Similarly, the schema S is called a subschema of the statement *while e do S* and the expression e is called the *guard* of *while e do S* .

Structured statements. *if e then S_1 else S_2* and *while* loops are called *structured statements*.

Predicate expressions. Given a schema S , a *predicate expression* in S is one which occurs as the guard of a structured statement.

Function names. The head of an expression that occurs on the right-hand side of an assignment statement is called a *function name*.

Predicate names. The head of a predicate expression is called a *predicate name*. In a Schema, the sets of function names and predicate names are disjoint.

Names. For conciseness, it is useful to define a *name* to be either a function name or a predicate name.

The alphabet of a schema. For each schema, S , the alphabet, $\alpha(S)$, of S is defined by

$$\alpha(S) = A \cup B \cup C,$$

where $A = \{v := e \mid v := e \text{ is an assignment in } S\}$; $B = \{e = \text{True} \mid e \text{ is a predicate expression in } S\}$; and $C = \{e = \text{False} \mid e \text{ is a predicate expression in } S\}$.

Elements of A are called *assignment symbols*, and elements of B and C are called *predicate symbols*.

Sequences. We write λ to represent the empty sequence.

If A is a set of symbols, we write A^* for the set of finite words over A .

If L is a set of words, we write L^* for the set of finite concatenations of words in L .

If $\Sigma_1, \dots, \Sigma_n$ are sets of words, we write $\Sigma_1 \dots \Sigma_n$ to represent the set of words

$$\{\sigma_1 \dots \sigma_n \mid \sigma_i \in \Sigma_i\}.$$

Also, if Σ is a set of words and \underline{a} is a symbol, we will write $\Sigma \underline{a}$ to represent the set of words

$$\{\sigma \underline{a} \mid \sigma \in \Sigma\}.$$

Similarly, we will write $\underline{a} \Sigma$ to represent the set of words

$$\{\underline{a} \sigma \mid \sigma \in \Sigma\}.$$

Following the usual conventions, we use \underline{a} to represent both the symbol, \underline{a} and the word of length, one which contains the symbol \underline{a} . The meaning is always obvious from the context.

The set of finite words of a schema. The set of all finite words of schema S is defined inductively as follows:

If S is empty,

$$\Sigma(\lambda) = \{\lambda\}$$

If S is a non-empty sequence, $S_1 S_2 \dots S_r$ of statements,

$$\Sigma(S_1 S_2 \dots S_r) = \Sigma(S_1) \dots \Sigma(S_r).$$

The set of finite words of a statement. The set, $\Sigma(S) \subseteq (\alpha(S))^*$, of all finite words of a statement is defined inductively as follows:

$$\Sigma(skip) = \{\lambda\}.$$

$$\Sigma(v := e) = \{v := e\}.$$

$$\Sigma(\text{if } e \text{ then } T_1 \text{ else } T_2) = \underline{e = \text{True}} \Sigma(T_1)$$

$$\cup \underline{e = \text{False}} \Sigma(T_2).$$

$$\Sigma(\text{while } e \text{ do } T) = \underline{e = \text{True}} \Sigma(T)^* \underline{e = \text{False}}.$$

In other words, a path is generated by recording the value of the guard followed by a path of the corresponding branch.

Prefixes. Given a schema S , any prefix of an element of $\Sigma(S)$ is called a prefix of S .

Infinite words. Formally, an infinite word is a mapping from the natural numbers to a set of symbols. A prefix of an infinite word is its restriction to an initial segment $0, \dots, n$.

Given a schema S , π is an infinite word of S if and only if π is an infinite word over the alphabet of S such that all prefixes of π are prefixes of S .

Terms. A term is either a variable or of the form $n(t_1, \dots, t_k)$, where n is a name and t_1, \dots, t_k is a possible empty list of terms.

State. A state Δ is either \perp or a (total) function from terms to terms such that

$$\Delta n(t_1, \dots, t_k) = n(\Delta t_1, \dots, \Delta t_k)$$

for all terms $n(t_1, \dots, t_k)$.

We use v to stand both for the variable v and the term v . It is always clear from the context whether an expression is a term or a variable. Similarly, it can be seen that expressions are also terms. Note that a state is fully defined by stating how it maps variables.

The identity state. The identity function on terms is written \mathcal{I} .

Predicate terms. Given a predicate expression e and a state Δ , the result of evaluating e in Δ , Δe , is called a *predicate term*.

2.2. Semantics

The semantics of prefixes. The meaning of a prefix is a state. It is the sequential composition of the meanings of its elements. In other words,

$$[\lambda] = \mathcal{I}$$

$$[a_1 \cdots a_k] = [a_1] \circ \cdots \circ [a_k].$$

The semantics of an assignment symbol. The meaning of an assignment symbol $\underline{v} := e$ is the state $[\underline{v} := e]$ defined by

$$[\underline{v}_1 := e]v_2 = \begin{cases} v_2 & \text{if } v_1 \neq v_2, \\ e & \text{if } v_1 = v_2. \end{cases}$$

The semantics of predicate symbols. The meanings of $e = \text{True}$ and $e = \text{False}$ are both the identity state \mathcal{I} .

Herbrand interpretations. A Herbrand interpretation is a function from predicate terms to $\{\text{True}, \text{False}\}$.

In order to give the semantics of a general schema S , first the path, $\mathcal{P}[[S]]i$, of S with respect to Herbrand interpretation, i , is defined.

The semantics of schemas. Given a Herbrand interpretation i , $\mathcal{P}[[S]]i$ is defined to be the unique word π of S satisfying

the property that for every prefix, $\pi'p = X$, of π , (where $X \in \{\text{True}, \text{False}\}$) we have

$$i([\pi']p) = X.$$

The meaning of a schema is a mapping from Herbrand interpretations to states, defined as follows:

$$\mathcal{M}[[S]]i = \begin{cases} [\mathcal{P}[[S]]i] & \text{if } \mathcal{P}[[S]]i \text{ is finite} \\ \perp & \text{otherwise.} \end{cases}$$

2.3. Further definitions

Simple Herbrand interpretations. A simple Herbrand interpretation is a Herbrand interpretation that does not map infinitely many terms to True.

Terminating Herbrand interpretations. A Herbrand interpretation i is said to be *terminating* for S if and only if $\mathcal{P}[[S]]i$ is finite. The interpretation i is said to be *non-terminating* for S if and only if $\mathcal{P}[[S]]i$ is infinite.

Paths of S . For every Herbrand interpretation, i , $\mathcal{P}[[S]]i$ is called a *path* of S .

Legal Prefixes. A (finite) prefix of a path of S is called a *legal prefix* of S .

Liberal prefixes. A word (finite or infinite) σ is said to be *liberal* if and only if for all distinct prefixes $\sigma_1 \underline{v}_1 := e_1$ and $\sigma_2 \underline{v}_2 := e_2$ of σ we have

$$[\sigma_1 \underline{v}_1 := e_1]v_1 \neq [\sigma_2 \underline{v}_2 := e_2]v_2.$$

2.4. Classes of schema

Free schemas. A schema S is said to be *free* if every word of S is a path of S .

Informally, a schema is free if all words are possible. An example of a free schema S_2 is illustrated in Figure 2. For all n there is an interpretation which will take S_2 exactly n times round the loop. There is also an interpretation which will take S_2 infinitely many times round the loop. Schema S_1 in Figure 1, however, is not free. Since the variable c is assigned a constant value, $f_3()$, there is no Herbrand interpretation, for example, that will execute the loop exactly four times alternating between the true and false branches of the statement in the loop.

Linear schemas. A *linear schema* is one where each name in the schema occurs at most once. All the schemas mentioned in this paper are linear (apart from the following one!). A simple example of a non-linear schema is $x := f(x); x := f(x)$. This is non-linear because the name f occurs more than once.

Liberal schemas. A schema is *liberal* [40] if and only if all its legal prefixes are liberal.

Informally, a schema is liberal if no variable gets assigned the same term more than once. Schema S_1 , in Figure 1, is non-liberal since the variable c may be assigned the same constant value more than once. An example of a liberal is schema S_2 in Figure 2 since a repetition of terms is not possible as the terms assigned to variables s and i get bigger each time round the loop.

2.5. Basic results

LEMMA 2.1. *For every terminating Herbrand interpretation, i of S , there is a simple Herbrand interpretation, j , such that $\mathcal{P}[\![S]\!]i = \mathcal{P}[\![S]\!]j$.*

Proof. Let j map every term that does not occur in $\mathcal{P}[\![S]\!]i$ to False. □

LEMMA 2.2. *If S is free and $\sigma e_1 = X$ and $\tau e_2 = Y$ are distinct prefixes of the same prefix of S . Then*

$$[\sigma] e_1 \neq [\tau] e_2.$$

Proof. Let S be a free schema and let $\sigma e_1 = X$ and $\tau e_2 = Y$ be distinct prefixes of the same prefix of S . Without loss of generality, let $\sigma e_1 = X$ be a prefix of $\tau e_2 = Y$. Suppose

$$[\sigma] e_1 = [\tau] e_2.$$

This means that the value of the expression e_1 after executing prefix σ is the same as the value of the expression e_2 after executing prefix τ . Clearly, since S is free, $\tau e_2 = \neg Y$ is a word of S but there is no Herbrand interpretation that gives rise to this word, because the same term cannot be mapped to different values by a Herbrand Interpretation. This provides a contradiction as required. □

LEMMA 2.3. *Let S be a free schema. If Herbrand interpretation, i , is simple then i is terminating.*

Proof. Follows immediately from Lemma 2.2. □

3. SLICING AND SCHEMAS

We now show how the syntax and semantics of slicing can be defined using schemas. Having defined dataflow minimality and Weiser’s algorithm in terms of schemas, the theory is further developed, leading to the main result of the paper that Weiser’s algorithm (and consequently, other traditional approaches) produces dataflow minimal slices for schemas which are liberal and free.

Weiser defined the semantic relationship that must exist between a program and its slice in terms of state trajectories: a state trajectory is a sequence of label, state pairs (l_i, σ_i) where σ_i represents the state immediately before executing the statement labelled l_i . It should be noted that here, a *state* is the *program state*, namely a function from variable names to values; not the states which map variable names to terms used in the semantics of schemas introduced in Section 2.

DEFINITION 3.1. (Weiser Slices). *A slice s of a program p on a slicing criterion $c = (V, i)$ is any executable program with the following property. Whenever p halts on an input I*

with a state trajectory T , then s also halts on input I with state trajectory T' with

$$\text{Proj}_c(T) = \text{Proj}_c(T').$$

$\text{Proj}_c(T)$ is obtained first by deleting all elements of T whose label component is not i and then, by restricting the state components to V^2 .

When slicing at the end of the program, the trajectories will all be of length one (since the ‘exit’ statement is executed only once). This gives rise to a simplified form of slicing called *end slicing*.

DEFINITION 3.2. (Weiser’s semantic definition of an end slice). *Program p' is a v -semantic-end-slice³ of p with respect to a variable v such that if and whenever p terminates, so does p' with the same final value of v .*

We now restate Weiser’s definitions in terms of linear schemas and further develop our theory of end-slicing schemas. Conventionally, a slice must be a *syntactic subset* of the program being sliced. We express this in terms of schemas.

DEFINITION 3.3. (Syntactic subsets of a schema). *Let S and T be schemas. Then T is said to be syntactic subset of S whenever T can be produced by replacing any substatements t of S by a syntactic subset of t .*

DEFINITION 3.4. (Syntactic subsets of a statement). *Let s and t be statements. Then t is said to be syntactic subset whenever $s = t$, t is skip or t can be obtained from s by replacing any subschema, T of s by a syntactic subset of T .*

Semantically, a v -semantic-end-slice with respect to v must behave the same with respect to variable v . We define this in terms of schemas as follows:

DEFINITION 3.5. (v -semantic-end-slices). *Let S be a schema, and let v be a variable. A v -semantic-end-slice of S is a schema T such that for all terminating Herbrand interpretations i for S we have $(\mathcal{M}[\![T]\!]i)v = (\mathcal{M}[\![S]\!]i)v$.*

The Luckham–Park–Paterson theorem [31] (also see [38, Theorem 4.1]) ensures that if S' is a v -semantic-end-slice of S then for all interpretations, i , the corresponding program p'_i of S' will be a v -semantic-end-slice of the program p_i corresponding to S .

LEMMA 3.1. (Syntactic minimal subsets). *Given a linear schema S and a set N of function and predicate names of S , there is a unique minimal syntactic subset of S that contains all the symbols in N .*

Proof. First, replace all assignments whose function name is not in N by *skip*. Then, if a structured statement contains no elements of N then replace it by *skip*. Clearly, the resulting schema is a syntactic subset of S and is minimal in the sense that if we cannot further remove any more predicate names,

²This is a slight simplification of the true picture since we are assuming that i is in the slice of p with respect to c . A more complicated definition involving ‘nearest successors’ is required if i is not in the slice.

³This is our term, not Weiser’s.

since this, by definition will result in a schema which is not a syntactic subset of the original. \square

This tells us how to reconstruct a slice from a set of function and predicate names.

DEFINITION 3.6. (Dataflow minimal v -end-slice). *Let S be a schema and let v be a variable. Schema T is a dataflow minimal v -end-slice of S if and only if*

- (i) T is a syntactic subset of S ,
- (ii) T is a v -semantic-end-slice of S and
- (iii) every proper syntactic subset of T is not a v -semantic-end-slice of S .

DEFINITION 3.7. (Dataflow minimal program slices). *A program q is a dataflow minimal v -end-slice of p if and only if there exist linear schemas S and T where S and T are representations (under identical interpretations) of p and q , respectively with T a dataflow minimal v -end-slice of S .*

Weiser's algorithm (and most subsequent work on program dependence) uses two relations [41] between the nodes of a program's control flow graph. These are *data dependence* (D) and *control dependence* (C). In order to compute these dependences, all that is required is the set of variables mentioned at each node of the program's control flow graph. This coincides with our notion of DefRef abstraction.

Data dependence is the transitive closure of *direct data dependence*, where node n_2 is *directly data dependent* on node n_1 if there is a variable v referenced in n_2 which is defined in n_1 and there is a path in the control flow graph from n_1 to n_2 with no intervening assignments to v . We write $n_1 D n_2$ to mean n_2 is data dependent on n_1 . Consider:

Node n_1	$x := y;$
.	...
Node n_2	$z := x$

If there are no intervening assignments to x between n_1 and n_2 , then node n_2 is data dependent on node n_1 since the value of x at n_2 is 'affected by' the value of y at n_1 . Similarly, consider:

	<i>while b do</i>
	<i>begin</i>
A	...
Node n_2	$z := x;$
.	.
.	.
.	.
Node n_1	$x := y;$
B	...
	<i>end</i>

Again, if there are no assignments to x in the portions of code labelled A and B, then node n_2 is data dependent on node n_1 since the value of x at n_2 is 'affected by' the value of y at n_1 . This is an example of a *loop carried* data dependence [42].

Given a linear schema S , we can define data dependence in terms of the function and predicate names occurring in the schema. f is data dependent on g if and only if there is a finite prefix $\tau v := f(\mathbf{y})$ of a word of S with g occurring as the head name of an outermost subterm of $[\tau]f(\mathbf{y})$.

Informally, the execution of a predicate node 'controls' the execution of other nodes in the control flow graph by determining whether or not control will definitely pass to these nodes or not. For each predicate node, b , the set of nodes that depend on the outcome of b in this way are termed *the controlled nodes* of b . For the structured programs considered in this paper, the statements controlled by a predicate are simply the 'top level' statements in its body.

Weiser's algorithm is now expressed in terms of linear schemas. Since other slicing algorithms [34, 43] produce the same slices, this definition also captures these 'traditional' approaches to slicing.

DEFINITION 3.8. (Weiser's algorithm expressed in terms of linear schemas). *Given a linear schema S , Weiser's algorithm produces a set of predicate and function names. The slice at v , produced by Weiser's algorithm is the smallest set W_v satisfying the following definition:*

Case 1. A function name f , is in W_v if f is a function name in the term $[\sigma]v$ for some finite word, σ , of S .

Case 2. If p is a predicate name of S such that there is a function name f in the body of p which is in W_v then p is in W_v .

Case 3. A function name f is in W_v if there exists a predicate, p in W_v such that for some prefix, σ , of S ending in $p(w_1 \cdots w_k) = X$, f occurs in the term $[\sigma]p(w_1 \cdots w_k)$.

We observe that a *path* of the control flow graph corresponds to our notion of a *word* of the corresponding linear schema. There is an assumption, therefore, in conventional slicing that the control flow graphs are 'free'. In general, this assumption leads to unnecessarily large slices, because dependencies resulting from infeasible paths will be inferred.

We now extend the theory of schemas in order to express the dependences used in slicing.

DEFINITION 3.9. (Differing only at n). *Let n be a name and i and j Herbrand interpretations. We say i and j differ only at n if and only if a term t does not contain the name n , and we have $i(t) = j(t)$.*

We now define what it means for a variable to *need* a function or predicate name in a schema. We then show that the set of function and predicate names needed by the variable v are in every v -semantic-end-slice.

DEFINITION 3.10. (v needs n in S). *Let v be a variable and let n be a name that occurs in S . We say v needs n if either n is a function name in the term $(\mathcal{M}[\![S]\!]i)v$ for some terminating Herbrand interpretation i , or there exist two terminating Herbrand interpretations i and j differing only at n such that $(\mathcal{M}[\![S]\!]i)v \neq (\mathcal{M}[\![S]\!]j)v$.*

LEMMA 3.2. *Let S be a linear schema, let v be a variable and let n be a function or predicate name. If v needs n then n is in every v -semantic-end-slice of S .*

Proof. Let T be a v -semantic-end-slice of S . If i is a terminating Herbrand interpretation and n is a function name in the term $(\mathcal{M}[\![S]\!]i)v$, n is a function name in the term $(\mathcal{M}[\![T]\!]i)v$ since $(\mathcal{M}[\![S]\!]i)v = (\mathcal{M}[\![T]\!]i)v$. It follows that n appears in T . Alternatively, there exist two terminating Herbrand interpretations i and j differing only on terms containing n such that $(\mathcal{M}[\![S]\!]i)v \neq (\mathcal{M}[\![S]\!]j)v$. Thus $(\mathcal{M}[\![T]\!]i)v \neq (\mathcal{M}[\![T]\!]j)v$ and hence n appears in T in this case as well. \square

In this section we show that for linear free schemas, names included by virtue of cases 1 and 2 of Weiser’s algorithm are needed. These cases are fairly straightforward and do not require the schema to be liberal.

DEFINITION 3.11. (Consequence of a prefix). *Let t be a predicate term and $X \in \{True, False\}$. We say $t = X$ is a consequence of the prefix $\sigma e = X$ if and only if $[\sigma]e = t$.*

DEFINITION 3.12. (Consequence of a path). *Let t be a predicate term and $X \in \{True, False\}$. We say $t = X$ is a consequence of the path π if there exists some prefix $\sigma e = X$ of π such that $t = X$ is a consequence of $\sigma e = X$.*

DEFINITION 3.13. (Differing at t). *Let $X, Y \in \{True, False\}$. Let π_1 and π_2 be paths and let t be a term. Then π_1 and π_2 differ at t means $t = X$ is a consequence of π_1 and $t = Y$ is a consequence of π_2 and $X \neq Y$.*

LEMMA 3.3. *Let S be a linear free schema and let n be a name occurring inside a structured statement whose guard has head p . For every finite path π passing through n there exists another finite path π' which differs from π only at p such that π' does not pass through n .*

Proof.

Case 1. p guards a *while* loop.

Let i be a terminating Herbrand interpretation which passes through n and let j be the same as i except all terms containing p are mapped to False. By Lemma 2.1, we can assume i and j are simple. By Lemma 2.3, j is terminating and it avoids n (by linearity).

Case 2. p guards *if e then S_1 else S_2* .

There are two cases to consider:

Case a. n is in S_1 . Again, let i be a terminating Herbrand interpretation which passes through n and let j be the same as i except all terms containing p are mapped to False. By Lemma 2.1, we can assume i and j are simple. By Lemma 2.3, j is terminating and it avoids n by linearity.

Case b. n is in S_2 . The proof is essentially the same as case (a). \square

Lemma 3.2 shows that if something is needed then it is contained in the end slice. Weiser showed [1] that his algorithm always produces valid slices (although they are not always dataflow minimal). All that remains, therefore, is to show that for the schemas considered here, if Weiser’s algorithm includes it, it is needed. Using our reformulation of Weiser’s algorithm, Definition 3.8, there are three cases to consider. Clearly, for all assignments $w := f(v_1 \dots v_k)$ that are in W_v by virtue of case (1), we have v needs f . It is now shown that the predicate names in W_v , by virtue of case (2), are also needed by v .

PROPOSITION 3.1. *Let S be a free linear schema, let v be a variable in S and let p be a predicate name in S . Suppose that f is a function name in the body of p , such that v needs f . Then v needs p .*

Proof. The variable v needs f . By Definition 3.10, this gives the two cases (1) and (2) to consider. First suppose v needs f because of case (1) i.e. f is a function name in the term $[\pi]v$ for some finite path π of S . By Lemma 3.3, there is a finite path π' that differs from π only at p such that π' does not pass through f and, therefore, f cannot be in $[\pi']v$. So v needs p .

Alternatively, now suppose that v needs f because there exist two Herbrand interpretations i and j differing only at f which give rise to finite paths with different final values for v . We may assume that i and j map to False every term that is not a consequence of the corresponding path of S . First assume that p is a *while* predicate. Let i' be the Herbrand interpretation that is the same as i except that all terms containing p are mapped to False and let j' be the Herbrand interpretation that is the same as j except that all terms containing p are mapped to False. The paths of S corresponding to i' and j' must be identical because i' and j' differ only on terms containing f and their paths do not pass through f . These paths must also be terminating and hence give the same values of v . From this it follows that, by transitivity of equality, either the final values of v with respect to i and i' are different or the final values of v with respect to j and j' are different. But these pairs of Herbrand interpretations differ only at p as required.

The case where p is an *if* predicate follows by an identical argument, so is omitted. \square

This completes the proof of case (2).

For case (3), we require that if p is needed and n percolates to p then n is also needed. This is not necessarily true for non-liberal schemas as can be seen by the example in Figure 3.

In order to prove the third case we first use a result which does not require liberality is Lemma 3.4 called the ‘difference lemma’. If S is free and v needs p , there exist two terminating Herbrand interpretations which have different final values of v differing on exactly one term, and this term contains p . The only result which requires liberality is Lemma 3.6 called the ‘prefixing lemma’: Let $\rho\sigma$ and $\rho\tau$ be liberal prefixes and v a variable. If $[\sigma]v \neq [\tau]v$ then $[\rho\sigma]v \neq [\rho\tau]v$. The importance of this result is that if two terms are distinct after executing two sequences σ and τ they will be different even if we prefix the same sequence of extra ‘instructions’ to the beginning of σ and τ , provided that these sequences are

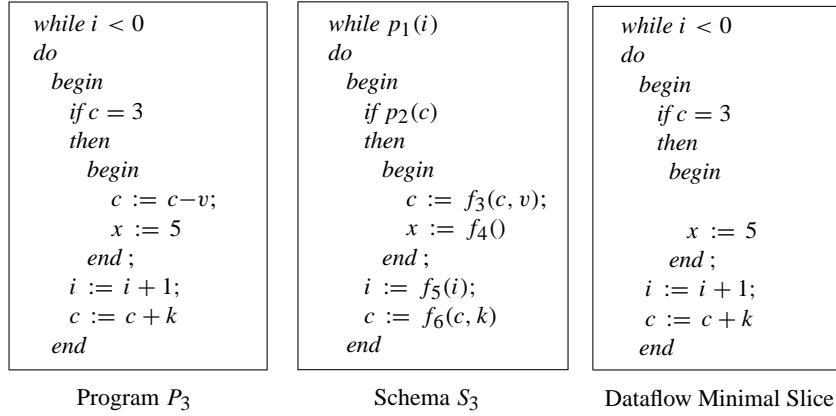


FIGURE 3. A non-liberal example.

liberal. The proof for the third case can be summarized as follows: as p is needed, by the difference lemma there are two interpretations differing only at one place with different final values for v . If we remove from both corresponding paths the initial segment up to this place where they differ, the corresponding final values of v will still be different. This follows from the fact that final values are produced by composing the state function corresponding to each symbol. Let σ be a prefix in which f percolates to p . By freeness, we can ‘prefix’ σ at the beginning of our shortened paths. By the prefixing lemma these new paths will still differ only at this one term containing p and have different final values of v . But this term also contains f and so by definition v needs f .

DEFINITION 3.14. ($d_p(i, j)$). Let p be a predicate name and i and j be two simple Herbrand interpretations. We define $d_p(i, j)$ to be the number of terms containing p on which i and j disagree.

LEMMA 3.4. (The difference lemma). Let v be a variable and let p be a predicate name. Suppose that S is free and v needs p . Then there exist terminating Herbrand interpretations i and j differing only at p such that $(\mathcal{M}[\llbracket S \rrbracket]i)v \neq (\mathcal{M}[\llbracket S \rrbracket]j)v$ and $d_p(i, j) = 1$.

Proof. Since p is a predicate name, p is not in the term $(\mathcal{M}[\llbracket S \rrbracket]i)v$ for any Herbrand interpretation i . Thus there exist terminating Herbrand interpretations i and j differing only on terms containing p such that $(\mathcal{M}[\llbracket S \rrbracket]i)v \neq (\mathcal{M}[\llbracket S \rrbracket]j)v$. Without loss of generality, it may be assumed that for all predicate names q , $i(q(\mathbf{t})) = \text{False} = j(q(\mathbf{t}))$, whenever $q(\mathbf{t}) = \text{True}$ is not a consequence of either of the paths, $\mathcal{P}[\llbracket S \rrbracket]i$ or $\mathcal{P}[\llbracket S \rrbracket]j$. Thus, $d_p(i, j)$ is finite and hence it may be assumed that i and j have been chosen such that $d_p(i, j)$ is minimal. From Lemma 2.1, we may assume that i and j are simple.

Suppose that $d_p(i, j) > 1$.

Let $p(\mathbf{u})$ be a term with $i(p(\mathbf{u})) \neq j(p(\mathbf{u}))$. Notice, by the minimality of $d_p(i, j)$, that $p(\mathbf{u}) = i(p(\mathbf{u}))$ must be a consequence of a prefix of the path, $\mathcal{P}[\llbracket S \rrbracket]i$, and similarly, $p(\mathbf{u}) = j(p(\mathbf{u}))$ must be a consequence of a prefix of the path, $\mathcal{P}[\llbracket S \rrbracket]j$. Let i' be the Herbrand interpretation differing

from i only at $p(\mathbf{u})$. From Lemma 2.3, it follows that i' is terminating. Since $d_p(i, j)$ is minimal and $d_p(i, j) > 1$, we must have i and i' give the same final values for v since $d_p(i, i') = 1$ and thus, i' and j give different final values for v . But $d_p(i', j) < d_p(i, j)$, contradicting the minimality of $d_p(i, j)$. Therefore, $d_p(i, j) = 1$. \square

To make further progress with case (3) in the definition of the Weiser slice, it is required that S be liberal. We have not used that fact up to now.

LEMMA 3.5. Let a be a symbol and $a\sigma$ and $a\tau$ be liberal prefixes and v a variable. If $[\sigma]v \neq [\tau]v$ then $[a\sigma]v \neq [a\tau]v$.

Proof. Let $[\sigma]v \neq [\tau]v$. We show that $[a\sigma]v = [a\tau]v$ implies that either $[a\sigma]$ is not liberal or $[a\tau]$ is not liberal.

Clearly, a is not a predicate symbol, so assume $a = w := e$. By Definition 2.2, $[a\sigma]v = a([\sigma]v)$ and $[a\tau]v = a([\tau]v)$. Now, since $[\sigma]v \neq [\tau]v$, we must have w not in e , and e must be in one of $[\sigma]v$ or $[\tau]v$. Let e be in $[\sigma]v$ without loss of generality.

As e is an expression, there is an assignment symbol $v := e$ in σ with no assignments to any of the variables in e prior to $v := e$. This gives a prefix $\rho v := e$ of σ such that $[\rho v := e]v = e$. As w is not in e then $[w := e\rho v := e]v = e$. Then $\underline{w := e}$ and $\underline{w := e\rho v := e}$ are distinct prefixes of $a\sigma$ with

$$[\underline{w := e}]w = [\underline{w := e\rho v := e}]v,$$

contradicting the liberality of $a\sigma$. \square

LEMMA 3.6. (The prefixing lemma). Let $\rho\sigma$ and $\rho\tau$ be liberal prefixes and let v be a variable. If $[\sigma]v \neq [\tau]v$ then $[\rho\sigma]v \neq [\rho\tau]v$.

Proof. Follows immediately from Lemma 3.5 by induction. \square

PROPOSITION 3.2. Let S be a liberal free linear schema. Let p be a predicate name such that variable v needs p in S . If there exists a prefix of S of the form $\sigma p(\mathbf{w}) = X$ with function name f in $[\sigma]p(\mathbf{w})$ then v needs f in S .

Proof. Since v needs p there exist, by Lemma 3.4, two terminating Herbrand interpretations i and j differing only at p such that $(\mathcal{M}[\![S]\!]i)v \neq (\mathcal{M}[\![S]\!]j)v$ and $d_p(i, j) = 1$.

Let $\mathcal{P}[\![S]\!]i$ be of the form $\rho p(\mathbf{w}) = Y\tau$ and $\mathcal{P}[\![S]\!]j$ be of the form $\rho p(\mathbf{w}) = Y'\tau'$, where $[\rho](p(\mathbf{w}))$ is the unique term where i and j do not agree. By definition:

$$\begin{aligned} (\mathcal{M}[\![S]\!]i)v &= [\rho p(\mathbf{w}) = Y\tau]v \\ &\neq \\ [\rho p(\mathbf{w}) = Y'\tau']v &= (\mathcal{M}[\![S]\!]j)v. \end{aligned}$$

By Definition 2.2,

$$[p(\mathbf{w}) = Y\tau]v \neq [p(\mathbf{w}) = Y'\tau']v.$$

By freeness, both $\sigma p(\mathbf{w}) = Y\tau$ and $\sigma p(\mathbf{w}) = Y'\tau'$ are also paths of S . Therefore, by Lemma 3.6,

$$[\sigma p(\mathbf{w}) = Y\tau]v \neq [\sigma p(\mathbf{w}) = Y'\tau']v.$$

The paths $\sigma p(\mathbf{w}) = Y\tau$ and $\sigma p(\mathbf{w}) = Y'\tau'$ differ only at the term $[\sigma]p(\mathbf{w})$, since, by Lemma 3.6, if they differ at any other predicate terms, $\rho p(\mathbf{w}) = Y\tau$ and $\rho p(\mathbf{w}) = Y'\tau'$ will also differ. Since there exist paths differing only at the term $[\sigma]p(\mathbf{w})$, there must exist corresponding Herbrand interpretations differing only at the term $[\sigma]p(\mathbf{w})$. Hence, by Definition 3.10, v needs f in S . \square

This completes the proof that if S is a linear, free, liberal schema, and if a function name f ‘percolates’ to a needed predicate, as in case (3) of Weiser’s algorithm, f is also needed.

Our main result now follows.

THEOREM 3.1. *Weiser’s algorithm produces dataflow minimal end slices for programs which can be represented as schemas which are free and liberal.*

Proof. Weiser proved [1] that his algorithm always produces valid slices. In Lemma 3.2 it was shown that, for linear schemas S , if name n in S is needed for v , n is in every v -end-slice of S . For minimality, it suffices to show, therefore, that if $n \in W_v$ then n is needed. There are three cases to consider:

Case 1. If f is a function name in the term $[\sigma]v$ for some finite word, σ , of S , then by Definition 3.10, since S is free, f is needed.

Case 2. If p is a predicate name of S such that there is a function name f in the body of p the sub-schema guarded by p which is in W_v , p is needed by Proposition 3.1.

Case 3. If there exists a predicate, p in W_v such that for some prefix, σ , of S ending in $p(w_1 \cdots w_k) = X$ with f occurring in the term $[\sigma]p(w_1 \cdots w_k)$, v needs f by Proposition 3.2. \square

4. RELATED WORK ON SLICING AND ITS SEMANTICS

The literature contains many different definitions of a program slice. Slices can be *backward* or *forward* [44, 45],

static or *dynamic* [46, 47, 48, 49], *intraprocedural* or *interprocedural* [34, 44]. Slicing has been applied to programs with *arbitrary control flow* (goto statements) [50, 51, 52, 53] and even concurrent programming languages, such as Ada [54, 55]. Most forms of slicing use DefRef abstraction, though a few [35, 37] exploit more detailed information.

A *backward slice* is the ‘conventional one’ [1] where it is asked:

Which statements affect the slicing criterion?

Forward slicing [34] is the converse of this. The question asked in forward slicing is:

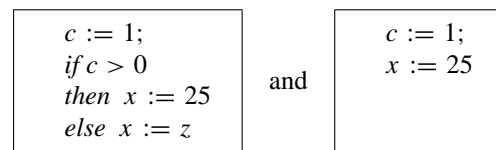
Given a particular statement in a program, which other statements are affected by this particular statement’s execution?

A *static slice* is the conventional one where the slice is required to agree with the program being sliced in all initial states. Dynamic slicing [46, 47, 48, 56, 57, 58] involves executing the program in a particular initial state and using trace information to construct a slice relevant to this particular initial state.

There are variants of slicing in between the two extremes of static and dynamic where some but not all properties of the initial state are known. These are known as quasi-static slicing [45], conditioned slicing [5, 59, 60] and constrained slicing [36].

Intraprocedural slicing means slicing programs which do not have procedures whereas *interprocedural* [33, 34, 44, 48, 61] slicing tackles the more complex problem of slicing programs where procedure definitions and calls are allowed.

This paper considers traditional (syntax-preserving) static backward slicing as introduced by Weiser. It also assumes that slicing algorithms use DefRef abstraction. In non-DefRef approaches [62, 37], infeasible paths are detected using a less abstract approach than DefRef analysis. Determining the fact that programs such as



are semantically equivalent, can in certain circumstances, be automated (although the general problem is clearly not computable).

4.1. The semantics of the PDG approach

Horwitz *et al.* [63] show that a program dependence graph (where the nodes contain the atomic statements and not just the defined and referenced variables) is an adequate structure for representing a program’s execution behaviour in the sense that two programs with the same program dependence graph have the same standard semantics. Reps and Yang [64] prove that the program dependence graph approach to slicing preserves Weiser’s semantics, i.e. it was shown that for any initial state where the original program terminates, the slice

1	<code>while(y>0)</code>
2	<code>do y := y+1;</code>
3	<code>x := 1;</code>

FIGURE 4. Non-termination preservation.

also terminates with the same sequence of values for each element of the slice. The converse is not true, i.e. in some states the slice may terminate when the original program does not.

4.2. Cartwright and Felleisen's work

Cartwright and Felleisen [65] define a *lazy semantics* of programs which they show is preserved by dataflow slicing algorithms like Weiser's algorithm [1] and the program dependence graph approach [66].

Lazy semantics is a term usually applied to functional languages. An interpreter that performs lazy evaluation will result in some programs terminating that would not do so if the opposite form of evaluation called *eager* evaluation were used. The reason this happens is that in lazy evaluation, when applying a function to some arguments, the arguments are evaluated only if their value is needed. In eager evaluation, however, the arguments are always evaluated before the function is applied. If evaluating an argument, therefore, leads to non-termination, and this argument is not needed, then eager evaluation will lead to non-termination but lazy evaluation may not.

The fact that slicing preserves lazy semantics has the consequence that slicing is allowed to *introduce termination*. Although lazy semantics is the norm for functional programming languages, it is not normally associated with the meaning of imperative programs, for which slicing is, almost exclusively, applied.⁴

Consider the example program in Figure 4. A static slice constructed with respect to $(x,3)$ will (conventionally) contain line 3 alone. The fact that line 3 will never be executed when y is initially >0 is of no consequence. In the lazy semantics of this program the final value of the variable x is 1, whatever the initial state. Harman *et al.* [67] show that slicing is also lazy with respect to faults and use this description to show how slicing algorithms can be modified to include faults in slices.

4.3. Venkatesh's work

The major aim of the work by Venkatesh [45] is to separate definitions of slices from the algorithms which compute them. He introduces and claims to formally define the semantics of a variety of already existing forms of slice as well as introducing some of his own. Slices are programs which preserve some projection of the semantics of the original program. Programs are all slices of themselves. The main contribution of Venkatesh's work is that it introduces the idea that there are many different feasible semantic definitions of a slice.

⁴Slicing has also been applied to functional style notations [69].

4.4. Hausler's work

Two years before Venkatesh, Hausler [68] stated the same definition of a slice as Weiser, namely that a slice S of P can be obtained from P by deleting zero or more statements and that if P halts on input i with values for the variables in the slicing criterion, so does S with the same values for these variables. Hausler, like Venkatesh and this paper, considers only end slicing. He gives a denotational definition of a slice. His definition is at the DefRef abstraction level. The strength of Hausler's work lies in the fact that he expresses a slicing algorithm without explicitly mentioning a control flow graph. His algorithm works directly on programs. He does not explicitly use data and control dependence but they are, nevertheless, encoded in his algorithm.

5. CONCLUSIONS AND FUTURE WORK

In all applications of slicing, the size of the slice is crucial. More the code removed by the slicing algorithm, the better. It is known that for programs as opposed to schemas statement minimal slices are not, in general, computable [1]. However, since Weiser posed the question in 1979, the question of dataflow minimality remained open [1]. This paper reformulates the dataflow minimal slicing question in terms of program schemas and proves that slicing algorithms do produce minimal slices for free liberal program schemas.

Future work will develop the schema-based theory to give semantic definitions of other forms of slicing and to formally analyse their properties in terms of schemas.

The theory of program schemas is a rich one. Its application to areas, such as program slicing has started to rekindle an interest in an area originally developed in the 1960s. Work by the authors [70, 71] indicates that the introduction of the linearity property, very natural in dataflow analysis, will lead to further positive results in program schematology.

For program dependence in general and program slicing in particular, it is now accepted that the most appropriate program semantics is a form of lazy or transfinite semantics which can 'look beyond' infinite loops [65, 72, 73]. One of our aims is to extend and generalize this semantics in terms of program schemas. We believe that this may lead to further insights into dataflow minimality and other areas of program dependence.

REFERENCES

- [1] Weiser, M. (1979) Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. PhD Thesis, University of Michigan, Ann Arbor, MI.
- [2] Canfora, G., Cimitile, A. and Munro, M. (1994) RE²: reverse engineering and reuse re-engineering. *J. Softw. Maint. Res. Pr.*, **6**(2), 53–72.
- [3] Simpson, D., Valentine, S. H., Mitchell, R., Liu, L. and Ellis, R. (1993) Recoup—maintaining fortran. *ACM Fortran forum*, **12**(3), 26–32.
- [4] De Lucia, A., Fasolino, A. R. and Munro, M. (1996) Understanding function behaviours through program slicing. In Proc. Proc. Proc. 4th IEEE Workshop on Program

- Comprehension*, Berlin, Germany, March 29–31, pp. 9–18. IEEE Computer Society Press, Los Alamitos, CA.
- [5] Harman, M., Hierons, R. M., Danicic, S., Howroyd, J. and Fox, C. (2001) Pre/post conditioned slicing. In *Proc. IEEE Int. Conf. Software Maintenance (ICSM'01)*, Florence, Italy, November 7–9, pp. 138–147. IEEE Computer Society Press, Los Alamitos, CA.
- [6] Canfora, G., Cimitile, A., De Lucia, A. and Di Lucca, G. A. (1994) Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'94)*, Victoria, Canada, September 19–23, pp. 424–433. IEEE Computer Society Press, Los Alamitos, CA.
- [7] Cimitile, A., De Lucia, A. and Munro, M. (1996) A specification driven slicing process for identifying reusable functions. *Softw. Maint. Res. Pr.*, **8**, 145–178.
- [8] Gallagher, K. B. (1992) Evaluating the surgeon's assistant: results of a pilot study. In *Proc. Int. Conf. Software Maintenance*, Orlando, FL, November 9–12, pp. 236–244. IEEE Computer Society Press, Los Alamitos, CA.
- [9] Gallagher, K. B. and Lyle, J. R. (1991) Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, **17**(8), 751–761.
- [10] Agrawal, H., DeMillo, R. A. and Spafford, E. H. (1993) Debugging with dynamic slicing and backtracking. *Software Pract. Exper.*, **23**(6), 589–616.
- [11] Kamkar, M. (1993) Interprocedural Dynamic Slicing with Applications to Debugging and Testing. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden.
- [12] Lyle, J. R. and Weiser, M. (1987) Automatic program bug location by program slicing. In *Proc. 2nd Int. Conf. Computers and Applications*, Peking, June, pp. 877–882. IEEE Computer Society Press, Los Alamitos, CA.
- [13] Weiser, M. and Lyle, J. R. (1985) Experiments on slicing-based debugging aids. Empirical studies of programmers, Soloway and Iyengar (eds). pp. 187–197. Molex.
- [14] Binkley, D. W. (1998) The application of program slicing to regression testing. In Harman, M. and Gallagher, K. (eds) *Inform. Software Tech.*, **40**, 583–594.
- [15] Gupta, R., Harrold, M. J. and Soffa, M. L. (1992) An approach to regression testing using slicing. In *Proc. IEEE Conf. Software Maintenance*, Orlando, FL, November 9–12, pp. 299–308. IEEE Computer Society Press, Los Alamitos, CA.
- [16] Harman, M. and Danicic, S. (1995) Using program slicing to simplify testing. *Softw. Test. Verif. Rel.*, **5**(3), 143–162.
- [17] Hierons, R. M., Harman, M. and Danicic, S. (1999) Using program slicing to assist in the detection of equivalent mutants. *Softw. Test. Verif. Rel.*, **9**(4), 233–262.
- [18] Hierons, R. M., Harman, M., Fox, C., Ouarbya, L. and Daoudi, M. (2002) Conditioned slicing supports partition testing. *Softw. Test. Verif. Rel.*, **12**, 23–28.
- [19] Beck, J. and Eichmann, D. (1993) Program and interface slicing for reverse engineering. In *Proc. IEEE/ACM 15th Conf. Software Engineering (ICSE'93)*, Baltimore, MD, May 17–21, pp. 509–518. IEEE Computer Society Press, Los Alamitos, CA.
- [20] Cimitile, A., De Lucia, A. and Munro, M. (1995) Identifying reusable functions using specification driven program slicing: a case study. In *Proc. IEEE Int. Conf. Software Maintenance (ICSM'95)*, Nice, France, October 16–20, pp. 124–133. IEEE Computer Society Press, Los Alamitos, CA.
- [21] Binkley, D. W., Horwitz, S. and Reps, T. (1995) Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Meth.*, **4**(1), 3–35.
- [22] Horwitz, S., Prins, J. and Reps, T. (1989) Integrating non-interfering versions of programs. *ACM Trans. Progr. Lang. Sys.*, **11**(3), 345–387.
- [23] Bieman, J. M. and Ott, L. M. (1994) Measuring functional cohesion. *IEEE Trans. Softw. Eng.*, **20**(8), 644–657.
- [24] Lakhota, A. (1993) Rule-based approach to computing module cohesion. In *Proc. 15th Conf. Software Engineering (ICSE-15)*, Baltimore, MD, May 17–21, pp. 34–44.
- [25] Ott, L. M. and Thuss, J. J. (1993) Slice based metrics for estimating cohesion. In *Proc. IEEE-CS Int. Metrics Symp.*, Baltimore, MD, May 21–22, pp. 71–81. IEEE Computer Society Press, Los Alamitos, CA.
- [26] Binkley, D. W. and Gallagher, K. B. (1996) Program slicing. In Zelkowitz, M. (ed.) *Advances in Computing* Vol. 43, pp. 1–50. Academic Press.
- [27] Binkley, D. W. and Harman, M. (2004) A survey of empirical results on program slicing. *Adv. Comput.*, **62**, 105–178.
- [28] De Lucia, A. Program slicing: methods and applications. In *Proc. 1st IEEE Int. Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 10, pp. 142–149. IEEE Computer Society Press, Los Alamitos, CA.
- [29] Harman, M. and Hierons, R. M. (2001) An overview of program slicing. *Softw. Focus*, **2**(3), 85–92.
- [30] Tip, F. (1995) A survey of program slicing techniques. *J. Progr. Lang.*, **3**(3), 121–189.
- [31] Luckham, D. C., Park, D. M. R. and Paterson, M. S. (1970) On formalised computer programs. *J. Comput. Sys. Sci.*, **4**, 220–249.
- [32] Laurence, M. R., Danicic, S., Harman, M., Hierons, R. and Howroyd, J. (2003) Equivalence of conservative, free, linear program schemas is decidable. *Theor. Comput. Sci.*, **290**, 831–862.
- [33] Weiser, M. (1984) Program slicing. *IEEE Trans. Softw. Eng.*, **10**(4), 352–357.
- [34] Horwitz, S., Reps, T. and Binkley, D. W. (1990) Interprocedural slicing using dependence graphs. *ACM Trans. Progr. Lang. Sys.*, **12**(1), 26–61.
- [35] Harman, M. and Danicic, S. (1997) Amorphous program slicing. In *Proc. 5th IEEE Int. Workshop on Program Comprehension (IWPC'97)*, Dearborn, MI, May 28–30, pp. 70–79. IEEE Computer Society Press, Los Alamitos, CA.
- [36] Field, J., Ramalingam, G. and Tip, F. (1995) Parametric program slicing. In *Proc. 22nd ACM Symp. Principles of Programming Languages*, San Francisco, CA, January 22–25, pp. 379–392. ACM Press, New York, NY, USA.
- [37] Snelting, G. (1996) Combining slicing and constraint solving for validation of measurement software. In *Static Analysis Symposium (SAS'96)*, Aachen, Germany, September 24–26, LNCS 1145, 332–348.
- [38] Manna, Z. (1974) *Mathematical Theory of Computation*. McGraw-Hill.
- [39] Greibach, S. (1975) *Theory of program structures: schemes, semantics, verification*. LNCS 36, Springer-Verlag Inc., New York, NY.
- [40] Paterson, M. S. (1967) Equivalence Problems in a Model of Computation. PhD Thesis, University of Cambridge, UK.
- [41] Hecht, M. S. (1977) *Flow Analysis of Computer Programs*. Elsevier.

- [42] Ferrante, J., Ottenstein, K. J. and Warren, J. D. (1987) The program dependence graph and its use in optimization. *ACM Trans. Progr. Lang. Sys.*, **9**(3), 319–349.
- [43] Danicic, S., Harman, M. and Sivagurunathan, Y. (1995) A parallel algorithm for static program slicing. *Inform. Process. Lett.*, **56**(6), 307–313.
- [44] Horwitz, S., Reps, T. and Binkley, D. W. (1988) Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, Atlanta, GA, June 20–24, pp. 25–46.
- [45] Venkatesh, G. A. (1991) The semantic approach to program slicing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Toronto, Canada, June 26–28, pp. 26–28.
- [46] Agrawal, H. and Horgan, J. R. (1990) Dynamic program slicing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, White Plains, NY, June 20–22, pp. 246–256. ACM Press, New York, NY, USA.
- [47] Gopal, R. (1991) Dynamic program slicing based on dependence graphs. In *Proc. IEEE Conf. Software Maintenance*, Sorrento, Italy, October 15–17, pages 191–200.
- [48] Kamkar, M., Shahmehri, N. and Fritzon, P. (1992) Interprocedural dynamic slicing. In *Proc. 4th Conf. Programming Language Implementation and Logic Programming*, Leuven, Belgium, August 26–28, pp. 370–384.
- [49] Korel, B. and Laski, J. (1988) Dynamic program slicing. *Inform. Process. Lett.*, **29**(3), 155–163.
- [50] Harman, M. and Danicic, S. (1998) A new algorithm for slicing unstructured programs. *J. Softw. Maint. Evol.*, **10**(6), 415–441.
- [51] Ball, T. and Horwitz, S. (1993) Slicing programs with arbitrary control-flow. In Fritzon, P. (ed.) *1st Conf. Automated Algorithmic Debugging*, Linköping, Sweden, May 3–5, pp. 206–222. Springer. Also available as University of Wisconsin–Madison, Technical report (in extended form), (1992) TR-1128.
- [52] Choi, J.-D. and Ferrante, J. (1994) Static slicing in the presence of goto statements. *ACM Trans. Progr. Lang. Sys.*, **16**(4), 1097–1113.
- [53] Agrawal, H. (1994) On slicing programs with jump statements. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Orlando, FL, June 20–24, pp. 302–312.
- [54] Cheng, J. (1993) Slicing concurrent programs—a graph-theoretical approach. In Fritzon, P. (ed.) *1st Automatic Algorithmic Debugging Conf. (AADEGUB'93)*, Linköping, Sweden, May 3–5, pp. 223–240. Springer.
- [55] Zhao, J., Cheng, J. and Ushijima, K. (1996) Static slicing of concurrent object-oriented programs. In *Proc. 20th IEEE Annual Int. Computer Software and Applications Conf. (COMPSAC'96)*, Seoul, Korea, August 19–23, pp. 312–320. IEEE Computer Society Press, Los Alamitos, CA.
- [56] Kamkar, M. (1998) Application of program slicing in algorithmic debugging. In Harman, M. and Gallagher, K. (eds), *Inform. Softw. Tech.*, **40**, 637–645.
- [57] Korel, B. (1995) Computation of dynamic slices for programs with arbitrary control flow. In Ducassé, M. (ed.) *2nd Int. Workshop on Automated Algorithmic Debugging (AADEBUG'95)*, Saint-Malo, France, May 22–24.
- [58] Korel, B. and Rilling, J. (1998) Dynamic program slicing methods. In Harman, M. and Gallagher, K. (eds) *Inform. Softw. Tech.*, **40**, 647–659.
- [59] Danicic, S., Fox, C., Harman, M. and Hierons, R. M. (2000) ConSIT: a conditioned program slicer. In *Proc. IEEE Int. Conf. Software Maintenance (ICSM'00)*, San Jose, CA, October 11–14, pp. 216–226. IEEE Computer Society Press, Los Alamitos, CA.
- [60] Canfora, G., Cimitile, A. and De Lucia, A. (1998) Conditioned program slicing. In Harman, M. and Gallagher, K. (eds) *Inform. Softw. Tech.*, **40**, pp. 595–607.
- [61] Ouarbya, L., Danicic, S., Daoudi, D. M., Harman, M. and Fox, C. (2002) A denotational interprocedural program slicer. In *Proc. IEEE Working Conf. Reverse Engineering (WCRE 2002)*, Richmond, VA, October 29–November 1, pp. 181–189. IEEE Computer Society Press, Los Alamitos, CA.
- [62] Krinke, J. and Snelling, G. (1988) Validation of measurement software as an application of slicing and constraint solving. In Harman, M. and Gallagher, K. (eds) *Inform. Softw. Tech.*, **40**, 661–675.
- [63] Horwitz, S., Prins, J. and Reps, T. (1988) On the adequacy of program dependence graphs for representing programs. In *Proc. Conf. Principles of Programming Languages, POPL '88*, San Diego, CA, January 13–15, pp. 146–157. ACM Press.
- [64] Reps, T. and Yang, W. (1988) *The Semantics of Program Slicing*. Technical Report 777, University of Wisconsin, USA.
- [65] Cartwright, R. and Felleisen, M. (1989) The semantics of program dependence. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Portland, OR, June 21–23, pp. 13–27. ACM Press, New York, NY, USA.
- [66] Ottenstein, K. J. and Ottenstein, L. M. (1984) The program dependence graph in software development environments. *ACM SIGPLAN Notices*, **19**(5), 177–184.
- [67] Harman, M., Simpson, D. and Danicic, S. (1996) Slicing programs in the presence of errors. *Formal Aspects of Computing*, **8**(4), 490–497.
- [68] Hausler, P. A. (1989) Denotational program slicing. In *Proc. 22nd Annual Hawaii Int. Conf. System Sciences*, Kailua-Kona, Hawaii, January 3–6, Volume II, pp. 486–495. ACM Press, New York, NY, USA.
- [69] Woodward, M. R. and Allen, S. P. (1998) Slicing algebraic specifications. *Inform. Softw. Tech.*, **40**(2), 105–118.
- [70] Laurence, M. R. (2004) Equivalence of Linear, Free, Liberal Program Schemas is Decidable in Polynomial Time. PhD Thesis, Goldsmiths College, University of London.
- [71] Laurence, M., Danicic, S., Harman, M., Hierons, R. M. and Howroyd, J. (2004) Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time. *Theor. Comput. Sci.*, submitted for publication.
- [72] Giacobazzi, R. and Mastroeni, I. (2003) Non-standard semantics for program slicing. *Higher-Order and Symbolic Computation*, **16**(4), 297–339.
- [73] Ouarbya, L. (2005) A Lazy Semantics for Program Slicing. PhD Thesis, Department of Computing, Goldsmiths College, University of London.