A Denotational Interprocedural Program Slicer

Lahcen Ouarbya, Sebastian Danicic & Mohamed Daoudi	Mark Harman	Chris Fox
Goldsmiths College	Brunel University	University of Essex
University of London	Uxbridge	Wivenhoe Park
New Cross	Middlesex	Colchester
London SE14 6NW	UB8 3PH	CO4 3SQ
United Kingdom	United Kingdom	United Kingdom

Keywords: Interprocedural, Program Slicing, side-effects, WSL

Abstract

This paper extends a previously developed intraprocedural denotational program slicer to handle procedures. Using the denotational approach, slices can be defined in terms of the abstract syntax of the object language without the need of a control flow graph or similar intermediate structure.

The algorithm presented here is capable of correctly handling the interplay between function and procedure calls, side-effects, and short-circuit expression evaluation.

The ability to deal with these features is required in reverse engineering of legacy systems, where code often contains side-effects.

1 Introduction

Hausler [23] presents a denotational program slicer for a very simple programming language without procedures. This paper extends Hausler's approach to a more realistic programming language containing (possibly side-effecting,but non-recursive) functions which can be called both as expressions and as statements.

In program slicing, 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, called the *slicing criterion*. Like program optimising, slicing can be thought of as a transformation that preserves certain semantic properties. Slicing has many applications including reverse engineering [8, 36], program comprehension [14, 22], software maintenance [7, 10, 15, 16], debugging [1, 29, 31, 42], testing [4, 18, 19, 25, 26], component re–use [2, 9], program integration [6, 27], and software metrics [3, 30, 34]. There are several surveys of slicing techniques, applications and variations [5, 13, 20, 37].

Slicing across functions and function-calls is complicated due to the problems of side-effects that can be generated by an expression [5, 20, 37]. Published approaches [12, 28, 40, 41] do not explicitly mention how side-effects are handled. The main contribution of this paper is to describe a denotational interprocedural slicing algorithm for programs in the presence of side-effects. We only consider end slicing. i.e. the slicing criterion is $<p, V>$, where V is the set of variable of interest and p is the end program point. Slicing in the presence of side-effects is complicated because of the necessity to translate the slicing criterion into and out of function calls.

```
int x, y, z;

int f(){
  y= 1;
  z =1;
  return z;}

main(){
   x =f();
   printf("%d",z);}
```

Original program P_1

```
int x, z;

int f(){

  z = 1;
  return z;}

main(){
   x = f();
   printf("%d", z);}
```

Slice of P_1 w.r.t. z

Figure 1. Program P_1 and its corresponding slice w.r.t z

The slice of an assignment `x = y;` with respect to a variable `z` is empty. The resulting slice of an assignment `x = f();` with respect to a variable `z` is not always empty as there are two cases to consider: The first one is when the function `f` is side-effect-free. In this case, the resulting

slice with respect to `z` is empty. However, if `f` contains a side-effect on `z`, the assignment `x =f();` is kept in the slice but we must still slice the body of the function `f` (keeping only the statements which affect the final value of the variable `z`).
In Figure 1, the right most fragment shows the resulting slice of the program P_1 with respect to `z`. The assignment `y = 1;` in the body of the function `f` is deleted, because it does not contribute the the final value of the variable `z`.

The rest of the paper is organised as follows: In Section 2, we describe Hausler's approach by defining his slicer for a simple intraprocedural side-effect free language. The main denotational algorithm for interprocedural slicing of programs with side-effects is given in Section 3. In Sections 4 and 5, we briefly discuss our implementation and related work. We conclude with Section 6 which gives directions for future work.

2 Intraprocedural side-effect Free Slicing

In order to demonstrate the denotational approach introduced by Hausler [23], in Figure 2, we give his definition of an intraprocedural slicer for a simple notional language consisting of loops, conditionals and assignments. This language has no function calls and no side-effects. He uses a functional notation where we define the slice directly in terms of each syntactic category of the language rather than first converting programs to control flow graphs as in [12]. Two functions are required:

$\mathfrak{S}(P, V)$, the slice of program P with respect to the set of variables V, which takes a statement P and a set of variables V and returns the resulting slice of P with respect to V.

$\mathfrak{N}(P, V)$, the needed set with respect to the set of variables, V, of program P. This is a function which returns the set of variables, whose initial values affect the final values of variables in V when P is executed.

In Figure 2, we define $\mathfrak{S}(P, V)$ and $\mathfrak{N}(P, V)$ for side-effect free programs in a case-by-case basis, in terms of the structure of the syntax. The key to understanding this algorithm are rules (5) and (6). When slicing a sequence $S_1; S_2$ with respect to a set of variables V (the slicing criterion) first the needed set W of S_2 with respect to V is computed and following this, the needed set of S_1 with respect to W is computed. In this way, the needed set 'flows backwards' through the program being transformed 'as it goes' (rule 5). As the needed set 'flows through' each statement, the statement is sliced (rule 6). Rules (1) - (4) are for slicing atomic statements. For assignments, $x = e$, there are two cases for $\mathfrak{N}(x = e, V)$: The first is when x is not in the set, V flowing through $x = e$. In this case, the assignment is removed and V 'flows through' unchanged. The second case is when x is in V. We transform the set flowing through $x = e$ by removing x and adding all the variables referenced by the expression e. In this case $x = e$ remains in the slice.

For conditionals (rules 7 and 8) we slice both the true and false parts and keep the conditional if and only if either of these slices is non-empty. The needed set of conditionals is the union of the needed sets of its components together with the variables referenced by the predicate. This corresponds to control dependence [24].

For loops of the form [`while` (B) S], the needed set of variables is produced by repeatedly calculating the needed set of variables of [`if` (B) S] and feeding this value backward until there is no further change. This is expressed in rule 9.

First : $\mathbf{N} \times [\mathbf{E}] \rightarrow \mathbf{E}$
First$(i, L) = L[1], \ldots, L[i-1]$

Eval : $[\mathbf{E}] \rightarrow \mathbf{E}$
Eval$(L) = L[1], \ldots, L[\text{length}(L)]$

$\mathbf{temp}_V$: $\mathbf{V} \times \mathbf{Names} \rightarrow \mathbf{V}$
$\mathbf{temp}_V(x, f) \;=\; x_f$

$\mathbf{Temp}_L$: $[\mathbf{V}] \times \mathbf{Names} \rightarrow [\mathbf{V}]$
$\mathbf{Temp}_L(L, f) =$
$[\mathbf{temp}_V(L[1], f), \ldots, \mathbf{temp}_V(L[length(L)], f)]$

$\mathbf{Temp}_S$: $[\mathbf{V}] \times \mathbf{Names} \rightarrow \mathcal{P}(\mathbf{V})$
$\mathbf{Temp}_S(L, f) =$
$\{\mathbf{temp}_V(L[1], f), \ldots, \mathbf{temp}_V(L(length(L)), f)\}$

Rename : $[\mathbf{V}] \times [\mathbf{V}] \times \mathbf{S} \rightarrow \mathbf{S}$
Rename$(L_1, L_2, S) = [L_1 \rightarrow L_2]S$
where $[L_1 \rightarrow L_2]S$ means for each i, substitute $L_1[i]$ by $L_2[i]$ in S

Figure 3. Auxiliary functions

3 Interprocedural Slicing

In many languages, function calls can occur either as expressions or as a statements. Since such function calls may have side-effects, we need to be able to slice *expressions* as well as *statements*.

In this paper, the characteristics of the subroutines are based on those in WSL [33]. In WSL, functions may have both *value* and *var* parameters. Multiple *return* statements are not allowed. The value returned by a function is give by a single expression occurring at the end of the function's body. Our system cannot at present handle recursive functions or multiple returns.

We consider function calls to be atomic and therefore will remain in their entirety or be completely deleted. However, it is the bodies of the functions which can be simplified in the case where at least one of their corresponding calls

$\mathfrak{N} : \mathbf{S} \times \mathcal{P}(\mathbf{V}) \to \mathcal{P}(\mathbf{V})$
$\mathfrak{S} : \mathbf{S} \times \mathcal{P}(\mathbf{V}) \to \mathbf{S}$
$\mathrm{Ref} : \mathbf{E} \to \mathcal{P}(\mathbf{V})$The set of variables upon which the value of E depends.

`skip`

(1) $\mathfrak{N}(\texttt{skip}, V) \triangleq V$
(2) $\mathfrak{S}(\texttt{skip}, V) \triangleq \texttt{skip}$

Assignments

(3) $$\mathfrak{N}(x = e,\ V) \triangleq \begin{cases} V & \text{if } x \notin V \\ (V \setminus \{x\}) \cup \mathrm{Ref}(e) & \text{if } x \in V \end{cases}$$

(4) $$\mathfrak{S}(x\ = e,\ V) \triangleq \begin{cases} \texttt{skip} & \text{if } x \notin V \\ x{=}e & \text{if } x \in V \end{cases}$$

Sequences of statements

(5) $\mathfrak{N}(S_1; S_2\ , V) \triangleq \mathfrak{N}(S_1, \mathfrak{N}(S_2, V)))$
(6) $\mathfrak{S}(S_1; S_2\ , V) \triangleq \mathfrak{S}(S_1, \mathfrak{N}(S_2, V));\ \mathfrak{S}(S_2, V)$

Conditionals

(7) $$\mathfrak{N}(\texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2, V) \triangleq \begin{cases} V & \text{if } \mathfrak{S}(S_1, V) = \mathfrak{S}(S_2, V) = \texttt{skip} \\ \mathfrak{N}(S_1, V)\ \cup\ \mathfrak{N}(S_2, V)\ \cup\ \mathrm{Ref}(B) & \text{if } \mathfrak{S}(S_1, V) \neq \texttt{skip} \text{ or } \mathfrak{S}(S_2, V) \neq \texttt{skip} \end{cases}$$

(8) $$\mathfrak{S}(\texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2, V) \triangleq \begin{cases} \texttt{skip} & \text{if } \mathfrak{S}(S_1, V) = \mathfrak{S}(S_2, V) = \texttt{skip} \\ \texttt{if}\ (B)\ \mathfrak{S}(S_1, V)\ \texttt{else}\ \mathfrak{S}(S_2, V) & \text{if } \mathfrak{S}(S_1, V) \neq \texttt{skip} \text{ or } \mathfrak{S}(S_2, V) \neq \texttt{skip} \end{cases}$$

Whiles

(9) $\mathfrak{N}(\texttt{while}\ (B)\ S\ \ , V) \triangleq \text{fix } \lambda y \cdot \mathfrak{N}(\texttt{if}\ (B)\ S\ \texttt{else skip}, y \cup V)$

(10) $$\mathfrak{S}(\texttt{while}\ (B)\ S, V) \triangleq \begin{cases} \texttt{skip} & \text{if } \mathfrak{S}(S, V) = \texttt{skip} \\ \texttt{while}(B)\ \mathfrak{S}(S, \mathfrak{N}(\texttt{while}\ (B)\ S\ \ , V)) & \text{if } \mathfrak{S}(S, V) \neq \texttt{skip} \end{cases}$$

Figure 2. Intraprocedural Slicer

remains in the slice.

Slicing across functions and functions-calls is complicated by side-effects. An expression E can have side-effects upon the set of variables of interest. Therefore we need to work out, $\mathfrak{N}(E, V)$, the set of variables whose initial values affect the final value of variables in V when E is executed and the set, $\mathfrak{D}(E)$ of variables whose initial values determine the outcome of the value of the expression E.

3.1 Slicing Expressions

The algorithm for slicing expressions is given in Figure 4. It shows how to work out $\mathfrak{N}$ and $\mathfrak{D}$ for different types of expressions. We now consider each case in turn:

3.1.1 Slicing Compound Expressions

Since expressions can have side-effects, the order in which expressions are evaluated has to be considered. Therefore in order to determine $\mathfrak{D}(E)$ and $\mathfrak{N}(E, V)$, for compound expressions we need to know the order in which the sub-expressions of E are evaluated. We assume sub-expressions are evaluated form left to right. Slicing compound separated expressions involves considering the three followings cases:

1. Comma operator separated expressions
   Comma expressions are always evaluated from left to right. $\mathfrak{N}(E, V)$ and $\mathfrak{D}(E)$ for comma expressions is given by formulae (5) and (6).

2. Arithmetic operator separated expressions
   Rules (7) and (8) in Figure 4 show $\mathfrak{N}(E, V)$ and $\mathfrak{D}(E)$ for arithmetic operator separated expressions.

3. Boolean operator separated expressions
   For Boolean expressions, the issue of side-effects is complicated further by short circuit evaluation. In evaluating the boolean expression B_1 op B_2, it is possible that only B_1 gets evaluated. In Figure 4, rules (9) and (10) shows how to work out $\mathfrak{N}$ and $\mathfrak{D}$ of Boolean operator separated expressions.

$\mathfrak{N} : \mathbf{E} \times \mathcal{P}(\mathbf{V}) \to \mathcal{P}(\mathbf{V})$
$\mathfrak{D} : \mathbf{E} \to \mathcal{P}(\mathbf{V})$
$\alpha : \mathbf{Names} \to \mathcal{P}(\mathbf{V})$
$\beta : \mathbf{Names} \to \mathcal{P}(\mathbf{V})$
$\mathbf{SideEffect} : \mathbf{E} \times \mathcal{P}(\mathbf{V}) \to \{T, F\}$

Constants

(1) $\mathfrak{N}(E, V) \triangleq V$
(2) $\mathfrak{D}(E) \triangleq \emptyset$

Variables

(3) $\mathfrak{N}(E, V) \triangleq V$
(4) $\mathfrak{D}(E) \triangleq E$

Comma Expressions

(5) $\mathfrak{N}(E_1, E_2, V) \triangleq \mathfrak{N}(E_1, \mathfrak{N}(E_2, V))$
(6) $\mathfrak{D}(E_1, E_2) \triangleq \mathfrak{D}(E_1) \cup \mathfrak{N}(E_1, \mathfrak{D}(E_2))$

Arithmetic Expressions

(7) $\mathfrak{N}(E_1 \text{ op } E_2, V) \triangleq \mathfrak{N}((E_1, E_2), V)$
(8) $\mathfrak{D}(E_1 \text{ op } E_2) \triangleq \mathfrak{D}((E_1, E_2), V)$

Boolean Expressions

(9) $\mathfrak{N}(B_1 \text{ op } B_2, V) \triangleq \mathfrak{N}(B_1, V) \cup \mathfrak{N}(B_1, B_2, V)$
(10) $\mathfrak{D}(B_1 \text{ op } B_2) \triangleq \mathfrak{D}(B_1, B_2)$

Function Calls

(11) $\mathfrak{N}(f(A_{val}, A_{var}), V) \triangleq \mathfrak{N}(A_{val},\ (V' \setminus \mathbf{Temp}_S(F_{val}, f)) \ \cup \ \bigcup_{i=1}^{n}(C_i)$

$$\text{where} \begin{cases} V' = \mathfrak{N}(S', \mathfrak{N}(e, V)) \\ S' = \text{Rename}(F_{var}F_{val}, A_{var}\mathbf{Temp}_L(F_{val}, f), S) \\ S \text{and } e \text{ are the body and the return expression of } f \\ C_i = \begin{cases} Needed(\mathbf{First}(i, A_{val}), \mathfrak{D}(A_{val}[i])) & \text{if } \mathbf{Temp}_L(F_{val}, f)[i] \in V' \\ \emptyset & \text{otherwise} \end{cases} \end{cases}$$

(11.1) $\alpha(f) \to \alpha(f) \cup rename(temps(F_{val}, f), F_{val}, \mathfrak{S}(S', V'))$

(12) $\mathfrak{D}(f(A_{val}, A_{var}), V) \triangleq \mathfrak{D}(\mathbf{Eval}(A_{val}),\ (V' \setminus \mathbf{Temp}_S(F_{val}, f)) \ \cup \ \bigcup_{i=1}^{n}(C_i)$

$$\text{where} \begin{cases} V' = \mathfrak{N}(S', \mathfrak{D}(e')) \\ e' = \text{Rename}(F_{var}F_{val}, A_{var}\mathbf{Temp}_L(F_{val}, f), e) \\ S',\ e \text{ and } C_i \text{ are defined above} \end{cases}$$

(12.1) $\alpha(f) \to \alpha(f) \cup rename(temps(F_{val}, f), F_{val}, \mathfrak{S}(S', V'))$

(13) $\mathbf{SideEffect}(f(A_{val}, A_{var}), V) \triangleq \begin{cases} T & \text{if } \mathfrak{S}(S', V') \neq \texttt{skip} \text{ or } \exists\, i \text{ such that } \mathbf{SideEffect}(A_{val}[i], V) \text{ where} \\ & S' \text{and} V' \text{ are defined in rule 11.} \\ F & \text{otherwise} \end{cases}$

Figure 4. Interprocedural Slicing of Expressions

3.1.2 Slicing Function Call Expressions

A function call expression is of the form $f(A_{val}, A_{var})$, where A_{val} and A_{var} are respectively the lists of actual value and var parameters. A function definition is of the form $f(F_{val}, F_{var}, S, e)$, where F_{val}, F_{var}, S and e are, respectively, the lists of formal value and var parameter, the body of the function and the the return expression[1] of the function.

Slicing a function call (Figure 4) involves computing $\mathfrak{N}$ and $\mathfrak{D}$ for the function call and slicing the corresponding function definition.

$\mathfrak{N}(E, V)$ for expression of form $f(A_{val}, A_{var})$ is given by formula (11) in Figure 4. Let S and e be the body and the return expression of the function f: First we tag every formal value-parameter occurring in S and then replace every formal var-parameter occurring in S by its corresponding actual one. As a result of doing this, we get a new statement S'. We then slice S' with respect to $\mathfrak{N}(e, V)$ to get V'. In order to proceed, we remember that evaluating the actual value-parameters of a function call can contain a side-effect. Each element of V' which is not a tagged value parameter can be affected by whole the list A_{val}. Every tagged element in V' can only be affected by the actuals evaluated before it. This is captured by the C_i in Figure 4.

Rule (12) in Figure 4 computes $\mathfrak{D}$(E) for function calls. It is almost identical to computing $\mathfrak{N}(E, V)$. The same tagging occurs. The only difference is that, in this case, $V' = \mathfrak{N}(S', \mathfrak{D}(e))$.

Each time a function call $f(A_{val}, A_{var})$ is encountered, the body of its corresponding function definition has to be sliced with respect to a set of variables V, taking into account the relationship between formal parameters and actual parameters. To be able to slice the function definition f, a record, $\alpha(f)$, is kept of all statements in the body of the function f, needed at each of the corresponding calls. During slicing each time we come across a call to a function f, $\alpha(f)$ is updated to reflect that parts of f's body that are needed in the slice. Rules (11.1) and (12.1) in Figure 4 show how $\alpha(f)$ is updated each time a function call is sliced.

3.2 Interprocedural Slicing of Statements

Before we can discuss the interprocedural slicing of statements, we need to define how to ascertain whether an expression E has a side-effect on a set of variables V (See Rule 13 in Figure 4). If E does not contain a function call it cannot have a side-effect on V. The only way that E can side-effect an element of V is if at least one of the function calls in E has a side-effect on an element of V. A function call has a side-effect on an element of V if and only if when we tag the body (as described in Section 3.1.2) and slice it with respect to V we do not get `skip`. The slice($\mathfrak{S}$) and needed set($\mathfrak{N}$) with respect to set of variables V of each form of statement is given in Figure 5.

[1] We only consider functions with one return statement.

3.2.1 Assignment Statements

Rules (1) and (2) in Figure 5 compute $\mathfrak{N}(x = e, V)$ and $\mathfrak{S}$(x=e,V). we have three cases to consider

1. If x is not an element of V and e has no side-effect on V, then $\mathfrak{N}(x = e, V)$ is just V. The assignment, in this case is deleted.

2. If x is not an element of V and e may have a side-effect on V, then $\mathfrak{N}(x = e, V) = \mathfrak{N}(e, V)$. In this case the assignment is kept in the slice.

3. If x is an element of V, then to compute $\mathfrak{N}(x = e, V)$ we remove x from V and add in all the variables affecting the final value of e. In this case, the assignment is kept in the slice.

3.2.2 Conditionals

Rule (3) and (4) in Figure 5 compute the needed set and the slice for an `if` statement. First we slice both the true and the false parts with respect to V. Again there are three cases to consider:

1. If the slices of both subcomponents are empty and B as no side-effect on V, then the whole `if` statement is deleted and the needed set of variables is just V.

2. If the slices of both subcomponents are empty and B may have a side-effect on V, then the needed set of variables is $\mathfrak{N}(B, V)$ and the resulting slice is just `if` (B) In this case the needed set of variables is $\mathfrak{N}(B, V)$.

3. If either of the slices of the subcomponents are not empty, then the resulting slice is `if` (B) $\mathfrak{S}(S_1, V)$ `else` $\mathfrak{S}(S_2, V)$, and the needed set of variables is $\mathfrak{N}(S_1, V) \cup \mathfrak{N}(S_2, V) \cup \mathfrak{D}(B)$

3.2.3 Loops

Interprocedural slicing of loops is similar to the intraprocedural case. The `while` loop can be deleted in the case where neither the predicate nor the body affect the set of variables V. In the case either the predicate or the body affects V we must consider the affect of the predicate on the set of variables, V as well as the variables that determine the value of the predicate (rules 5 and 6).

Assignments

$$(1)\quad \mathfrak{N}(x = e,\ V) \triangleq \begin{cases} V & \text{if } x \notin V \ \wedge\ \neg\textbf{SideEffect}(e,V) \\ \mathfrak{N}(e,V) & \text{if } x \notin V \ \wedge\ \textbf{SideEffect}(e,V) \\ \mathfrak{N}(e, V\setminus\{x\}) \ \cup\ \mathfrak{D}(e) & \text{if } x \in V x \in V \end{cases}$$

$$(2)\quad \mathfrak{S}(x = e\ , V) \triangleq \begin{cases} \texttt{skip} & \text{if } x \notin V \ \wedge\ \neg\textbf{SideEffect}(e,V) \\ x = e\text{otherwise} \end{cases}$$

Conditionals

$$(4)\quad \mathfrak{N}(\texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2, V) \triangleq \begin{cases} V & \text{if } \mathfrak{S}(S_{i_{(1,2)}}, V) = \texttt{skip} \wedge \neg\textbf{SideEffect}(B,V) \\ \mathfrak{N}(B,V) & \text{if } \mathfrak{S}(S_{i_{(1,2)}}, V) = \texttt{skip} \ \wedge\ \textbf{SideEffect}(B,V) \\ \mathfrak{N}(B, \mathfrak{N}(S_1,V)\ \cup\ \mathfrak{N}(S_2,V)) \cup \mathfrak{D}(B) & \text{if } \mathfrak{S}(S_{i_{(1,2)}}, V) \neq \texttt{skip} \end{cases}$$

$$(5)\quad \mathfrak{S}(\texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2, V) \triangleq \begin{cases} \texttt{skip} & \text{if } \mathfrak{S}(S_{i_{(1,2)}}, V) = \texttt{skip} \wedge \neg\textbf{SideEffect}(B,V) \\ \texttt{if}\ (B)\ \mathfrak{S}(S_1,V)\ \texttt{else}\ \mathfrak{S}(S_2,V) & \text{otherwise} \end{cases}$$

Whiles

$$(5)\quad \mathfrak{N}(\texttt{while}\ (B)\ S\ \ , V) \triangleq \begin{cases} V & \text{if } \neg\textbf{SideEffect}(B,V)\ \wedge\ \mathfrak{S}(S,V) = \texttt{skip} \\ \text{fix}\ \ \lambda y \cdot \mathfrak{N}(\texttt{if}\ (B)\ S\ , y \cup (\mathfrak{N}(B,V) \cup \mathfrak{D}(B)) & \text{otherwise} \end{cases}$$

$$(6)\quad \mathfrak{S}(\texttt{while}\ (B)\ S, V) \triangleq \begin{cases} \texttt{skip} & \text{if } \neg\textbf{SideEffect}(B,V)\ \wedge\ \mathfrak{S}(S,V) = \texttt{skip} \\ \texttt{while}\ (B)\ \mathfrak{S}(S, \mathfrak{N}(\texttt{while}\ (B)\ S\ \ , V)) & \text{otherwise} \end{cases}$$

Expression Statements

$(7)\mathfrak{N}(e,V)$ (See Figure 4)

$$(8)\mathfrak{S}(e,V) \triangleq \begin{cases} \texttt{skip} & \text{if } \neg\textbf{SideEffect}(E,v) \\ e & \text{otherwise} \end{cases}$$

Function Definitions

$$(11)\quad \mathfrak{S}(f(A_{val}, A_{var}, S, e)\ , V) \triangleq \begin{cases} \texttt{skip} & \text{if there is no call to the function } f \text{ the slice} \\ f(F_{var}, fval, \alpha(f), e) & \text{otherwise} \end{cases}$$

where $\alpha(f)$ is defined in Figure 4.

Figure 5. Interprocedural Slicing of Statements

3.2.4 Slicing Expression Statements

To compute the needed set of an expression statement we just use the rules for calculating $\mathfrak{N}$ for expressions given in Figure 4.

To compute $\mathfrak{S}$ for expression statements, we simply keep the expression statement in the slice if and only if it may have a side-effect on V. This function is defined in Figure 5.

3.2.5 Slicing Function Definitions

To slice a function definition we simply take the union of the slices of the function definition corresponding to each call to it (Rule 11 in Figure 5). If there is no call then the function definition will not be included.

4 The Implementation

The implementation of our slicing algorithm was achieved using a language called *WSL* [33]. *WSL* is both the language that the slicer was written as well as the object language to be sliced. The reason for our choice is that *WSL* has a built in *WSL* parser that can be called from within a *WSL* program as well as a whole transformation system which is useful for the simplification of slices. Transformations were not a major design criterion of most popular programming languages, they are difficult to transform. However, *WSL* (wide spectrum language) and transformation theory form the basis of the 'Maintainer's Assistant'tool [39] used for analysing programs by transformations. *WSL* is also the basis of the FermaT transformation system [38]. The FermaT transformation system applies correctness-preserving transformations to programs written in *WSL* language. It is an industrial-strength engine with many applications in program comprehension and language migration, it has been used in migration IBM assembler to C and to COBOL [32]. Low-level programming constructs and high-level abstract specifications are both included in *WSL* language; hence the transformation of a program from abstraction specification to a detailed implementation can be expressed in a single language. The syntax and semantics of *WSL* are described in [33].

5 Related Work

Hausler [23] presents a denotational program slicer for a very simple programming language without procedures.

The System Dependence Graph *SDG*, [35] approach contains a program dependence graph for each procedure in the program. The *SDG* contains additional nodes to model procedure calls and parameter passing. Parameters are passed by value-result and access to global variables is achieved via additional parameters of the procedure:

- Call-site nodes represent the call site.
- Actual-in and actual-out represent the input and output parameters at the call sites. They are control dependent on the call site node.
- Formal-in and formal-out represent the input and output parameters at the called procedure, they are control dependent on the procedure entry node.

In order to link the call site with the program dependence graph of its corresponding procedure, additional edges have been introduced:

- Call edges link the call-site nodes with the procedure entry node.
- Parameter-in edges link the actual-in with the formal-in nodes.
- Parameter-out edges link the formal-out with the actual-out nodes
- Summary edges represent the transitive dependence due to the calls.

Interprocedural slicing using *SDG* is implemented as reachability over the the *SDG*.

CodeSurfer [17] is a commercial interprocedural slicing tool based on reachability problem over *SDG*. In order to slice a program P using with respect to a program point p and a variable x, CodeSurfer highlights all codes of the program P that affect the value of the variable of interest x at the program point p.

5.1 Advantages of Our Approach

The advantage of the denotational approach is that slicing can be expressed as mathematical transformations on abstract syntax without the need to introduce intermediate structures such as control flow graphs. Such definitions are highly amenable both to correctness proof and implementation in the functional style.

Our algorithm not only returns an executable program, but also produces variable dependence information which is very useful in comprehension and testing [21].

6 Conclusion and Future Work

In this paper we have denotationally defined an algorithm for the interprocedural slicing of program with side-effects. Slicing across functions and function-calls is complicated due to the problems of side-effects that can be generated by an expression. This is the first published algorithm to address this problem. As a proof of concept, we have implemented the slicer for a large subset of the language WSL. Our slicer is written in WSL.

Future work will attempt to prove correctness of the slicer and extend it to handle other features. These include programs with local scope, multiple return statements and recursive functions which are not handled at present.

References

[1] AGRAWAL, H., DEMILLO, R. A., AND SPAFFORD, E. H. Debugging with dynamic slicing and backtracking. *Software Practice and Experience 23*, 6 (June 1993), 589–616.

[2] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)* (1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 509–518.

[3] BIEMAN, J. M., AND OTT, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[4] BINKLEY, D. W. The application of program slicing to regression testing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 583–594.

[5] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. In *Advances of Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[6] BINKLEY, D. W., HORWITZ, S., AND REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4*, 1 (1995), 3–35.

[7] CANFORA, G., CIMITILE, A., DE LUCIA, A., AND LUCCA, G. A. D. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)* (Victoria, Canada, Sept. 1994), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 424–433.

[8] CANFORA, G., CIMITILE, A., AND MUNRO, M. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance : Research and Practice 6*, 2 (1994), 53–72.

[9] CIMITILE, A., DE LUCIA, A., AND MUNRO, M. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)* (Nice, France, 1995), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 124–133.

[10] CIMITILE, A., DE LUCIA, A., AND MUNRO, M. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice 8* (1996), 145–178.

[11] DANICIC, S. *Dataflow Minimal Slicing*. PhD thesis, University of North London, UK, School of Informatics, Apr. 1999.

[12] DANICIC, S., HARMAN, M., AND SIVAGURUNATHAN, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[13] DE LUCIA, A. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[14] DE LUCIA, A., FASOLINO, A. R., AND MUNRO, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[15] GALLAGHER, K. B. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance* (Nov. 1992), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 236–244.

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

[17] GRAMMATECH INC. www.grammatech.com /products/codesurfer/codesurfer.html, 1999.

[18] GUPTA, R., HARROLD, M. J., AND SOFFA, M. L. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance* (Orlando, Florida, USA, 1992), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 299–308.

[19] HARMAN, M., AND DANICIC, S. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5*, 3 (Sept. 1995), 143–162.

[20] HARMAN, M., AND HIERONS, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.

[21] HARMAN, M., HIERONS, R. M., AND DANICIC, S. The relationship between program dependence and mutation analysis. In *Mutation Testing for the New Century (proceedings of Mutation 2000)* (San Jose, California, USA, Oct. 2001), W. E. Wong, Ed., Kluwer, pp. 5–13.

[22] HARMAN, M., HIERONS, R. M., DANICIC, S., HOWROYD, J., AND FOX, C. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)* (Florence, Italy, Nov. 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 138–147.

[23] HAUSLER, P. A. Denotational program slicing. In 22^{nd}, *Annual Hawaii International Conference on System Sciences, Volume II* (Jan. 1989), pp. 486–495.

[24] HECHT, M. S. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[25] HIERONS, R. M., HARMAN, M., AND DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability 9*, 4 (1999), 233–262.

[26] HIERONS, R. M., HARMAN, M., FOX, C., OUARBYA, L., AND DAOUDI, M. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability* (Mar. 2002). to appear.

[27] HORWITZ, S., PRINS, J., AND REPS, T. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345–387.

[28] HORWITZ, S., REPS, T., AND BINKLEY, D. W. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 25–46. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[29] KAMKAR, M. *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.

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

[31] LYLE, J. R., AND WEISER, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 877–882.

[32] M.WARD. Assembler to c migration using the fermat transformation system. *Internation Conference on Software Maintainance* (August 1999).

[33] M.WARD, H. . A multiple backtracking algorithm. *Journal of Symbolic Computation*, 1 (1994), 1–40.

[34] OTT, L. M., AND THUSS, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (Baltimore, Maryland, USA, May 1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 71–81.

[35] S.HORWITZ, T. R., AND D.BINKLEY. Interprocedural slicing using dependance graphs. *ACM Trans. Program. Lang. Syst 12*, 1 (Jan. 1990), 26–60.

[36] SIMPSON, D., VALENTINE, S. H., MITCHELL, R., LIU, L., AND ELLIS, R. Recoup – Maintaining Fortran. *ACM Fortran forum 12*, 3 (Sept. 1993), 26–32.

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

[38] WARD, M. Assembler to c migration using the fermat transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)* (Oxford, UK, Aug. 1999), IEEE Computer Society Press, Los Alamitos, California, USA.

[39] WARD, M., CALLISS, F. W., AND MUNRO, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

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

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

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