Static Program Slicing Algorithms are Minimal for Free Liberal Program Schemas

SEBASTIAN DANICIC[1], CHRIS FOX[2], MARK HARMAN[3], ROB HIERONS[4], JOHN HOWROYD[5] AND MICHAEL R. LAURENCE[6]

[1]*Dept. of Computing, Goldsmiths College, University of London, New Cross, London SE14 6NW, UK.*
[2]*Dept. of Computer Science, University of Essex, Colchester CO4 3SQ, UK.*
[3]*Dept. of Computer Science, King's College 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]*Dept. 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 of 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.

Keywords: program slicing, minimal slicing, static slicing

Received month date, year; revised month date, year; accepted month date, year

1. INTRODUCTION

In program slicing [69], 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 [13, 65], program comprehension [22, 34], software maintenance [12, 18, 26, 25], debugging [2, 45, 57, 71], testing [7, 30, 31, 39, 40], component re–use [5, 17], program integration [10, 42], and software metrics [6, 52, 59]. There are several surveys of slicing techniques, applications and variations [8, 9, 21, 35, 67].

In all applications of slicing, the size of the slice is crucial. The more code removed by the slicing algorithm the better. It is known however, that statement minimal slices are not, in general, computable [69]. However, since Weiser posed the question in 1979, the question of dataflow minimality has remained open [69]. 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 [56] are 'programs' where all expressions in the program have been replaced by *symbolic expressions* of the form $n(v_1, \ldots, v_m)$ where n is a function or predicate name and $v_1, \ldots, v_m$ are variables (see 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 [53]. 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.

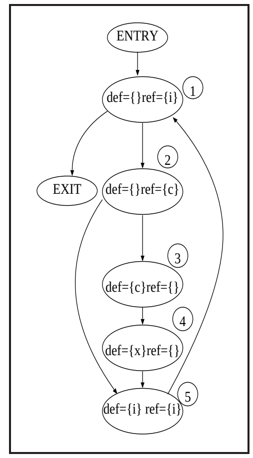
Weiser's dataflow based approach [69, 70] and the program dependence graph approach [44], produce the same slices for the same slicing criterion, so the results of this paper apply equally to both. Furthermore, both approaches operate at a level of abstraction where, in a program, the only information that can be utilised about each expression, e, is the set of variables

```
while i > 0
do
  begin

    if c = 1
    then

      begin
        c  := 4;

        x := 5
      end ;

    i := i − 1
  end
```

Program P_1

```
while p1(i)
do
  begin

    if p2(c)
    then

      begin
        c := f3();

        x := f4()
      end ;

    i := f5(i)
  end
```

Schema S_1



Annotated CFG

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

referenced by e. Weiser termed this approach *dataflow analysis*, but we call it *DefRef abstraction*[7], 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 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, Reps and Binkley [44], since this algorithm produces the same intraprocedural slices as Weiser's algorithm.

In this paper we shall be concerned only with end slicing [52], 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.

[7]Other approaches to program slicing exist [32, 24, 66, 12, 18] which do not use DefRef abstraction, but we are concerned with the theoretical properties of traditional static slicing which do.

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 statements 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 $c{:=}f_3()$ and so, 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. That is, 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 organised 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.

```
s := f1();
while b1(i)
do
begin
   s := f2(s,i);
   i := f3(i)
end
```

Schema S_2

```
s := 0;
while i > 0
do
begin
   s := s + i;
   i := i - 1
end
```

Program P_2

```
s := 1;
while i > 0
do
begin
   s := s * i;
   i := i - 1
end
```

Program P_2'

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

2. PROGRAM SCHEMAS

2.1. Basic Terminology

In this section, for completeness, the basic definitions and terminology used in the theory of program schemas introduced. Similar definitions can be found in Manna [58]. For a complete and excellent survey of the theory of program schemas see Greibach [29].

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

Statements
A statement, S, satisfies:

$$\begin{array}{lcl} S & \rightarrow & \textit{skip} \\ S & \rightarrow & V{:=}E \\ S & \rightarrow & \textit{if } E \textit{ then } S \textit{ else } S \\ S & \rightarrow & \textit{while } E \textit{ do } S \end{array}$$

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

The schemas S_1 and S_2 are called *sub-schemas* 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 sub-schema 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

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

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

$$\{\sigma_1 \cdots \sigma_n | \sigma_i \in \Sigma_i\}.$$

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

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

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

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

Following the usual conventions, we use $\underline{a}$ to represent both the symbol, $\underline{a}$ and also to represent 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, $S_1 S_2 \ldots S_r$ of statements:

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

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:

$$\begin{aligned}
&\Sigma(\mathit{skip}) = \{\lambda\}.\\
&\Sigma(v{:=}e) = \{\underline{v{:=}e}\}.\\
&\Sigma(\mathit{if}\ e\ \mathit{then}\ T_1\ \ \mathit{else}\ T_2) = \underline{e = \text{True}}\Sigma(T_1)\ \cup\\
&\qquad\qquad\qquad\qquad\qquad\quad \underline{e = \text{False}}\Sigma(T_2)\\
&\Sigma(\mathit{while}\ e\ \mathit{do}\ T) = [\underline{e = \text{True}}\Sigma(T)]^*\underline{e = \text{False}}
\end{aligned}$$

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 $0, \ldots, n$.

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
- Of the form $n(t_1, \cdots, t_k)$ where n is a name and $t_1, \cdots, t_k$ is a possible empty list of terms.

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

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

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

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 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, $\Delta\ e$, 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. i.e.

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

$$[a_1 \ldots a_k] = [a_1]\ \circ \ldots \circ\ [a_k]$$

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

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

The Semantics of Predicate Symbols
The meanings of $\underline{e = \text{True}}$ and $\underline{e = \text{False}}$ 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, $\pi'\underline{p = X}$, 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 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, on the other hand, 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 if 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 [63] 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 getter 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 $\sigma \underline{e_1 = X}$ and $\tau \underline{e_2 = Y}$ 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 $\sigma \underline{e_1 = X}$ and $\tau \underline{e_2 = Y}$ be distinct prefixes of the same prefix of S. Without loss of generality, let $\sigma \underline{e_1 = X}$ be a prefix of $\tau \underline{e_2 = Y}$. 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, $\tau \underline{e_2 = \neg Y}$ 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

$$\mathrm{Proj}_c(T) = \mathrm{Proj}_c(T')$$

$\mathrm{Proj}_c(T)$ is obtained first by deleting all elements of T whose label component is not i and then, by restricting the state components to V[8].

[8]This is a slight simplification of the true picture since we are

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-slice[9] of p with respect to a variable v if 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 sub-statements 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 either

1. $s = t$,
2. t is *skip*
3. 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 [56] (also see [58, 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 since this, by definition will result in a schema which is not a syntactic subset of the original. □

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

1. T is a syntactic subset of S
2. and T is a v-semantic-end-slice of S
3. and 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 [38] 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:

	while b do
	begin
A	...
Node n_2	$z{:=}x$;
.	.
.	.
.	.
Node n_1	$x{:=}y$;
B	...
	end

---

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

[9] This is our term - not Weiser's.

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 [23].

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 $\tau \underline{v{:=}f(\mathbf{y})}$ of a word of S with g occurring as the head name of an outermost sub term 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 [20, 44] 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 $\underline{p(w_1 \cdots w_k) = X}$, 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 whenever a term t does not contain the name n, 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

1. n is a function name in the term $(\mathcal{M}[\![S]\!]i)v$ for some terminating Herbrand interpretation i or

2. there exist two terminating Herbrand interpretations i, 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 $(\mathcal{M}[\![S]\!]i)v$ then 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, 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. □

In this section we show that for linear free schemas, names included by virtue of case 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 $X \in \{\text{True}, \text{False}\}$. We say $t = X$ is a consequence of the prefix $\sigma\underline{e = X}$ if and only if $[\sigma]e = t$.

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

Definition 3.13. *(Differing at t)*
Let $X, Y \in \{\text{True}, \text{False}\}$. 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.

□

Lemma 3.2 shows that if something is needed then it is contained in the end-slice. Weiser showed [69] 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 $w{:=}f(v_1 \cdots v_k)$ 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. □

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, Lemma 3.4 called the 'Difference Lemma': If S is free and v needs p then 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 liberal. The proof for the third case can be summarised 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 only differ 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, j differing only at p such that $(\mathcal{M}[\![S]\!]i)v \neq (\mathcal{M}[\![S]\!]j)v$ and $d_p(i,j) = 1$.*

Proof. Since p is a predicate name, p is not in the term $(\mathcal{M}[\![S]\!]i)v$ for any Herbrand interpretation i.

```
while i < 0
do
  begin
    if c = 3
    then
      begin
         c := c−v;
         x := 5
      end ;
    i := i + 1;
    c := c + k
  end
```

Program P_3

```
while p1(i)
do
  begin
    if p2(c)
    then
      begin
         c := f3(c, v);
         x := f4()
      end ;
    i := f5(i);
    c := f6(c, k)
  end
```

Schema S_3

```
while i < 0
do
  begin
    if c = 3
    then
      begin

         x := 5
      end ;
    i := i + 1;
    c := c + k
  end
```

Dataflow Minimal Slice

FIGURE 3. A non liberal example

Thus there exist terminating Herbrand interpretations i, j differing only on terms containing p such that $(\mathcal{M}[\![S]\!]i)v \neq (\mathcal{M}[\![S]\!]j)v$. Without loss of generality it may be assumed that, for all predicate names q that $i(q(\mathbf{t})) = \text{False} = j(q(\mathbf{t}))$ whenever $q(\mathbf{t}) = \text{True}$ is not a consequence of either of the paths, $\mathcal{P}[\![S]\!]i$ or $\mathcal{P}[\![S]\!]j$. Thus $d_p(i, j)$ is finite and hence it may be assumed that i and j have been chosen so that $d_p(i, j)$ is minimal. By 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, $\mathcal{P}[\![S]\!]i$, and similarly $p(\mathbf{u}) = j(p(\mathbf{u}))$ must be a consequence of a prefix of the path, $\mathcal{P}[\![S]\!]j$. 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 that 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, $d_p(i, j) = 1$. □

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

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

contradicting the liberality of $a\sigma$. □

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.

□

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

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

By Definition 2.2,

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

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

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

The paths $\sigma\underline{p(\mathbf{w}) = Y}\tau$ and $\sigma\underline{p(\mathbf{w}) = Y'}\tau'$ differ only at the term $[\sigma]\ p(\mathbf{w})$, since, by Lemma 3.6, if they differ at any other predicate terms then so do $\rho p(\mathbf{w}) = Y\tau$ and $\overline{\rho p(\mathbf{w})} = Y'\tau'$. 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. □

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, then 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 [69] 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 then 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 then 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 $[\sigma]\ \overline{p(w_1 \cdots w_k)}$ then v needs$f$ by Proposition 3.2.

□

4. RELATED WORK ON SLICING AND ITS SEMANTICS

The literature contains many different definitions of a program slice. Slices can be *backward* or *forward* [43, 68], *static* or *dynamic* [3, 28, 47, 49], *intra–procedural* or *inter–procedural* [44, 43]. Slicing has been applied to programs with *arbitrary control flow* (goto statements) [33, 4, 16, 1] and even concurrent programming languages like Ada [15, 73]. Most forms of slicing use DefRef abstraction, though a few [66, 32] exploit more detailed information.

A *backward slice* is the 'conventional one' [69] where it is asked:

> Which statements affect the slicing criterion?

Forward slicing [44] 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 [3, 28, 47, 46, 48, 50] 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 [68], conditioned slicing [34, 19, 11] and constrained slicing [24].

Intra-procedural slicing means slicing programs which do not have procedures whereas inter–procedural [70, 43, 44, 47, 62] 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 [51, 66], infeasible paths are detected using a less abstract approach than DefRef analysis. Determining the fact that programs like:-

```
c:=1;
if c > 0
then  x:=25
else  x:=z
```

and

```
c:=1;
x:=25
```

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. [41] 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 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 [14] define a *lazy semantics* of programs which they show is preserved by dataflow slicing algorithms like Weiser's Algorithm [69] and the program dependence graph approach [60].

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*

```
1  while(y>0)
2  do  y:=y+1;
3  x:=1;
```

FIGURE 4. Non-Termination Preservation

evaluation were used. The reason this happens is that in lazy evaluation, when applying a function to some arguments, the arguments are only evaluated if their value is need. In eager evaluation, on the other hand, 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*. While 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[10].

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 greater than 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, Danicic and Simpson [36] 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 [68] 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.

4.4. Hausler's Work

Two years before Venkatesh, Hausler [37] states 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, then so does S with the same values for these variables. Hausler, like Venkatesh and this paper, only considers 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.

[10] Slicing has also been applied to functional style notations [72].

5. CONCLUSIONS AND FUTURE WORK

In all applications of slicing, the size of the slice is crucial. The more 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 [69]. However, since Weiser posed the question in 1979, the question of dataflow minimality remained open [69]. 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 [54, 55] 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 [14, 27, 61]. One of our aims is to extend and generalise 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] Hiralal Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

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

[4] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control–flow. In Peter Fritzson, editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer. Also available as University of Wisconsin–Madison, technical report (in extended form), TR-1128,

December, 1992.

[5] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM 15th Conference on Software Engineering (ICSE'93)*, pages 509–518. IEEE Computer Society Press, Los Alamitos, California, USA, 1993.

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

[7] David Wendell Binkley. The application of program slicing to regression testing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

[8] David Wendell Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[9] David Wendell Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[10] David Wendell Binkley, Susan Horwitz, and Tom Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[11] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[12] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, September 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] Gerardo Canfora, Aniello Cimitile, and Malcolm Munro. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance : Research and Practice*, 6(2):53–72, 1994.

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

[15] Jingde Cheng. Slicing concurrent programs – a graph–theoretical approach. In Peter Fritzson, editor, *1st Automatic Algorithmic Debugging Conference (AADEGUB'93)*, pages 223–240, 1993. Appears as Springer Lecture Notes in Computer Science vol 749.

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

[17] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)*, pages 124–133, Nice, France, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[18] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice*, 8:145–178, 1996.

[19] Sebastian Danicic, Chris Fox, Mark Harman, and Rob Mark Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

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

[21] Andrea De Lucia. Program slicing: Methods and applications. In 1st *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[22] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In 4th *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[23] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[24] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In 22nd *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995.

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

[26] Keith Brian Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*, pages 236–244. IEEE Computer Society Press, Los Alamitos, California, USA, November 1992.

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

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

[29] Sheila Greibach. *Theory of program structures: schemes, semantics, verification*, volume 36 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1975.

[30] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Orlando, Florida, USA, 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[31] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

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

[33] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance and Evolution*, 10(6):415–441, 1998.

[34] Mark Harman, Rob Mark Hierons, Sebastian Danicic, John Howroyd, and Chris Fox. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Florence, Italy, November 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[35] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[36] Mark Harman, Dan Simpson, and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4):490–497, 1996.

[37] Philip A. Hausler. Denotational program slicing. In 22^{nd}*, Annual Hawaii International Conference on System Sciences, Volume II*, pages 486–495, January 1989.

[38] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, 1977.

[39] Robert Mark Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[40] Robert Mark Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, March 2002.

[41] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 146–157, New York, NY, USA, 1988. ACM Press.

[42] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

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

[44] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

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

[46] Mariam Kamkar. Application of program slicing in algorithmic debugging. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 637–645. Elsevier, 1998.

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

[48] Bogdan Korel. Computation of dynamic slices for programs with arbitrary control flow. In Mireille Ducassé, editor, 2^{nd} *International Workshop on Automated Algorithmic Debugging (AADEBUG'95)*, Saint–Malo, France, May 1995.

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

[50] Bogdan Korel and Jurgen Rilling. Dynamic program slicing methods. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 647–659. Elsevier, 1998.

[51] Jens Krinke and Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 661–675. Elsevier, 1998.

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

[53] Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd. Equivalence of conservative, free, linear program schemas is decidable. *Theoretical Computer Science*, 290:831–862, January 2003.

[54] Michael Rupen Laurence. *Equivalence of Linear, Free, Liberal Program Schemas is Decidable in Polynomial Time.* PhD thesis, Goldsmiths College, University of London, 2004.

[55] Mike Laurence, Sebastian Danicic, Mark Harman, Robert Mark Hierons, and John Howroyd. Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time. *Theoretical Computer Science*. Submitted 28th July 2004.

[56] D C Luckham, D M R Park, and M S Paterson. On formalised computer programs. *Journal of Computer and System Sciences*, 4:220–249, 1970.

[57] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[58] Zohar Manna. *Mathematical Theory of Computation.* McGraw–Hill, 1974.

[59] Linda M. Ott and Jeff J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Baltimore, Maryland, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

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

[61] Lahcen Ouarbya. *A Lazy Semantics for Program Slicing*. PhD Thesis, Department of Computing, Goldsmiths College, University of London, 2005.

[62] Lahcen Ouarbya, Sebastian Danicic, Dave (Mohammed) Daoudi, Mark Harman, and Chris Fox. A denotational interprocedural program slicer. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 181 – 189, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[63] M S Paterson. *Equivalence Problems in a Model of Computation*. PhD thesis, University of Cambridge, UK, 1967.

[64] Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[65] Dan Simpson, Samuel H. Valentine, Richard Mitchell, Lulu Liu, and Rod Ellis. Recoup – Maintaining Fortran. *ACM Fortran forum*, 12(3):26–32, September 1993.

[66] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis Symposium (SAS'96), LNCS 1145*, pages 332–348, 1996.

[67] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

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

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

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

[71] Mark Weiser and James R. Lyle. *Experiments on slicing-based debugging aids*, chapter 12, pages 187–197. Empirical studies of programmers, Soloway and Iyengar (eds.). Molex, 1985.

[72] Martin R. Woodward and Stephen P. Allen. Slicing algebraic specifications. *Information and Software technology*, 40(2):105–118, 1998.

[73] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Static slicing of concurrent object-oriented programs. In 20^{th} *IEEE Annual International Computer Software and Applications Conference (COMPSAC'96)*, pages 312–320, Seoul, Korea, August 1996. IEEE Computer Society Press, Los Alamitos, California, USA.