A Parallel Algorithm for Static Program Slicing

Sebastian Danicic, Mark Harman and Yoga Sivagurunathan
Project *Project*,
School of Computing,
University of North London,
Eden Grove, London, N7 8DB.
tel: +44 (0)171 753 5751
fax: +44 (0)171 753 7009
e-mail: s.danicic@uk.ac.unl

Keywords: Slicing, Parallel Algorithms, Program Simplification

Abstract

Program Slicing is the process of deleting statements in a program that do not affect a given set of variables at a chosen point in the program.

In this paper the first parallel slicing algorithm for static program slicing is introduced. It is shown how the control flow graph of the program to be sliced is converted into a network of concurrent processes, thereby producing a parallel version of Weiser's original static slicing algorithm.

The parallel slicing algorithm is formally defined as the least solution (containing the slicing criterion) of a set of recursion equations, derived from the program's control flow graph and is proved correct with respect to Weiser's algorithm.

The parallel slicing algorithm has a number of other interesting and potentially useful properties. For example, in addition to constructing a slice, it produces a set of equivalent slicing criteria for the slice it constructs.

1 Introduction

A slice of a program p is constructed with respect to a slicing criterion, (V, n), by deleting statements from p which have no effect on any of the variables[1] in V as execution reaches line n. A slice is thus (typically) simpler than the original program.

Weiser's original algorithm [27, 29] has been modified by several authors, who have improved upon the accuracy and speed of slice construction [22, 15, 18, 23, 14].

The applications of slicing include program comprehension [28], program maintenance and debugging [6, 19], program testing [8], the calculation of cohesion metrics [21, 4, 9] and program integration [12]. These applications rely on the way in which slicing allows for program simplification, by extracting a thread of the original program's computation.

[1]Throughout this paper, the set V in the slicing criterion is refered to as the 'slice set'.

Weiser's original approach solved the static slicing problem. Other authors have extended slicing to the dynamic [2, 7, 16, 17] and quasi–static paradigms [26, 24].

The algorithm presented in this paper also solves the problem of static program slicing. It is essentially a parallel version of Weiser's and is defined in terms of the control flow graph(CFG)[10]. The CFG is compiled into a network of concurrent processes similar to those defined in [1]. Each process communicates with the others along the arcs of the CFG. The messages include sets of variables, which as will be shown, are closely related to Weiser's 'relevant variables' defined in section 2.1.

To our knowledge, the algorithm introduced here is the first to exploit concurrent execution in program slice construction [22, 29].

The rest of this paper is organised as follows:-
In section 2 background material is presented. Section 3 describes the parallel slicing algorithm and section 4 contains a detailed example, showing how it constructs a slice. In section 5 the parallel slicing algorithm is formally defined in terms of recursion equations and in section 6 it is proved correct with respect to Weiser's algorithm. Section 7 describes some other interesting and potentially useful properties of the parallel slicing algorithm over and above slice construction and section 8 concludes with some directions for future work.

2 Background

2.1 Weiser's Algorithm

In Weiser's original thesis[27] and Tip's later exposition[25] it is shown how slices can be computed by solving a set of data and control flow equations derived directly from the CFG of the program being sliced. These equations are solved using an iterative process which entails computing sets of 'relevant variables' for each node in the CFG.

2.1.1 Directly Relevant Variables

Suppose a slice is to be constructed for the slicing criterion (V, n). First, the *directly relevant variables* of node i, $R^0_C(i)$, are defined inductively as follows:-

1. The set of directly relevant variables at the slice node, n, is simply the slice set, V.

2. The set of directly relevant variables at every other node i, is defined in terms of the set of directly relevant variables of all nodes j leading directly from i to j (written $i \rightarrow_{CFG} j$) in the CFG. $R^0_C(i)$ contains all the variables v such that either

   (a) $v \in R^0_C(j)$ and $v \notin$ DEF(i) or

   (b) $v \in$ REF(i) and DEF$(i) \cap R^0_C(j) \neq \emptyset$.

2.1.2 Referenced and Defined Variables

For each node i, the sets REF(i) and DEF(i) are derived directly from the statement from which i was obtained [3, 29]. For example, if node i was obtained from the assignment statement `x=y+z;` it would have REF(i)=\{`y,z`\} and DEF(i)=\{`x`\}.

2.1.3 Directly Relevant Statements

In terms of the directly relevant variables, a set of *directly relevant statements* S_C^0 is defined:-

$$S_C^0 = \{i \mid \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } \mathrm{DEF}(i) \cap R_C^0(j) \neq \emptyset\}$$

This completes the first iteration of Weiser's algorithm.

2.1.4 Controlled Nodes

The execution of a predicate node 'controls' [5] the execution of other nodes in the CFG by determining whether or not control will definitely pass to these nodes or not. For each predicate, p, the set of nodes that depend on the outcome of p in this way are termed *the controlled nodes* of p.

In a block structured language, controlled nodes may be calculated from the block structure of the language – the controlled nodes of a conditional statement's predicate node are those nodes which form the body of the consequent and alternative branch of the conditional. For a `while` loop, the controlled nodes of the predicate node are simply those in the body of the loop. For unstructured languages, the calculation of controlled nodes can be achieved using the algorithm of Ferrante, Ottenstein and Warren [5].

2.1.5 Indirectly Relevant Variables

The subsequent iterations of Weiser's algorithm calculate the *indirectly relevant variables*, R_C^K where $K \geq 1$:-

$$R_C^{K+1}(i) = R_C^K(i) \cup \bigcup_{b \in B_C^K} R^0_{(b,\mathrm{REF}(b))}(i)$$

where

$$\mathrm{B}_C^K = \{b \mid \exists i \in S_C^K \text{ such that } b \text{ controls } i\}$$

B_C^K is the set of all predicate nodes that *control* a statement in S_C^K.

2.1.6 Indirectly Relevant Statements

Adding the predicate nodes to S_C^K includes further indirectly relevant statements in the slice:-

$$S_C^{K+1} = B_C^K \cup \{i \mid \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } \mathrm{DEF}(i) \cap R_C^{K+1}(j) \neq \emptyset\}$$

As Tip[25] states, this process will eventually terminate since S_C^K and R_C^K are non–decreasing subsets of the program's variables.

2.2 Process Networks as Recursion Equations

In [1], it is shown that an attractive model for a network of concurrent processes is a graph whose nodes represent processes and whose arcs represent communication channels.

The behaviour of such networks is defined in terms of a set of recursion equations, whose solutions represent the (possibly infinite) streams of data communicated between the processes. Each process is, in effect, defined as a recursive function on streams.

It is shown how properties of networks defined in this way can be proved using inductive techniques.

3 The Parallel Slicing Algorithm

The slice of the original program is created as follows:-

1. The original program is converted into a CFG.
2. The CFG is compiled into a network of concurrent processes.
3. The network of concurrent processes is executed. The nodes of the CFG that correspond to statements to be kept in the final slice are obtained from the resulting output.

3.1 Constructing a Process Network

From the program to be sliced, a process network is constructed. The topology of the network is obtained directly from the inverse of the program CFG, called the Reverse Control Flow Graph(RCFG) [3], with one process for each node in the CFG and with communication channels corresponding to the arcs of the RCFG.

As is the convention, arcs entering a node, i, represent inputs to process, i, and arcs leaving i represent outputs from process, i. When a process outputs a message, it shall mean that the message is output on all output channels.

3.2 Process Behaviour

Each process sends and receives messages that are sets of variable names and node identifiers. The behaviour of each process, i, depends precisely on the following information, derived directly from the CFG of the program being sliced:-

i	The identifier of the corresponding node of the CFG
REF(i)	The set of variables referenced by i
DEF(i)	The set of variables defined by i
$C(i)$	The set of nodes controlled by i

Processes with more than one input in the RCFG correspond to predicate nodes (section 2.1.4). The present paper is concerned with side-effect free languages, so all such processes will have DEF(i) $= \emptyset$. Conversely, processes with only one input do not correspond to predicate nodes, and therefore, by definition, they control no other nodes and so have $C(i) = \emptyset$.

The behaviour of each process, i, can be defined in CSP style notation [11] as follows:-

$$
\begin{array}{l}
P(i) = \\
\qquad ?\mathrm{S} \rightarrow (\text{if } \mathrm{S} \cap (\mathrm{DEF}(i) \cup \mathrm{C}(i)) \neq \emptyset \\
\qquad\qquad \text{then } !((\mathrm{S} \backslash \mathrm{DEF}(i)) \cup \mathrm{REF}(i) \cup \{i\}) \\
\qquad\qquad \text{else } !\mathrm{S}); \\
\qquad\qquad P(i)
\end{array}
$$

If the input, S, to process i, has any elements in common with the defined variables of i or with the controlled nodes of i then the process, i, outputs the set consisting of :-

1. all its input variables (elements of S) that it does not define,

2. all variables that it references,

3. its node identifier, i.

On the other hand, if S has no elements in common with the defined variables or controlled nodes of i then the process i merely outputs S. The process i then repeats this action, waiting for the next input message.

3.3 Initiating Network Communication

In order to construct a slice for the criterion (V, n), network communication is initiated by outputting[2] the message V from process n.

3.4 Constructing the Slice

For any process i, the node i should be included in the final slice if and only if i has output its node identifier i. The slice of a program computed by this algorithm can be found by including the set of nodes whose identifiers are input to the entry node[3] of the CFG, because the entry node is reachable via every node in the RCFG and thus messages output by all nodes will eventually reach the entry process.

Once the set of nodes to be included in the final slice is obtained using the parallel slicing algorithm, the problem of converting this information back into the syntax of the original program arises. The solution is well known [27, 13] and is therefore not discussed further in this paper.

Definition 3.2 above, should be thought of as a 'specification' of process behaviour rather than an 'implementation'. The important aspect of the definition is that each process should be thought of as a function from the union of all its inputs to the union of all its outputs.

In a valid implementation of the parallel slicing algorithm, to ensure termination, processes should, in fact, not output the same message more than once (see section 6.2) as is the case in the following example execution of an implementation of definition 3.2.

---

[2]According to Weiser's original definition of a static slice [27], a slice constructed for the slicing criterion (V, n), need not contain the node n. Korel and Laski [17] argue that the omission of the slice node is unfortunate; a programmer finds the absence of the slice node confusing when attempting to relate the slice to the original. In order to include the slice node in a slice constructed by the parallel slicing algorithm, process communication should be initiated by outputting, not just the set of variables V, but also the node identifier, n.

[3]This observation is due to Balasubramaniam Sivagurunathan, whom the authors gratefully acknowledge.

4 A Worked Example

Let the slicing criterion be ({c},7) and the program to be sliced be the one shown in figure 1.

```
1    a = 0;
2    while (s < t) {
3      if (t==4)
4        c = t;
5      s = 2;
6      c = t + 7;
7      t = a + 4;
     }
```

Figure 1: The program to be sliced

The RCFG of the program to be sliced is shown in figure 2:-

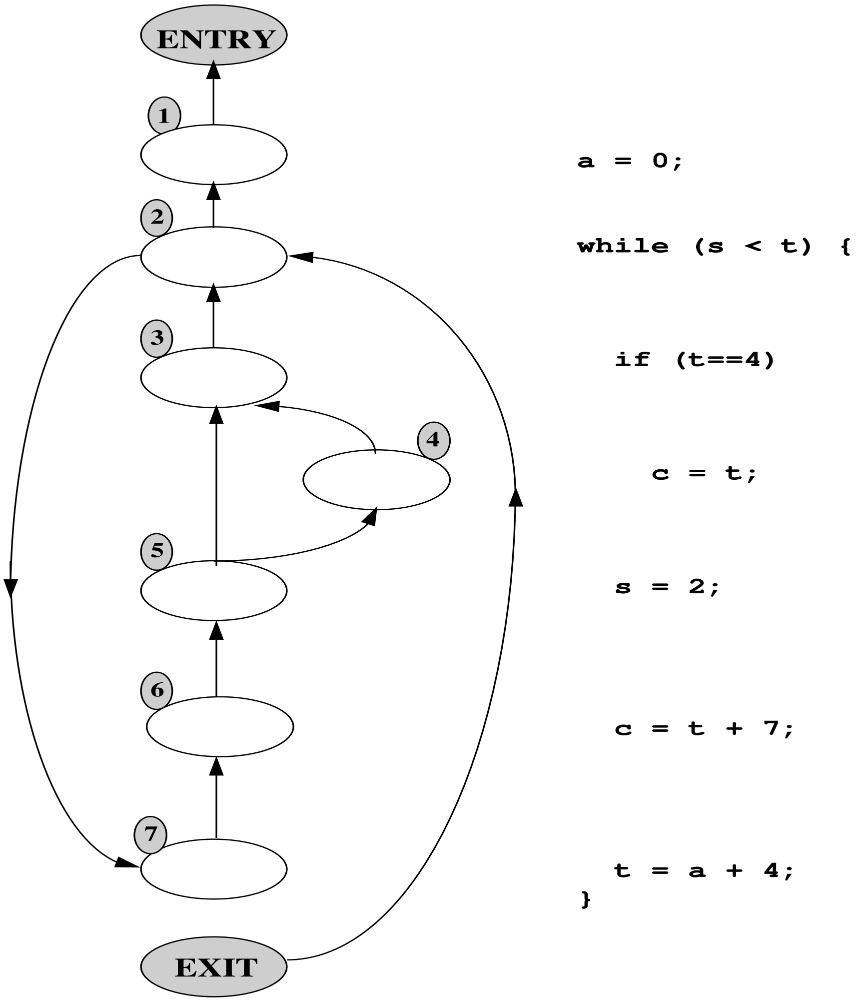

Figure 2: The RCFG obtained from the initial program

The process network obtained from the RCFG is shown in figure 3. A slice is to be constructed for the criterion ({c},7), so process communication is initiated by outputting {c} from node 7.

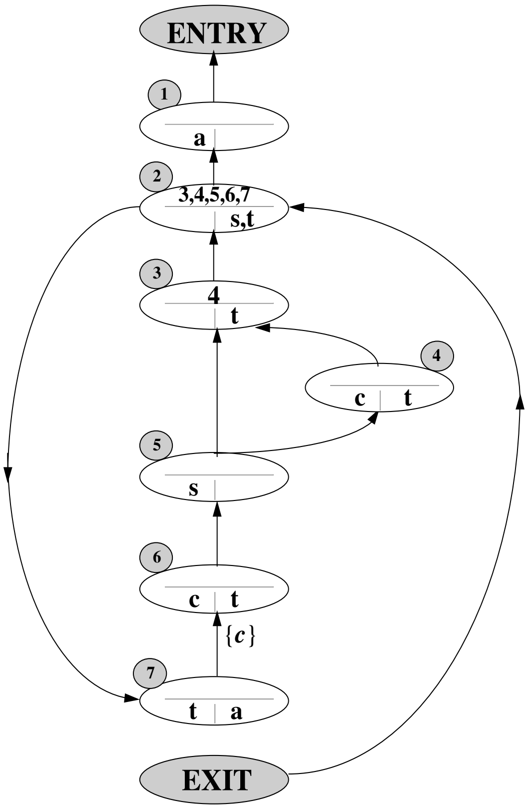

Figure 3: Initial state of the process network

To show the progression of the state of the system, we label the arcs in the RCFG with the messages communicated by the relevant processes during execution. New messages communicated at each stage are labelled in bold typeface.

Processes are drawn like this:

Where C is the set of controlled nodes, DEF is the set of defined variables and REF is the set of referenced variables of each process.

After receiving {`c`} through its input channel, process six outputs {`t,6`}. Processes five, four and three all eventually receive {`t,6`} which they simply output because {`t,6`} is disjoint from the defined variables of these processes.

The resulting state is shown in figure 4.

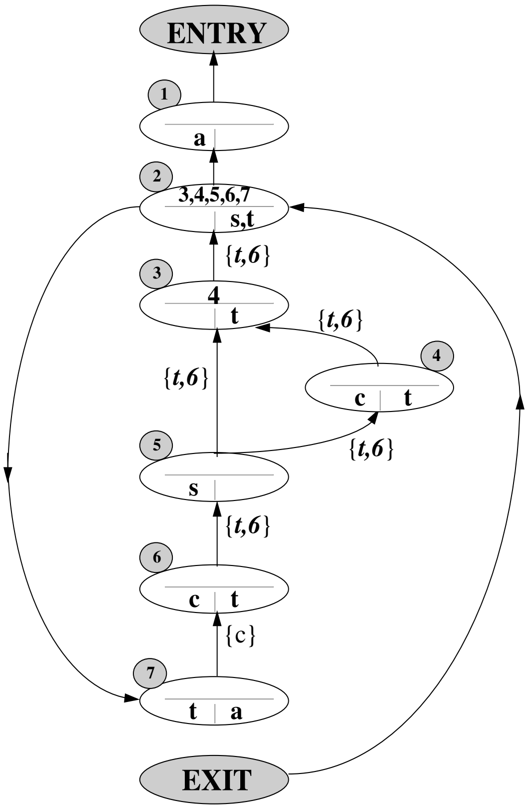

Figure 4: The state just before process two outputs its first message

When {`t,6`} is input to process two, it causes process two to output {`s,t,2,6`} to processes one and seven. This is an instance of a process responding to an input containing the identifier of a node that it controls. Process two will therefore output a message including its own node identifier, representing the fact that node two will also be included in the final slice.

The resulting state is shown in figure 5.

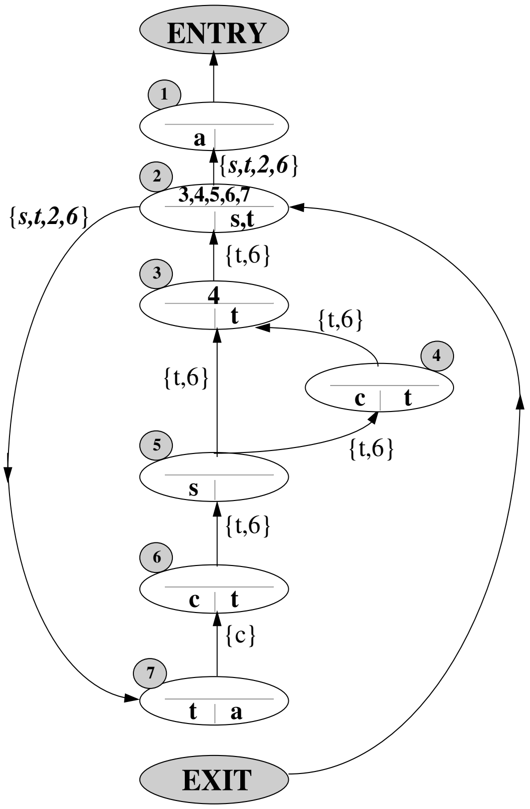

Figure 5: The state just after process two has output its first message

The message {`s,t,2,6`} passes through process one to the `ENTRY` node. On receiving {`s,t,2,6`}, process seven outputs the message {`s,a,2,6,7`} because it defines `t`. This output message passes unaffected through process six.

When process five receives {`s,a,2,7`} it outputs {`a,2,5,7`}.

The resulting state is shown in figure 6.

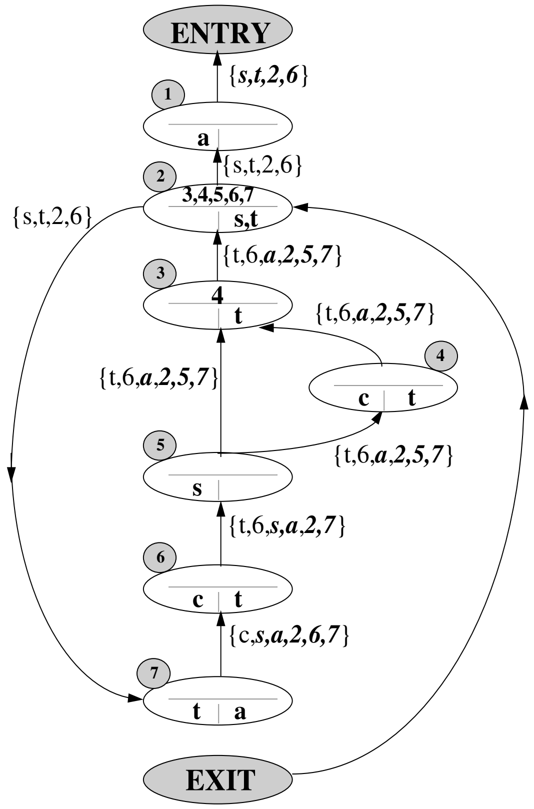

Figure 6: The state after one more pass around the loop

Continuing process communication passes the extra messages in the network to all reachable nodes, but causes no new messages to be introduced into the system. Finally the network terminates in the state shown in figure 7.

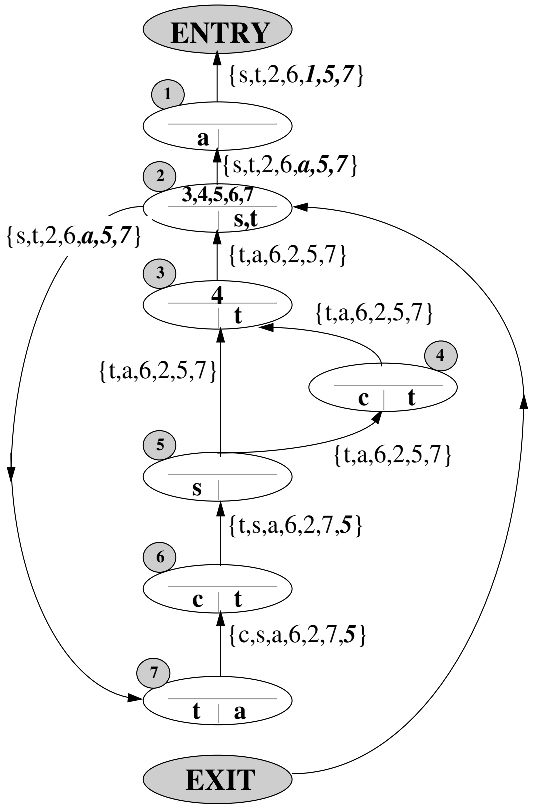

Figure 7: The final state of the process network

From the final state of the network the slice of the original program is constructed by including those statements and predicates whose node identifiers have reached the `ENTRY` node.

1	`a = 0;`	1	`a = 0;`
2	`while (s < t) {`	2	`while (s < t) {`
3	`  if (t==4)`	3	
4	`    c = t;`	4	
5	`  s = 2;`	5	`  s = 2;`
6	`  c = t + 7;`	6	`  c = t + 7;`
7	`  t = a + 4;`	7	`  t = a + 4;`
	`}`		`}`
The Original Program		The Slice on ({`c`},7)	

Figure 8: The original program and its slice

5 Functional Networks

The processes networks used in the parallel slicing algorithm can be defined in terms of recursion equations, not over infinite streams as in [1], but over finite sets of variable names and node identifiers.

5.1 Example

Consider the process network described in section 4 with each arc and node labelled as follows:-

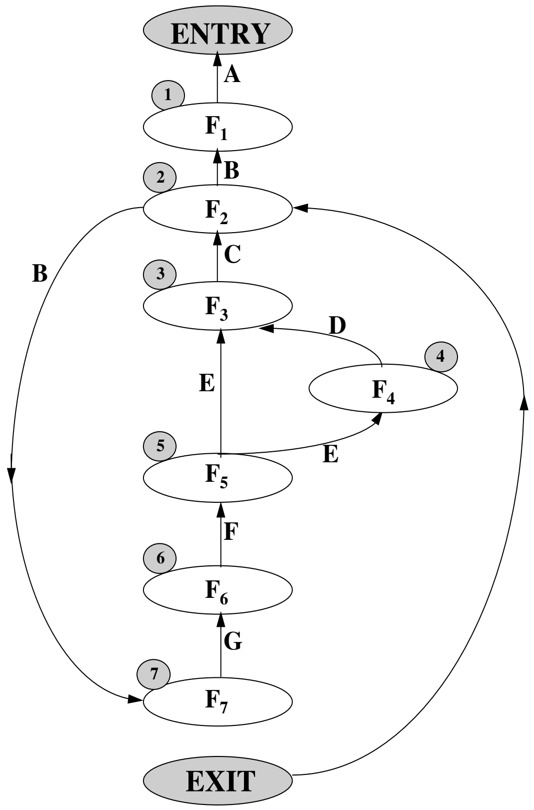

Figure 9: The functional network derived from the example program

From the diagram, the following recursion equation is derived:-

$$G = F_7(F_2(F_3(F_5(F_6(G)) \cup F_4(F_5(F_6(G))))))$$

Where each F_i corresponds to the behaviour of the process i as a function on sets as described in section 3. i.e.

$F_i(S)=$

if S $\cap$ (DEF(i) $\cup$ C(i)) $\neq \emptyset$
then (S\DEF(i)) $\cup$ REF(i) $\cup$ $\{i\}$
else S

Inputs to processes are represented by arguments to the corresponding functions, F_i, and outputs of processes by the results of the corresponding functions. Clearly, different network topologies are achieved by composing the functions in different ways[4]. Notice that if a process has more than one input, then the argument of the corresponding function is the union of the individual inputs. In the above example there is one loop and hence only one equation. In general however, there will be an equation for each cycle in the RCFG.

5.2 Solving the Equations to Produce Slices

The equations, in isolation, represent the static properties of network. A solution to the equations represents a possible labelling with a set of variable names and node identifiers of all the arcs of the RCFG. For each arc, this label corresponds to the union of all messages that, in a valid implementation (see section 5.3), will be transmitted along the communication channel represented by that arc.

5.3 Valid Implementations

In general, of course, there are many solutions to such equations. Following [20], a *valid implementation* of the parallel slicing algorithm is defined as one which produces the *least* solution to the equations; that is, the least solution relative to the partial order of arc–wise set inclusion, i.e., using $\sqsubseteq$ to denote the partial order, for networks L_1 and L_2, $L_1 \sqsubseteq L_2$ if and only if, for every arc, the corresponding label for L_1 is a subset of the corresponding label for L_2.

In order to produce the slice, the solution sought must be the least solution which contains the slice set as a subset of the label at the slice node (because the parallel slicing algorithm is initiated by the slice node outputting the slice set).

In section 5.1 above, since the slice set is $\{\mathtt{c}\}$, the least solution to the recursion equation that has $\{\mathtt{c}\} \subseteq G$ is required.

6 Correctness of the Parallel Slicing Algorithm

In this section the parallel slicing algorithm is proved correct, in the sense that every statement included in a Weiser slice will also be included using the parallel slicing algorithm. Some preliminary results are first stated and proved.

6.1 Existence of Solutions

It is necessary to verify that solutions to such equations exist. This follows Kleene's first recursion theorem [20], from the fact that the functions F_i, introduced in section 5.1 are monotonic

[4]The possibility of describing the network in this way suggests an implementation of the algorithm by compiling the RCFG into a functional program.

with respect to set inclusion and hence can be solved by constructing Kleene chains.

6.2 Termination

It is also important to show that such recursion equation systems give rise to terminating computations. To do this, it must be shown that *finite* solutions always exist. This follows from the fact that the labelling of each arc must be contained in the finite set consisting of all node identifiers and variable names of the CFG.

It can easily be shown that the functions representing processes possess the *additive* property i.e.

$$F(A \cup B) = F(A) \cup F(B)$$

This ensures that, in a valid implementation of the parallel slicing algorithm, each process never need output the same value more than once, ensuring that it need not output messages indefinitely.

6.3 Correctness Proof

Definition 6.1 (outputs$_{(V,i)}(n)$)
Let (V, i) be a slicing criterion for a CFG and let n be a node of the CFG,

Outputs$_{(V,i)}(n)$ is defined to be the union of all messages output by node n when slicing using the parallel slicing algorithm with respect to (V, i).
More rigorously, outputs$_{(V,i)}(n)$ is the labelling of all arcs emerging from n in the least solution of the equations derived from the CFG where the labelling of all arcs emerging from i contain V.

6.3.1 Lemma 1

Let (V, i) be a slicing criterion for a CFG then $V \subseteq$ outputs$_{(V,i)}(i)$

Proof obvious.

6.3.2 Lemma 2

Let (V, i) be a slicing criterion for a CFG
If $K \subseteq$ outputs$_{(V,i)}(b)$ then for all nodes, j, of the CFG, outputs$_{(K,b)}(j) \subseteq$ outputs$_{(V,i)}(j)$

Proof

Since $K \subseteq$ outputs$_{(V,i)}(b)$, the least solution containing V on all arcs emerging from node i contains K on all arcs emerging from node b.
For all nodes j, let outputs$_{(K,b)}(j) = X_j$
So by definition, the least solution containing K on all arcs emerging from b has X_j on all arcs emerging from j, for all nodes j.
So the least solution containing V on all arcs emerging from i has X_j on all arcs emerging from

j, for all nodes j.
So for all nodes j of the CFG, $\text{outputs}_{(K,b)}(j) \subseteq \text{outputs}_{(V,i)}(j)$ as required.

The main theorem will now be proved. It is proved that every statement included in a Weiser slice will also be included in the slice obtained using the parallel slicing algorithm.

It shall be proved by induction that for all $n \geq 0$, for all slicing criteria C, for all nodes i that:-

1. $\mathrm{R}_C^n(i) \subseteq \text{outputs}_C(i)$
   i.e. the label of the arc(s) from node i include the relevant variable for node i.
2. $j \in S_j^n \Rightarrow j \in \text{outputs}_C(j)$
   i.e. if j is a relevant statement then it will output its node identifier.

6.4 Base Case

First it is proved that:-

3. $\forall i, c \quad \mathrm{R}_C^0(i) \subseteq \text{outputs}_C(i)$
4. $\forall j, c \quad j \in S_C^0 \Rightarrow j \in \text{outputs}_C(j)$

6.4.1 Proof of 3

Part 3 is proved by induction on the *maximum distance* of i from the slice node. The maximum distance from node i to node j is the maximum number of distinct nodes in a path from i to j.

First, if i is the slice node then by definition $\mathrm{R}_C^0(i) = V$ (V is the slice set) and by definition of the parallel slicing algorithm (sections 3.3 and 5.3), $V \subseteq \text{outputs}_C(i)$.

If i is not the slice node, suppose for all arcs at a maximum distance $\leq N$ from the slice node that $\mathrm{R}_C^0(i) \subseteq \text{outputs}_C(i)$.

Let i be a node at a maximum distance $N+1$ from the slice node, then by definition, all nodes which input to i are at a distance $\leq N$ from i and inductively, it can be concluded that:-

5. The inputs to i contains $\bigcup_{j \rightarrow_{RCFG} i} \mathrm{R}_C^0(j)$

Let $v \in \mathrm{R}_C^0(i)$, then, by definition,

either there exists a j such that $j \rightarrow_{RCFG} i$ and $v \notin \mathrm{DEF}(i)$ and $v \in \mathrm{R}^0_c(j)$
in which case v is input to node i (by 5 above) and will be output by i (by definition of process behaviour (sections 5.1 and 3.2)).

or $v \in \mathrm{REF}(i)$ and there exists a j such that $\mathrm{DEF}(i) \cap \mathrm{R}^0_C(j) \neq \emptyset$
in which case by 5 above and again by definition of process behaviour, it follows that i outputs $\mathrm{REF}(i)$ and hence i outputs v.

Concluding that $\mathrm{R}^0_C(i) \subseteq \mathrm{outputs}_C(i)$

6.4.2 Proof of 4

$$\text{By definition } S^0_C = \{i \mid \mathrm{DEF}(i) \cap \mathrm{R}^0_C(j) \neq \emptyset\}$$

$$\begin{aligned}
\text{hence } j \in S^0_C \;&\Rightarrow\; \exists k \text{ such that } k \rightarrow_{RCFG} j \text{ and } \mathrm{DEF}(j) \cap \mathrm{R}^0_C(k) \neq \emptyset \\
&\Rightarrow\; \exists k \text{ such that } k \rightarrow_{RCFG} j \text{ and } \mathrm{DEF}(j) \cap \mathrm{outputs}_C(k) \neq \emptyset \\
&\Rightarrow\; j \in \mathrm{outputs}_C(j) \;\; \text{by definition of process behaviour (5.1, 3.2)}
\end{aligned}$$

This concludes the proof of the base case.

6.5 Inductive Step

Now assume

6. $\forall i, C \;\; \mathrm{R}^N_C(i) \subseteq \mathrm{outputs}_C(i)$

and

7. $\forall j, C \;\; j \in S^N_C \Rightarrow j \in \mathrm{outputs}_C(j)$

It must be proved that:-

8. $\forall i, C \;\; \mathrm{R}^{N+1}_C(i) \subseteq \mathrm{outputs}_C(i)$ and
9. $\forall j, C \;\; j \in S^{N+1}_C \Rightarrow j \in \mathrm{outputs}_C(j)$

6.5.1 Proof of 8

Weiser defines

$$\mathrm{R}_C^{K+1}(i) = \mathrm{R}_C^K(i) \cup \bigcup_{b \in \mathrm{B}_C^K} \mathrm{R}^0_{(b,\mathrm{REF}(b))}(i)$$

and

$$S_C^{K+1} = \mathrm{B}_C^K \cup \{i \mid \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } \mathrm{DEF}(i) \cap \mathrm{R}_C^{K+1}(j) \neq \emptyset\}$$

and

$$\mathrm{B}_C^K = \{b \mid \exists i \in S_C^K \text{ such that } b \text{ controls } i\}$$

Now let $v \in \mathrm{R}_C^{N+1}(i)$

$$\text{Assume } \exists b \in \mathrm{B}_C^N \text{ such that } v \in \mathrm{R}^0_{(b,\mathrm{REF}(b))}(i)$$

Then by (3) it follows that

$$v \in \text{outputs}_{(b,\mathrm{REF}(b))}(i)$$

But, by definition, b controls a node in S_C^K. So by the induction hypothesis (7) this node will have output its node identifier. b will therefore receive this node identifier, and by definition of process behaviour (sections 5.1 and 3.2), b will output $\mathrm{REF}(b)$.

$$\text{i.e. } \mathrm{REF}(b) \subseteq \text{outputs}_{(v,i)}(b)$$

$$\text{so outputs}_{(b,\mathrm{REF}(b))}(i) \subseteq \text{outputs}_C(i) \text{ by Lemma 2 (section 6.3.2)}$$

$$\text{so } v \in \text{outputs}_C(i)$$

as required for proof of (8).

6.5.2 Proof of 9

Let $i \in S_C^{N+1}$,

then $i \in \mathrm{B}_C^N$ or $\mathrm{DEF}(i) \cap \mathrm{R}_C^{N+1}(j) \neq \emptyset$ for some node j inputting to i.

if $\quad \mathrm{DEF}(i) \cap \mathrm{R}_C^{N+1}(j) \neq \emptyset$
then $\mathrm{DEF}(i) \cap \mathrm{outputs}_C(j) \neq \emptyset$ for some node j inputting to i which, by definition of process behaviour (5.1 and 3.2), implies $i \in \mathrm{outputs}_C(i)$
or if $i \in \mathrm{B}_C^N$
then i controls an element j say of S_C^N
but by induction hypothesis (7), $j \in \mathrm{outputs}_C(i)$

This concludes the proof.

7 Discussion

The parallel slicing algorithm may be adapted to construct slices for multiple slicing criteria. The algorithm also produces some extra information about the slices it constructs, as a by–product.

7.1 Simultaneous Slicing

For the slicing criterion (V, n), process communication is initiated by outputting V as a message on all output channels from process n. In some situations it may be desirable to construct a slice which preserves the effect of the original program simultaneously on *several* slicing criteria. In such a situation, the slice would be constructed with respect to a set of m slicing criteria, $\{(V_1, n_1), \ldots, (V_m, n_m)\}$. In order to use parallel slicing algorithm to construct a slice with respect to $\{(V_1, n_1), \ldots, (V_m, n_m)\}$, process communication is initiated simply by outputting V_1 at n_1, V_2 at n_2 and so on, up to, V_m at n_m. This can also be achieved using the PDG approach [14], but has not previously been considered using Weiser's approach [27, 29].

7.2 Equivalence Classes of Slicing Criteria

The final state of the network also has an interesting property concerning the sets of variable names labelling each arc of the stable process network.

Consider slicing on the criterion (V, n). For each arc, a, in the network, consider the restriction of each labelling to just variable names (i.e. ignore the node identifiers). Now these sets represent the variables whose values must be preserved at arc a, in order to preserve the values of V at n (lemma 6.3.2). The slice constructed by the parallel slicing algorithm, for the slicing criterion (V, n), is thus also a slice for the set of criteria of the form (K, m), where K is the set of variable names contained in $\mathrm{outputs}_{(V,n)}(m)$. Thus constructing a single slice at one point in the program, using the parallel slicing algorithm, automatically produces an entire *set* of slicing criteria for which the program produced is *also* a slice.

For any program therefore, an equivalence relation $\sim$ on slicing criteria can be defined, such that $(V, n) \sim (K, m)$ if and only if they both lead to the same slice. The parallel slicing algorithm has the property that the set of labellings of stable process networks are all in the same equivalence class with respect to $\sim$.

8 Conclusion and Future Work

In this paper a new algorithm for static slicing has been introduced, formally specified and proved correct. The algorithm exploits the natural parallelism in the control flow graph and is essentially a parallel version of Weiser's algorithm based on the view of a process network as a set of recursion equations. It has a number of pleasing and pontentially useful properties which, allowing for simultaneous slicing (section 7.1) at no extra computational cost, and providing a set of equivalent slicing criteria (section 7.2).

The slicing problem considered in this paper is the int*ra*procedural one for language without pointers. The authors are working on a solution to the problems of int*er*procedural slicing in the presence of pointers.

The authors also believe that the view of a CFG as a process network with its correspondence to a set of recursion equations is extremely attractive and may form the basis for parallel solutions to other graph theoretic problems in source code analysis.

References

[1] Abramsky, S. Reasoning about concurrent systems. In *Distributed Computing* (London, 1984), pp. 307–319.

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

[3] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[4] Bieman, J. M., and Ott, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[5] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems 9* (July 1987), 319–349.

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

[7] Gopal, R. Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance* (1991), pp. 191–200.

[8] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability* (1995). To appear.

[9] Harman, M., Danicic, S., Sivagurunathan, B., Jones, B., and Sivagurunathan, Y. Cohesion metrics. In 8^{th} *International Quality Week* (San Francisco, May 29th – June 2nd. 1995), pp. Paper 3–T–2, pp 1–14.

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

[11] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[12] 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.

[13] Horwitz, S., and Reps, T. The use of program dependence graphs in software engineering. In *14th. International Conference on Software Engineering* (Melbourne, Australia, 1992), pp. 392–411.

[14] Horwitz, S., Reps, T., and Binkley, D. 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.

[15] Jiang, J., Zhou, X., and Robson, D. J. Program slicing for C – The problems in implementation. In *IEEE Conference on Software Maintenance* (Italy, 1991), pp. 182–190.

[16] Kamkar, M., Shahmehri, N., and Fritzson, P. Interprocedural dynamic slicing. In *Proceedings of the 4th. Conference on Programming Language Implementation and Logic Programming* (1992), pp. 380–384.

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

[18] Leung, H. K. N., and Reghbati, H. K. Comments on program slicing. *IEEE Transactions on Software Engineering 13*, 12 (1987), 1370–1371.

[19] Lyle, J. R., and Weiser, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), pp. 877–882.

[20] Manna, Z. *Mathematical Theory of Computation.* McGraw–Hill, 1974.

[21] Ott, L. M., and Thuss, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (1993), pp. 78–81.

[22] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in software development environments. *SIGPLAN Notices 19*, 5 (1984), 177–184.

[23] Qi, L., Fubo, Z., and Jiahua, Q. Program slicing. *Journal of Computer Science and Technology 3*, 1 (1988), 29–39.

[24] Tip, F. *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[25] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages* (1995). To appear.

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

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

[28] Weiser, M. Programmers use slicing when debugging. *Communications of the ACM 25*, 7 (July 1982), 446–452.

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