A New Algorithm for Slicing Unstructured Programs

MARK HARMAN[1]* and SEBASTIAN DANICIC[2]

[1] *Department of Mathematical and Computing Sciences, Goldsmiths College, University of London, Lewisham Way, New Cross, London SE14 6NW, UK.*

[2] *School of Informatics, University of North London, Eden Grove, London, N7 8DB.*

SUMMARY

Program slicing is an automatic program abstraction technique whose many applications include software maintenance, re–engineering and comprehension, all of which rely crucially upon the precision of the slicing algorithm used. When slicing is applied to maintenance problems, the programs to be sliced are typically legacy systems, often written in older, 'unstructured' programming styles. For slicing to be a useful tool to the software maintainer it is therefore important to have precise algorithms for slicing unstructured programs.

Unfortunately the standard algorithms for slicing structured programs do not extend correctly to the unstructured paradigm, and currently proposed modifications to these standard algorithms produce either unnecessarily large slices or slices which are not true subsets of the original program from which they are constructed.

This paper introduces a modification of Agrawal's algorithm for slicing unstructured programs, which overcomes these difficulties. The new algorithm produces thinner slices than any previously published algorithm while respecting both the semantic and syntactic constraints of slicing.

No. of Figures: 13. No of Tables 0. No of References: 40.

KEY WORDS: Slicing; Unstructured Programs; Legacy Systems.

*Address for Correspondence: Dr. Mark Harman, Department of Mathematical and Computing Sciences, Goldsmiths College, University of London, New Cross, London SE14 6NW, United Kingdom. Voice telephone: 44–171–919–7860; fax: 44–171–919–7853. Email: M.Harman@gold.ac.uk

Mr. Sebastian Danicic, School of Informatics and Multimedia Technology, University of North London, Eden Grove, London, N7 8DB, United Kingdom. Voice telephone: 44–171–753–5751; fax: 44–171–753–7009. Email: S.Danicic@unl.ac.uk

1 INTRODUCTION

Slicing is a technique for locating the statements of a program which may affect the value of a chosen set of variables at some point in a program. Weiser introduced slicing in his seminal thesis (Weiser, 1984). According to Weiser, a slice s, of a program p, is constructed with respect to a slicing criterion (V, n), where V is a set of variable identifiers and n is a program point. Statements in p which cannot affect the values of V when the next statement to be executed is at point n, may be removed from p to form s. Thus a slice preserves the effect of the original program with respect to the slicing criterion, while simplifying the program. Figure 1 depicts an example program and one of its slices.

Weiser's original definition of a slice has come to be known as the 'static backward' slice, distinguishing it from 'dynamic', 'forward' and 'conditioned' slices.

The dynamic slicing criterion contains the input sequence supplied to the program, I, in addition to the set of variables and program point of interest. The dynamic slicing criterion is thus an augmented form of the static criterion. There are two forms of dynamic slice, the first to be introduced (Korel and Laski, 1988), requires that the slice preserve the effect of the original program on V at n when the program is executed with the input sequence I. Subsequently, (Agrawal and Horgan, 1990), introduced a form of dynamic slice which is not executable. Rather it consists simply of the set of statements and predicates which may affect at least one variable in V at n for the input I.

Conditioned (Canfora et al., 1998), or constrained (Field et al., 1995) slicing is an attractive 'bridge' between the polar extremes of dynamic and static slicing. A conditioned slice is constructed with respect to a set of initial states and a static slicing criterion. The set of initial states can be singleton (essentially yielding dynamic slicing) or it can be the set of all states (yielding static slicing).

In forward slicing (Horwitz et al., 1990), the slice contains those statements which are affected by the slicing criterion rather than those which affect it. Forward slicing thus looks 'forward' from the slice point, whereas backward slicing looks 'backwards' from the slice point.

This paper is concerned solely with the static backward slicing paradigm, though the results may be extended to other paradigms. Hereinafter a static backward slice shall be referred to simply as a 'slice'.

(Weiser, 1979) has shown that statement minimal slices are neither computable nor unique. The non–computability of statement–minimal slicing means that the precision of slicing algorithms is of critical importance — the more statements an algorithm can delete, the better the algorithm.

Because a slice captures the thread of computation pertinent to a chosen set of variables at some point in the original program, it has many applications, all of which are based upon the fact that a slice is a simplified version of the original which maintains a projection of its semantics. Some applications of slicing are listed below:

- cohesion measurement (Bieman and Ott, 1994; Ott and Thuss, 1993; Lakhotia, 1993),
- algorithmic debugging (Shahmehri, 1991; Kamkar, 1993),
- re-engineering (Liu and Ellis, 1993; Simpson et al., 1993; Lakhotia and Deprez, 1998),
- component re-use (Beck and Eichmann, 1993; Cimitile et al., 1996),
- automatic parallelisation (Weiser, 1983),
- program comprehension (De Lucia et al., 1996; Harman and Danicic, 1997; Jackson and Rollins, 1994),
- maintenance and debugging (Gallagher and Lyle, 1991; Lyle and Weiser, 1987),
- testing (Harman and Danicic, 1995; Binkley, 1998; Gupta et al., 1992),
- program integration (Horwitz et al., 1989)
- validation and certification (Krinke and Snelting, 1998).

(Tip, 1995) and (Binkley and Gallagher, 1996) provide detailed surveys of the paradigms, applications and algorithms for program slicing.

Gallagher and Lyle (Gallagher and Lyle, 1991) introduced the decomposition slice with the aim of aiding the software maintainer in assessing the impact of system changes. The decomposition slice captures all computation on a single variable and so isolates the 'decomposition' of a program with respect to that variable. The decomposition slice with respect to a variable v, consists of the union of the set of static slices constructed with respect to $(\{v\}, n_i)$, where each n_i is a point in the program at which the value of v is output (together with the last line of the program).

For many of these applications (and particularly for maintenance problems) the slicing algorithm must be capable of constructing slices from 'spaghetti' programs, written before the benefits of structured programming were fully appreciated. The archetype of this unstructured programming style is the `goto` statement; all forms of 'jump' statement, such as `break` and `continue` can be regarded as special cases of the `goto` statement. Hereinafter the words 'goto' and 'jump' should be viewed as denoting any of these forms of statement.

Several authors have published results concerning the problems associated with slicing programs which contain `goto` statements in the static (Ball and Horwitz, 1993; Choi and Ferrante, 1994; Agrawal, 1994) and dynamic (Korel, 1995) paradigms. Such programs are said to exhibit 'arbitrary control flow' and are considered to be 'unstructured'. Unfortunately, the published approaches concerned with the static paradigm either produce unnecessarily large slices or involve the introduction into the slice of new `goto` statements which are not present in the original program. For example, consider the program in the left hand box of Figure 2.

The algorithms for slicing structured programs incorrectly fail to include any `goto` statements in any slice of this program. The modified algorithms of (Agrawal, 1994), (Ball and Horwitz, 1993) and (Choi and Ferrante, 1994) (first algorithm) all produce the entire program as the slice on the criterion $(\{x\}, 8)$. This is unnecessarily large. A thinner slice is depicted in the right hand box of Figure 2 for the same criterion.

The algorithm introduced in this paper produces the slice in the right hand box of Figure 2. Based upon Agrawal's approach (Agrawal, 1994), it produces syntactic subsets of the original program and which are thinner than the slices produced by (Agrawal, 1994), (Ball and Horwitz, 1993) and the first algorithm of (Choi and Ferrante, 1994).

The rest of the paper is organised as follows: Sections 2 and 3 review the problem of slicing unstructured programs and the solution proposed by Agrawal and others. Section 4 introduces the new algorithm, based on two propositions concerning CFGs of unstructured programs. In section 4 these propositions are stated informally. A more formal proof can be found in the appendix.

The definition of a Control Flow Graph (CFG), post dominance and control dependence used in this paper are those used by (Ferrante et al., 1987). These standard definitions are also to be found in the appendix.

2 BACKGROUND

The rest of this paper depends upon some (mostly standard) definitions which are reproduced (for convenience) in section 2.1 below.

2.1 Preliminary Definitions and Notation

Definitions 2, 3 and 5 are taken from (Ferrante et al., 1987). Definition 4 is taken from (Agrawal, 1994) and definition 1 is taken from (Hecht, 1977).

In what follows, the set of edges and nodes of a graph G shall be denoted $E(G)$ and $N(G)$ respectively.

Definition 1 (Path) A *path* is a finite sequence of two or more nodes, denoted $< x_1, \ldots, x_k >$, such that $(x_i, x_{i+1}) \in E(G)$ for $1 \leq i \leq k-1$.

Definition 2 (Control Flow Graph (CFG)) A *Control Flow Graph* (CFG) is a directed graph, G, augmented with a unique *entry* node START and a unique *exit* node STOP such that each node in the graph has at most two successors. It is assumed that nodes with two successors have attributes "T" (*true*) and "F" (*false*) associated with the outgoing edges in the usual way. It is further assumed that for any node N in G there exists a path from START to N and a path from N to STOP.

For slicing applications, edge labellings are not required.

Definition 3 (Post Dominance) A Node V is *post dominated* by a node W in G iff $W \neq V$ and every path from V to STOP contains W.

Observe that the post dominance relation is transitive and irreflexive and therefore is also asymmetric. Thus it will not be possible for two nodes to be post dominated by one another.

Definition 4 (Immediate Post Dominance) A post dominator m of a node p is the (unique) *immediate post dominator* of p, if m is post dominated by every other post dominator of p.

Definition 5 (Control Dependence) Let G be a control flow graph. Let x and y be nodes in G. y is *control dependent* on x iff

1. there exists a path P from x to y such that for all z in P, if $z \neq x$ and $z \neq y$ then z is post dominated by y and

2. x is not post dominated by y.

Observe that if a node q is control dependent upon a node p then when execution follows one edge of p it will definitely arrive at q (provided that the program terminates), whereas when execution follows the other edge of p it may avoid q. If a node x is control dependent upon a node y then y is said to *control* x.

The set of controlled nodes of a node n in a CFG, G, will be denoted $\mathcal{C}_G(n)$, the immediate lexical successor of a node n will be denoted ILS(n), the immediate post dominator of a node n will be denoted IPD(n) and the set of `goto` nodes in a CFG, G, will be denoted $\mathcal{G}(G)$.

The additional definition of a sub CFG is also required and is defined below.

Definition 6 (Sub CFG) From a node p in a CFG G, a sub CFG $\mathcal{S}_G(p)$ is constructed as follows. The entry node is p and the exit node is IPD(p). Let X be the set of paths in G from p to IPD(p), which contain only one occurrence of IPD(p) (by definition, the last node on the path). The nodes of $\mathcal{S}_G(p)$ are those found on any path in X. The edges of $\mathcal{S}_G(p)$ are those found along any path in X.

Observe that, as X consists of paths which only contain one occurrence of IPD(p) (at the end of the path), IPD(p) has no outgoing edges in a sub CFG. Also, observe that a sub CFG is a CFG. That is, since p is post dominated by IPD(p) in G, all paths from p to the exit node in G pass through IPD(p) and therefore all nodes in $\mathcal{S}_G(p)$ can reach IPD(p). Furthermore, as the nodes in X are taken from paths starting with p, all nodes in X can be reached from p. Finally, as the nodes in $\mathcal{S}_G(p)$ are drawn from G they must have at most two successors.

2.2 The Problem of Slicing Unstructured Programs

Both the algorithm introduced here and those previously published, are based upon the Program Dependence Graph (Horwitz et al., 1990; Ottenstein and Ottenstein, 1984), although the results presented here could easily be applied to the iterative data and control flow algorithm first proposed by (Weiser, 1979) and to the algorithm introduced by (Danicic et al., 1995), which is essentially a 'parallelised' version of Weiser's algorithm.

The Program Dependence Graph (PDG), when applied to the problem of slice construction, contains two kinds of dependence edge: a data (or flow) dependence edge and a control dependence edge. There is a data dependence edge between x and y if and only if x defines a variable v which is used at node y and there is a path in the Control Flow Graph from x to y in which no node other than x (and possibly y) defines the variable v. There is a control dependence edge between x and y if and only if the node x 'decides' whether node y is definitely executed or not (Ferrante et al., 1987).

Using the PDG, a slice is constructed with respect to a slightly restricted form of slicing criterion (a node n), in which the set of variables V, mentioned in Weiser's criterion (Weiser, 1984), is the set of referenced variables at node n. The slice consists of all nodes obtained by applying the transitive closure of the inverse of the PDG to the node n. Using the PDG, slicing is thus reduced to a problem of graph reachability. Hereinafter the slice produced by the conventional PDG-based approach shall be referred to as the 'PDG slice'.

The conventional PDG-based algorithm for program slicing, developed for the structured programming paradigm, behaves incorrectly (Agrawal, 1994; Choi and Ferrante, 1994; Ball and Horwitz, 1993) in the presence of arbitrary control flow, because it fails to include any `goto` node in a slice. `goto` nodes are (incorrectly) not included because

1. A `goto` node is not a predicate and so it cannot be the source of a transitive control dependence in the PDG.

2. A `goto` node does not affect any variable and so it cannot be the source of a transitive data dependence in the PDG.

Consider for example the program in Figure 3. The slice constructed from the PDG of this program for the slicing criterion $(\{p\}, 9,)$ will consist of nodes 2, 3, 4 and 7. It should, however, also include the `goto` statements (denoted by nodes 5 and 8). As (Agrawal, 1994) points out, the PDG–based algorithm can be trivially modified to recognise *conditional* `goto` statements:

> "if the predicate in a conditional jump statement is included in a slice because some other statement is control dependent on it, the associated jump must also be included, for the predicate will not serve any purpose in the slice without the accompanying jump." (Agrawal, 1994)

According to this simple amendment to the conventional PDG–based algorithm, the slice of the program in Figure 3 would consist of nodes 2, 3, 4, 5 and 7. The problem is with the inclusion or otherwise of *unconditional* `goto` statements (such as node 8 in this example). This problem has been partially addressed by the approaches described in the next section, but as will be shown, each approach is not quite satisfactory.

3 RELATED WORK

This section presents related work on slicing unstructured programs. The algorithm introduced in this paper is a modification of Agrawal's algorithm, which is described in section 3.1, the other two approaches are due to Ball and Horwitz and Choi and Ferrante. These two approaches are presented in section 3.2.

3.1 Agrawal's Algorithm

(Agrawal, 1994) proposes a method for slicing programs with arbitrary control flow, in which a `goto` statement g from the original program is added to the PDG slice if and only if the immediate lexical successor of g in the slice is different to its immediate post-dominator in the slice. The immediate lexical successor of a statement is the point to which control would pass, were the statement to be removed. Agrawal's definition of immediate lexical succession is reproduced in definition 7 below. Having included such a `goto` statement, the algorithm goes on to include any nodes upon which the `goto` has transitive dependence.

Observe that, as a `goto` statement references no variables, it cannot be directly *data* dependent upon any node. The inclusion of the transitive dependencies of a `goto` node thus reduces to the problem of including any predicate upon which the goto is control dependent and consequently including the transitive dependencies (namely the slice) of any such predicate.

Agrawal's algorithm is reproduced in Figure 4.

Definition 7 (Agrawal's Immediate Lexical Successor) A statement, S', is said to be the *immediate lexical successor* of a statement, S, in a program if deleting S from the program will cause the control to pass to S' whenever it reaches the corresponding location in the new program. If S is a component statement, such as an *if* or a *while* statement, deleting it means deleting it along with the statements that constitute its body.

Agrawal's approach produces correct, but unnecessarily large slices, since it includes predicates and the program components upon which they depend which, as demonstrated formally in Proposition 1, turn out to be unnecessary according to Weiser's definition of a program slice (Weiser, 1984). The inclusion of such unnecessary predicates may make slices very large due to the transitive dependencies of the unnecessary predicate. To see why the slices produced by Agrawal's algorithm are unnecessarily large, consider the program[1] and its CFG given in Figure 5. In Figure 5 control flow edges are represented by solid edges, whilst the dotted edge represents the immediate lexical successor of the goto node. The PDG slice constructed with respect to the slicing criterion $(\{\mathtt{y}\}, 5)$ consists of nodes 1 and 4 alone. In this case this turns out to be a *correct* and *statement-minimal* slice.

The algorithm introduced by (Agrawal, 1994) will produce the (unnecessary large, but semantically correct) slice containing the entire original program; node 3 will be included because its immediate lexical successor (node 5) differs from its immediate post dominator (node 4) and node 2 will be included because node 3 is control dependent upon it.

As pointed out by (Choi and Ferrante, 1994), the unnecessary inclusion of node 2 can lead to the inclusion of many other nodes, because (using Agrawal's approach) its inclusion requires the inclusion of any nodes upon which node 2 is either transitively control or data dependent. That is, the slice with respect to the predicate (node 2) is also added to the PDG slice to preserve the meaning of the predicate.

Observe that the inclusion of such a predicate creates a program (the slice) in which a predicate controls only goto nodes. Clearly, in most 'normal' programs, one would not expect to find such predicates. However, this is not, as one might initially expect, merely a 'pathological special case' as slices are not 'normal' programs; slicing will remove statements, making the existence of such predicates more likely. For example, consider again the program in Figure 2. The CFG of this program is depicted in Figure 6. In this program, the goto statement at node 5 serves to avoid the execution of node 6 should the predicate at node 4 evaluate to true. The goto node thus plays the role of a continue statement. This is typical in legacy systems written in older languages where the programmer is forced to use goto nodes to define control flow. After slicing the program, for example with respect to the slicing criterion $(\{\mathtt{x}\}, 8)$, node 6 may disappear, creating a predicate (at node 4) which controls only goto nodes. (This example will be treated in more detail in section 4.4.)

3.2 Other Algorithms for Slicing Unstructured Programs

(Ball and Horwitz, 1993) and (Choi and Ferrante, 1994) introduced two identical algorithms for constructing slices of unstructured programs. The approach involves modifying the CFG of a program, creating an 'Augmented Control Flow Graph' (ACFG). The PDG constructed from the ACFG contains the PDG constructed from the CFG.

The ACFG is identical to the CFG, except that a goto node contains two outgoing edges, one of which is the 'normal' edge, which connects the goto node to its target[2], with the other (dummy) edge, being introduced to model the effect of deleting the goto. This dummy edge connects a goto statement to what Agrawal terms its 'immediate lexical successor' (Agrawal, 1994).

As (Ball and Horwitz, 1993) point out, the introduction of such a dummy edge need not alter the semantics of the ACFG with respect to the corresponding CFG[3], because the dummy edge may be labelled 'False', and the normal edge 'True'. The goto node can then be considered to be a 'degenerate' predicate node in which the controlling boolean expression always evaluates to 'True'.

(Ball and Horwitz, 1993) prove that a PDG constructed from and ACFG will include all the necessary goto statements in a slice in order to make it correct with respect to the slicing criterion. (Choi and Ferrante, 1994) independently discovered the same approach.

[1]This example is the one used by (Choi and Ferrante, 1994).

[2]Observe that the target of a goto statement is its immediate post dominator.

[3]The CFG corresponding to an ACFG is constructed by removing all dummy edges from the ACFG.

(Choi and Ferrante, 1994) report that the ACFG produces identical slices to those constructed using Agrawal's algorithm, and so it suffers from the same weakness; unnecessary control predicates are included along with the nodes upon which they transitively depend. It is less clear, however, how this approach might be modified in light of Propositions 1 and 2, and so the approach taken here takes Agrawal's algorithm as a starting point.

The dummy edges result in spurious control dependence relations in the PDG, which in turn, can cause the inclusion of unnecessary predicates. For example, consider the program and its CFG encountered earlier in Figure 5. This figure depicts the ACFG of the program fragment; solid edges represent CFG edges, the dotted edge represents the additional ACFG edge. The PDG constructed from this ACFG is depicted in Figure 7. In Figure 7 solid edges represent control dependence, while dotted edges represent data dependence. The PDG in Figure 7 contains two control dependence edges that would not have been found in the PDG constructed from the CFG of the program in Figure 5. Namely

1. The edge from node 3 to node 4.
   This edge is added to ensure that the `goto` (node 3) and its controlling predicate (node 2) are included in any slice which includes node 4.

2. The edge from node 2 to node 4.
   This is what Choi and Ferrante (Choi and Ferrante, 1994) term a 'spurious' control dependence, which occurs because the `goto` in the CFG has become a predicate in the ACFG, with the result that there is a new path from node 2 to the exit which does not contain node 4.

The slice constructed from this (artificially large) PDG for the slicing criterion $(\{y\}, 5)$ consists of the entire program. As previously observed, this slice is unnecessarily large (nodes 2 and 3 need not be included, and indeed, are not included by the new algorithm).

It should be pointed out that it is not only the inclusion of spurious control dependencies which leads to the construction of unnecessarily large slices by the ACFG–based algorithm. The theoretical results presented here demonstrate that there exist `goto` nodes whose target *differs* from their immediate lexical successor, but which, nonetheless, need *not* be added to the PDG slice in order to construct a correct final slice.

Choi and Ferrante introduce a second algorithm which introduces *new* `goto` statements into the conventional PDG slice, so as to make it correct with respect to the slicing criterion. Although Choi and Ferrante prove that this approach does yield a reduced program which respects the slicing criterion, their algorithm does have the drawback that the 'slices' so created, are not syntactic subsets of the programs from which they are constructed.

The Choi and Ferrante algorithm is defined for an intraprocedural language with similar (non-block structured) statement constructs as those found in Fortran IV. (Lakhotia and Deprez, 1997) have an algorithm for computing slices which may also add new `goto` statements (like the Choi and Ferrante algorithm), but which is applicable to a wider class of languages, which allow `goto` statements to jump into and out of statement blocks in the way that, for example, Fortran 77 allows.

4 THE NEW ALGORITHM

This section presents the new algorithm for slicing unstructured programs. The starting point for the new algorithm is the approach adopted by Agrawal. The modifications to Agrawal's approach do not require a major re-implementation, but they do require formal justification. This justification is provided by two propositions concerning CFGs in which a predicate controls only `goto` nodes (and perhaps itself). In such a situation it is shown that:

1. The outcome of the evaluation of the predicate can only serve to cause non-termination. That is, either the program fails to terminate or control reaches the immediate post dominator of the predicate without a change of state.

2. Either the immediate post dominator of the predicate is its immediate lexical successor or there must exist a cycle-free path from the predicate to its immediate post dominator containing at least one `goto` node.

The first of these observations demonstrates that such a predicate is unnecessary to preserve the *semantic* requirement of a slice, while the second suggests a method for avoiding such predicates while retaining the *syntactic* requirement that a slice must be a subset of the program from which it is constructed.

More formally

1. Let p be a side-effect free predicate in a CFG G. If $\mathcal{C}_G(p) \subseteq \{p\} \cup \mathcal{G}(G)$ then on every path from p to its immediate post dominator no state change will occur. The only observable affect p could have would therefore be to cause non–termination.

2. Let p be a predicate in a CFG G.
If $\mathcal{C}_G(p) \subseteq \{p\} \cup \mathcal{G}(G)$ and $\mathrm{IPD}(p) \neq \mathrm{ILS}(p)$ then there must exist a cycle-free path from p to $\mathrm{IPD}(p)$ consisting of a non-empty set of `goto` nodes in $\mathcal{C}_G(p)$.

These two propositions are formally proved correct in the appendix. Section 4.1 explains how the new algorithm is constructed by appeal to these two propositions. The algorithm itself, is presented in Figure 8.

4.1 Justification of the New Algorithm

The new algorithm is presented in Figure 8. This section justifies the algorithm by appeal to Propositions 1 and 2.

Several collections of nodes will be referred to. The PDG slice and the PDG slice to which several nodes have been added and the final slice (the result of the algorithm). Strictly speaking, a collection of nodes does not form a CFG. Where a collection of nodes is referred to, there is however, an *implied* CFG, constructed as a projection (onto this collection of nodes) of the original program's CFG. This projection is formalised in definition 8 below.

Definition 8 (Projected CFG)
Let G be a CFG and let N be the set of nodes in G. From G and some set of nodes X, $X \subseteq N$, the projected CFG, $G \rhd X$, is defined as follows:

$$\begin{array}{rll} (s,t) \in G \rhd X & \text{iff} & \{s,t\} \subseteq X \\ & \text{and} & \text{there exists a path from } s \text{ to } t \text{ in } G \\ & & \text{containing no node in } X \text{ other than } s \text{ and } t \end{array}$$

Thus $G \rhd X$ is constructed by 'connecting up' the nodes in X which become detached when those nodes not in X are removed. $G \rhd X$ forms the basis for Choi and Ferrante's second algorithm (Choi and Ferrante, 1994), in which new `goto` nodes are added to the CFG to achieve this 'connecting up' explicitly.

Let *Slice* denote the set of nodes in the PDG slice, let *Org* denote the CFG of the program to be sliced and let Γ denote the set of `goto` nodes in *Org*.

Starting with Agrawal's approach, a predicate p will be considered for inclusion in the final slice because it controls a `goto` node whose immediate lexical successor differs from its immediate target.

If p is already in the PDG slice, then all that is required is to add the `goto` node. If p is not in the PDG slice then it is to be included solely because it controls a `goto` node, g, in which case Proposition 1 will apply to the the sub CFG constructed from the PDG slice, the predicate p and the `goto` nodes it controls. More formally, Proposition 1 will apply to the sub-CFG constructed from p and the projected CFG, $Org \rhd (Slice \cup \Gamma \cup \{p\})$.

Proposition 1 establishes that the predicate need not be included in the final slice to preserve the effect of the original upon the slicing criterion. Since the predicate is not required in the final slice, the transitive dependencies of the predicate are also not be required.

There remains the problem of constructing a final slice from the original program, so as to avoid the execution of p. For the final slice to be correct it must be ensured that control passes from all immediate predecessors of p to its immediate post-dominator.

The sub CFG constructed for p and the CFG $Org \rhd (Slice \cup \Gamma \cup \{p\})$ must take the form of one of the two CFGs pictured in Figure 9, where vectors denote paths consisting of zero or more `goto` nodes[4].

If, in the CFG $Org \rhd (Slice \cup \Gamma \cup \{p\})$, the immediate post dominator of p is the immediate lexical successor of p then the whole sub CFG need not be included. With neither the predicate nor the `goto` nodes it controls in the final slice, control will pass from any immediate predecessor of p to its immediate post dominator (namely, its immediate lexical successor), as required.

If the immediate post dominator of p differs from its immediate lexical successor, then by Proposition 2, there must exist, in the sub CFG constructed from p and the CFG, $Org \rhd (Slice \cup \Gamma \cup \{p\})$, a cycle-free path,

[4]More formally (in terms of the lemmata in the appendix), by lemma 5, there must exist a cycle-free path from p to $\mathrm{IPD}(p)$ and, by lemma 2, any path from p to $\mathrm{IPD}(p)$ may only contain `goto` nodes and, by lemma 4, any cycle must contain p. Thus, either both paths are cycle-free (right-hand CFG of Figure 9) or only one is (left-hand CFG of Figure 9).

π, from p to its immediate post dominator, consisting of a non-empty set of `goto` nodes. The first node, after p, along π, must either be the consequent or alternative node of p. Suppose it is the consequent node. The final slice can be constructed by including the consequent node of p and all nodes on π. The predicate p itself, and its alternative branch are not required. In the final slice, control will thus pass from any immediate predecessor of p onto its consequent node, and along π to the immediate post dominator of p.

If the first node along π, after p, is not the consequent node of p, then it must be the alternative node of p. Therefore, to form the final slice, p and its consequent node need not be included. The alternative node and all other nodes along π must be included in the final slice. Control will thus pass from any immediate predecessor of p to its alternative node, and along π to the immediate post dominator of p. Of course, if could turn out that both paths from p to its immediate post dominator are cycle free. In this case there will be a choice, but should one path be already in *Slice* then no further action need be taken.

Finally, observe that such paths, π, may contain `goto` nodes which target their immediate lexical successor. These need not be included in the final slice, but for clarity of exposition, their deletion has been relegated to a post–processing optimisation. A slightly more efficient version of the new algorithm would remove such `goto` nodes as they are encountered along π.

4.2 A Note on Side–effects

Proposition 1 is only valid for predicates which are side–effect free. However, where a predicate contains side–effects, the algorithm presented in Figure 8 will still produce correct slices. This is because any predicate considered for inclusion by Agrawal's algorithm is not in the PDG slice. Therefore, no node in the PDG slice can be data dependent upon it — the predicate is side–effect free *with respect to the slicing criterion* and so, in the special case of slice–construction, predicates not in the PDG slice may be considered to be 'side–effect free', in the sense in which the phrase is used in Proposition 1.

For example, suppose that node 2 of Figure 5 had been `k == v++`, a predicate which tests the boolean `k == v`, while having the side-effect of incrementing the value of the variable `v` by one. Suppose further that the slicing criterion is $(\{\mathtt{y}\}, 5)$, for which the PDG slice consists of nodes 1 and 4 alone.

Notice that the variable `v` can be (in this example) any variable except for `x`, because if it had been `x` then the predicate at node 2 would have been included in the PDG slice of the original program.

Now, removing node 2 from the program to form the slice *will* affect the semantics of the program with respect to the final value of `v`, but will *not* affect the slicing criterion because node 2 is 'side–effect free' with respect to the slicing criterion.

In general, if some predicate is not included in the PDG slice, then there can be no transitive data dependence between it and the slicing criterion, therefore any side effect of the predicate's evaluation cannot have any effect with respect to the slicing criterion. If, on the other hand, the predicate is included in the PDG slice, then the problems addressed by the algorithm presented here do not arise.

4.3 Complexity

The complexity of the algorithm depends upon the complexity of the algorithms for constructing the conventional PDG slice and for constructing the post dominator tree. If interprocedural slicing is being used, this will require the construction of a System Dependence Graph[5].

Recently, a linear algorithm for constructing post dominator trees was proposed (Alstrup et al., 1997).

The most commonly used algorithm (Horwitz et al., 1990) for (interprocedural) slice construction uses the System Dependence Graph (SDG) — a generalisation of the Program Dependence Graph which caters for interprocedural slicing. The SDG slicing algorithm consists of three phases.

1. The construction of PDGs for each procedure

2. The construction of the summary edges, forming an SDG

3. The construction of a slice from the SDG

[5]Throughout this paper it has been assumed that a `goto` statement may transfer control neither into nor out of a procedure body. However, if interprocedural slicing is used, slices may cross procedure boundaries, including statements from more than one procedure in the 'PDG slice'. The algorithm presented here (and those described in section 3) decide upon the inclusion of `goto` statements for *individual* procedures. If statements from more than one procedure are included then the algorithm should be executed once for each procedure which contributes to the slice.

The algorithm for computing the PDGs is quadratic in N, the number of vertices of the SDG, while the problem of computing the slice itself from the SDG is linear in E, the total number of edges in the SDG. The problem of constructing the summary edges is the most computationally expensive phase of the algorithm.

For this phase, (Horwitz et al., 1990) report a time-complexity of

$$O(T \cdot e \cdot F + T \cdot S^2 \cdot F^4)$$

Where T is the total number of call sites, e is the maximum number of edges of any of the constituent PDGs, F is the largest number of formal input parameters in any of the PDGs and S is the maximum number of call sites in any of the PDGs.

More recently, the problem of interprocedural slicing was recast as a context free language reachability problem (Reps et al., 1994), yielding an improvement in time-complexity for the construction of summary edges. For the new algorithm, the authors report a time-complexity for this phase of

$$O(P \cdot e \cdot F + T \cdot F^3)$$

where P denotes the number of procedures in the program.

Having constructed the PDG slice, the algorithm presented here walks over the program's post dominator tree in a preorder traversal, looking for `goto` nodes whose target differs from their lexical successor. In the worst case, every predicate in the program is not in the PDG slice, has an immediate lexical successor which differs from its immediate post dominator and controls a `goto` node. In this case the algorithm has to follow the `then` and `else` paths of each predicate to find a cycle free path to the predicate's immediate post dominator. The worst case for this search involves considering all nodes of the program's CFG.

The complexity of this phase of the algorithm is thus

$$O(\pi \cdot e)$$

where π is the maximum number of predicates in any PDG.

Thus the largest part of the algorithm can be seen to be the construction of the original PDG slice, while the problem of identifying which `goto` nodes to include is a lesser post processing phase from the point of view of time complexity.

Agrawal's algorithm involves re-slicing the program for each non-PDG-Slice predicate which controls a `goto`. This requires a similar time-complexity to the algorithm presented here because, at this stage, the SDG is already constructed and so slicing only requires linear time (though this time linear in E, not in e) for each predicate node.

4.4 Example Executions of the New Algorithm

Consider the program in Figure 5. The PDG slice for the criterion $(\{y\}, 5)$ consists of nodes 1 and 4. The `goto` statement at node 3 is a candidate for inclusion because its immediate lexical successor (node 5) differs from its target (node 4). The `goto` is also controlled by a predicate (node 2) which is not in the PDG slice. The immediate lexical successor (node 5) of the predicate is, however, identical to its immediate post dominator and so no further action is taken. The final slice is thus nodes 1 and 4, which, in this case, is statement minimal. The other algorithms for slicing unstructured programs described earlier would have included the whole program in the slice.

The program in Figure 5 is a rather 'pathological' example. The program in Figure 6 is more typical. The PDG slice of this program, constructed with respect to the criterion $(\{x\}, 8)$, consists of nodes 1, 3 and 8. Thus *Slice* is, initially nodes 1, 3 and 8. The `goto` nodes of the program must be considered in a preorder traversal of the post dominator tree (which is depicted in Figure 10).

Node 2 is considered first. Its target in $Slice \cup \Gamma$ (node 8) differs from its immediate lexical successor in $Slice \cup \Gamma$ (node 3), but it is not control dependent upon any node which is not in *Slice*, so it is simply added to *Slice*.

Next, node 5 is considered. Its target in $Slice \cup \Gamma$ (node 1) differs from its immediate lexical successor in $Slice \cup \Gamma$ (node 7). Node 5 is control dependent upon node 4, which is not in *Slice*. There is, however, a cycle free path from node 4 to its immediate post dominator (node 1). The path is $< 4, 5, 1 >$, so node 5 is added to *Slice*.

Finally, node 7 is considered. Its target in $Slice \cup \Gamma$ (node 1) also differs from its immediate lexical successor in $Slice \cup \Gamma$ (node 8). Node 7 is also controlled by node 4, but, as with node 5, the cycle free path $< 4, 5, 1 >$ is used to avoid the unnecessary inclusion of node 4. Node 5 is added to *Slice*. As it is already in *Slice*, the overall result of processing node 7 leaves *Slice* unaffected.

The final slice thus consists of nodes 1, 2, 3, 5 and 8. (The original program and the final slice produced by the new algorithm were depicted in Figure 2.)

The slice produced by Agrawal's algorithm would have contained the entire program. Node 7 would have been included because its immediate lexical successor (node 8) differs from its immediate post dominator (node 1). Node 4 would have been included because it controls nodes 5 and 7 and node 6 would have been included because node 4 is data dependent upon it. Other approaches to the problem (the first algorithm of (Choi and Ferrante, 1994) and the algorithm introduced by (Ball and Horwitz, 1993)) would also produce the entire program as the slice.

As a final example[6], consider the slightly more complex program in left hand section of Figure 11. The CFG of this program is depicted in Figure 12, and its post dominator tree is depicted in Figure 13.

Suppose the slicing criterion is $(\{\mathtt{x}\}, 15)$, so the slice must preserve the effect of the original on the final value of the variable `x`. The PDG slice consists of nodes 1,2,5,11,13 and 14. That is, all the assignments to `x` and the initial predicate.

The `goto` nodes are considered in a preorder traversal of the post dominator tree. This gives the order of consideration: 12,6,10,9,4.

The algorithm starts by considering node 12. The immediate lexical successor of this node is node 13, while its target is node 14. The only predicate which controls node 12 is node 1, which is already in *Slice*, so node 12 is simply added to *Slice*.

Node 6 is included because its immediate lexical successor in $Slice \cup \Gamma$ is node 9, while its target in $Slice \cup \Gamma$ is node 11. Node 6 is not controlled by any nodes, and so there are no 'predicate issues' to worry about.

Node 10 is included because its immediate post dominator (node 13) differs from its immediate lexical successor (node 11), but, like node 12, it is only controlled by node is node 1.

Node 9 is more interesting. Its target in $Slice \cup \Gamma$ is node 9, while its immediate lexical successor in $Slice \cup \Gamma$ is node 10. Furthermore it is controlled by node 7, which is not in the PDG slice. The immediate lexical successor of node 7 is node 11, while its immediate post dominator is node 10. Therefore, there must be a cycle free path from node 7 to its immediate post dominator (this is guaranteed by Proposition 1). The path is $< 7, 10 >$, so node 10 is included in *Slice*. As node 10 is already in *Slice* this inclusion has no effect.

Node 4 is less interesting. Its immediate lexical successor (node 5) differs from its immediate post dominator (node 9), but it is not controlled by any nodes not in the PDG slice, and so it is simply included in *Slice*.

The final slice therefore includes all the `goto` nodes except node number 9. It is depicted in the right hand section of Figure 11. Observe that, as with Agrawal's algorithm, labels from the original program are associated with their nearest lexical successor in the slice.

5 CONCLUSIONS AND FUTURE WORK

The work presented here is based upon previous work by Agrawal on slicing unstructured programs. Agrawal's approach produces correct (but unnecessarily large) slices due to the inclusion of unnecessary control predicates and their transitive dependencies. Other currently published algorithms either produce similar 'over-large' slices or include new `goto` nodes not in the original program.

By analysing the nature of control dependence in unstructured programs, the reason for the inclusion of unnecessary predicates has been identified, leading to the introduction of a new algorithm for slicing unstructured programs which avoids the problem. The analysis reveals that it is always possible to separate the problem of calculating program dependencies from the problem of deciding how to reconstruct a valid slice from the set of non–`goto` nodes identified during such a dependency analysis.

The introduction of a more precise algorithm for slicing unstructured programs improves the applicability of slicing to maintenance problems which are often concerned with programs which exhibit the kind of unstructured control flow considered here.

Work is underway to implement a slicing tool for FORTRAN 77, based upon the algorithm presented here. The completion and evaluation of this tool remains a problem for future work.

[6]This example was suggested by Arun Lakhotia.

Acknowledgements
Arun Lakhotia provided invaluable help in testing the algorithm presented here with imaginative examples. He also shared with the authors many extremely helpful insights which greatly improved their understanding of the problem.

The authors would also like to thank the anonymous referees for their suggestions and observations which also helped to improve the paper. Any errors which remain are, naturally, the sole responsibility of the authors.

A CORRECTNESS PROOF

This section presents a formal investigation of the two propositions upon which the correctness of the algorithm is based.

A.1 The 'No State Change' Proposition

Proposition 1 shows that a predicate which controls only `goto` nodes can affect, neither directly nor indirectly, the value of any program variable — it can only serve to cause non–termination.

The proof of Proposition 1 rests upon lemmas 1 and 2, which are stated and proved below.

Lemma 1 Let p be a predicate in a CFG G. IPD(p) is the only post dominator of p in $\mathcal{S}_G(p)$.

Proof
Suppose there exists in $\mathcal{S}_G(p)$ a node $n \neq$ IPD(p) which is a post dominator of p.

Since n is in $\mathcal{S}_G(p)$ it must, by definition 6 of a sub CFG, occur on a path from p to IPD(p) in which the only occurrence of IPD(p) is the last element of the path. Therefore there cannot be a path from n to the exit node which does not contain IPD(p) because this would mean that there was a path from p to the exit node (via n) which does not contain IPD(p) and therefore IPD(p) would not post dominate p.

Since there can be no path from n to the exit node which does not contain IPD(p), n must be post dominated by IPD(p), but as IPD(p) is the *immediate* post dominator of p, IPD(p) must also be post dominated by n. However this is impossible, because the post dominance relation is asymmetric.

Lemma 2 Let p be a predicate in a CFG G. If $\mathcal{C}_G(p) \subseteq \{p\} \cup \mathcal{G}(G)$ then any path in $\mathcal{S}_G(p)$ from p to IPD(p) (excluding IPD(p)), may only contain p and `goto` nodes controlled by p.

Proof
Suppose the contrary.

Let π be a path from p to IPD(p) (excluding IPD(p)) in $\mathcal{S}_G(p)$, which contains at least one node not in $\mathcal{G}(G)$, and let n be the first occurrence on π such a node. Now, since n is the first occurrence of a node not in $\mathcal{G}(G)$, any nodes which occur before n on π must be nodes in $\mathcal{C}_G(p)$, and as $\mathcal{C}_G(p) \cap \mathcal{G}(G)$ contains no predicates, control *must* reach n if execution continues from p along π. Since p does not control n this means (by definition 5 of control dependence) that *every* path from p to IPD(p) must contain n, thus n is a post dominator of p. However, as n cannot be the immediate post dominator of p it cannot (by lemma 1) be in $\mathcal{S}_G(p)$, contradicting the initial definition of π and n.

Proposition 1 Let p be a side-effect free predicate in a CFG G. If $\mathcal{C}_G(p) \subseteq \{p\} \cup \mathcal{G}(G)$ then on every path from p to its immediate post dominator no state change will occur. The only observable affect p could have would therefore be to cause non–termination.

Proof
No state change can occur on any path from p to IPD(p) since (by lemma 2) only `goto` nodes and the predicate p can occur on such a path. Thus, the program either fails to terminate (because p evaluates to true and is on a cycle), or control passes from p to IPD(p) without a change of state.

A.2 Examples

Consider the CFG depicted earlier in Figure 5. The predicate at node 2 controls only the `goto` node at node 3. Whatever the outcome of the evaluation of the predicate, control will pass to node 4 (the immediate post dominator of the predicate) without a change of state.

Consider the CFG depicted earlier in Figure 6. Suppose node 6 is deleted from this program (as may happen when the program is sliced). There would then exist a predicate, node 4, which controls only `goto` nodes (nodes

5 and 7). The evaluation of the predicate merely serves to decide whether control reaches node 1 (the immediate post dominator of the predicate) via node 5 or via node 7. In either case no state change occurs between the execution of a predicate and its immediate post dominator.

A.3 The 'Cycle-Free Path Existence' Proposition

The proof of Proposition 2 rests upon lemmas 3, 4 and 5, which are stated and proved below. It also rests upon an assumption (Assumption 1) concerning the lexical successor of a 'structured predicate'.

Assumption 1 If all cycle-free paths from p to IPD(p) contain no `goto` statements, then ILS(p) = IPD(p).

Assumption 1 is, in effect, a requirement upon the programming language. Approaches to slice construction for structured programs (Horwitz et al., 1990; Weiser, 1984; Danicic et al., 1995). all rest upon this assumption.

In the structured case, the assumption degenerates to the assumption that the immediate post dominator of a predicate is its immediate lexical successor. When constructing a slice from a structured program and where the slicing criterion is not transitively dependent upon the body of a condition–controlled construct, the predicate and its body are removed. In the resulting program, it is assumed that control will pass from any immediate predecessor of the predicate to its immediate post dominator.

However, in the unstructured paradigm, the presence of a `goto` node on some path from a predicate to its immediate post dominator could mean that the immediate post dominator of the predicate is not its lexical successor.

Lemma 3 Let p be a predicate. All controlled nodes of p must occur on some path from p to IPD(p).

Proof
Suppose the contrary. There must exist a node n, such that p controls n, but n does not occur on any path from p to IPD(p). By definition 5 of control dependence, there must exists a path, π, from p to n such that for all z in π, where $z \neq p$ and $z \neq n$ z is post dominated by n. Now since, IPD(p) must also occur on π, n must also post dominate IPD(p), and therefore, by transitivity of control dependence, it must post dominate p. However, by definition 5 of control dependence, p must not be post dominated by n.

Lemma 4 Let p be a predicate in a CFG G. If $\mathcal{C}_G(p) \subseteq \{p\} \cup \mathcal{G}(G)$ then in $\mathcal{S}_G(p)$ every cycle in a path from p to IPD(p) must contain p.

Proof
Suppose there exists a cycle from p to IPD(p) which does not contain p. This would mean (by lemma 2) that only `goto` nodes occur on the cycle and therefore there would be *no* path from any of the nodes on the cycle to the exit node, contradicting the definition of a CFG.

Lemma 5 There must exist a cycle free path from any node n to IPD(n).

Proof
Trivial. Suppose the contrary. If there is no cycle free path from n to IPD(n) then there would be no path from n to IPD(n) and therefore no path from n to the exit node, contradicting the definition of a CFG.

Proposition 2 Let p be a predicate in a CFG G.
If $\mathcal{C}_G(p) \subseteq \{p\} \cup \mathcal{G}(G)$ and IPD(p) $\neq$ ILS(p) then there must exist a cycle-free path from p to IPD(p) consisting of a non-empty set of `goto` nodes in $\mathcal{C}_G(p)$.

Proof
By lemma 5 there must exist a cycle-free path from p to IPD(p). By lemma 2 such a path may only contain `goto` nodes. Were all such paths to contain no `goto` nodes then, by Assumption 1, IPD(p) = ILS(p).

A.4 Examples

Consider the CFG depicted earlier in Figure 6. Suppose node 6 is removed from the program. This creates a predicate (node 4) which control only `goto` nodes (nodes 5 and 7), and whose immediate lexical successor (node 7, since node 6 has been removed) differs from its immediate post dominator (node 1). In this case it turns out that there exist *two* cycle free paths from the predicate to its immediate post dominator ($< 4, 5, 1 >$ and $< 4, 7, 1 >$).

References

Agrawal, H. (1994). On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida. Proceedings in SIGPLAN Notices, 29(6), June 1994.

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

Alstrup, S., Harel, D., Lauridsen, P. W., and Thorup, M. (1997). Dominators in linear time. Technical Report DIKU 97/28, Department of Computer Science, University of Copenhagen.

Ball, T. and Horwitz, S. (1993). Slicing programs with arbitrary control–flow. In Fritzson, P., editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222. Springer. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

Beck, J. and Eichmann, D. (1993). Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518.

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

Binkley, D. W. (to appear 1998). The application of program slicing to regression testing. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing.* Elsevier.

Binkley, D. W. and Gallagher, K. B. (1996). Program slicing. In Zelkowitz, M., editor, *Advances of Computing, Volume 43.* Academic Press.

Canfora, G., Cimitile, A., and De Lucia, A. (to appear 1998). Conditioned program slicing. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing.* Elsevier.

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

Cimitile, A., De Lucia, A., and Munro, M. (1996). A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice*, 8:145–178.

Danicic, S., Harman, M., and Sivagurunathan, Y. (1995). A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313.

De Lucia, A., Fasolino, A. R., and Munro, M. (1996). Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, Berlin, Germany.

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

Field, J., Ramalingam, G., and Tip, F. (1995). Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA.

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

Gupta, R., Harrold, M. J., and Soffa, M. L. (1992). An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Orlando, Florida, USA. IEEE Computer Society Press.

Harman, M. and Danicic, S. (1995). Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162.

Harman, M. and Danicic, S. (1997). Amorphous program slicing. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA.

Hecht, M. S. (1977). *Flow Analysis of Computer Programs*. Elsevier.

Horwitz, S., Prins, J., and Reps, T. (1989). Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–61.

Jackson, D. and Rollins, E. J. (1994). Chopping: A generalisation of slicing. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

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

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

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

Krinke, J. and Snelting, G. (to appear 1998). Validation of measurement software as an application of slicing and constraint solving. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing*. Elsevier.

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

Lakhotia, A. and Deprez, J.-C. (1997). Precise slices of block-structured programs with goto statements.

Lakhotia, A. and Deprez, J.-C. (to appear 1998). Restructuring programs by tucking statements into functions. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing*. Elsevier.

Liu, L. and Ellis, R. (1993). An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK.

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

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

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

Reps, T., Horwitz, S., Sagiv, M., and Rosay, G. (1994). Speeding up slicing. In *ACM Foundations of Software Engineering (FSE'94)*, pages 11–20, New Oleans, LA.

Shahmehri, N. (1991). *Generalized algorithmic debugging*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

Simpson, D., Valentine, S. H., Mitchell, R., Liu, L., and Ellis, R. (1993). Recoup – Maintaining Fortran. *ACM Fortran forum*, 12(3):26–32.

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189.

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

Weiser, M. (1983). Reconstructing sequential behaviour from parallel behaviour projections. *Information Processing Letters*, 17(10):129–135.

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

Author Biographies

Mark Harman received a M.Eng. in Software Engineering from Imperial College, University of London in 1988 and a Ph.D. from the University of North London in 1992. He is currently a lecturer in Computer Science at Goldsmiths' College, University of London. His research interests include program slicing, transformation, measurement and testing and meta-heuristic algorithms.

Sebastian Danicic received B.Sc. in Mathematics from Queen Mary College, University of London in 1977 and an M.Sc. in Computation from Oxford University in 1981. Since 1985 he has been a lecturer in Computer Science at the University of North London. His interests are slicing, functional programming and programming language semantics.

FIGURES

Captions at bottom of page

1	`scanf("%d",&n);`	1	`scanf("%d",&n);`
2	`s=0;`	2	`s=0;`
3	`p=1;`	3	
4	`while (n>0)`	4	`while (n>0)`
5	`{`	5	`{`
6	`s=s+n;`	6	`s=s+n;`
7	`p=p*n;`	7	
8	`n=n-1;`	8	`n=n-1;`
9	`}`	9	`}`
Original Program		Slice w.r.t. $(\{\texttt{s}\}, 9)$	

Figure 1: A Program and One of its Slices

`1    L: if (x==MAX)`	`1    L: if (x==MAX)`
`2            goto end;`	`2            goto end;`
`3        x=x+1;`	`3        x=x+1;`
`4        if (x>y)`	
`5            goto L;`	`5        goto L;`
`6        y=y-1;`	
`7        goto L;`	
`8    end:`	`8    end:`
Original program	Slice produced by the new conventional algorithm

Figure 2: Slicing with respect to the criterion $(\{x\}, 8)$

```
      s=0;
      p=1;
loop:  read(n);
      if (n<0)
         goto end;
      s=s+n;
      p=p*n;
      goto loop;
end:
```

Figure 3: A Typical ‘goto’ program

Slice = the PDG slice;

do begin

  Traverse the postdominator tree using the *preorder* traversal, and for each jump statement, J, encountered that is (i) not in *Slice* and (ii) whose nearest postdominator in *Slice* is different from the immediate lexical successor in *Slice*

  do begin

    Add J to *Slice*;

    Add the transitive closure of the dependencies of J to *Slice*;

  end

end

For each `goto` statement, *Goto* L, in *Slice*, if the statement labelled L is not in *Slice* then associate the label L with its nearest postdominator in *Slice*;
return (*Slice*);

Figure 4: Agrawal's Algorithm for Slicing Unstructured Programs

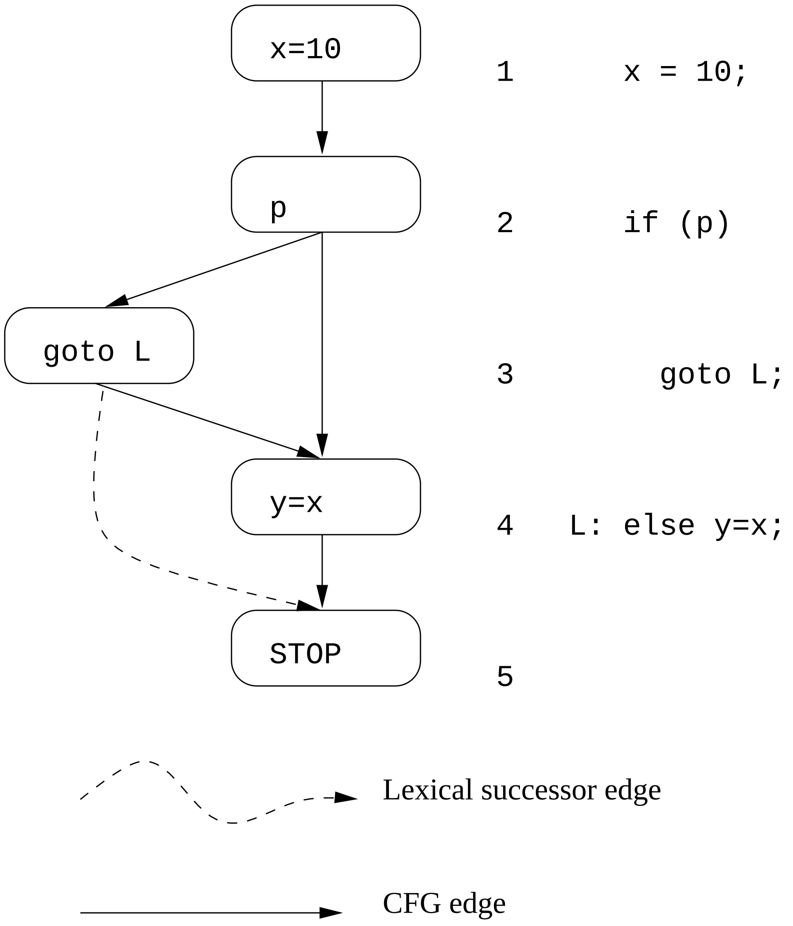

Figure 5: The Unnecessary Predicate Problem

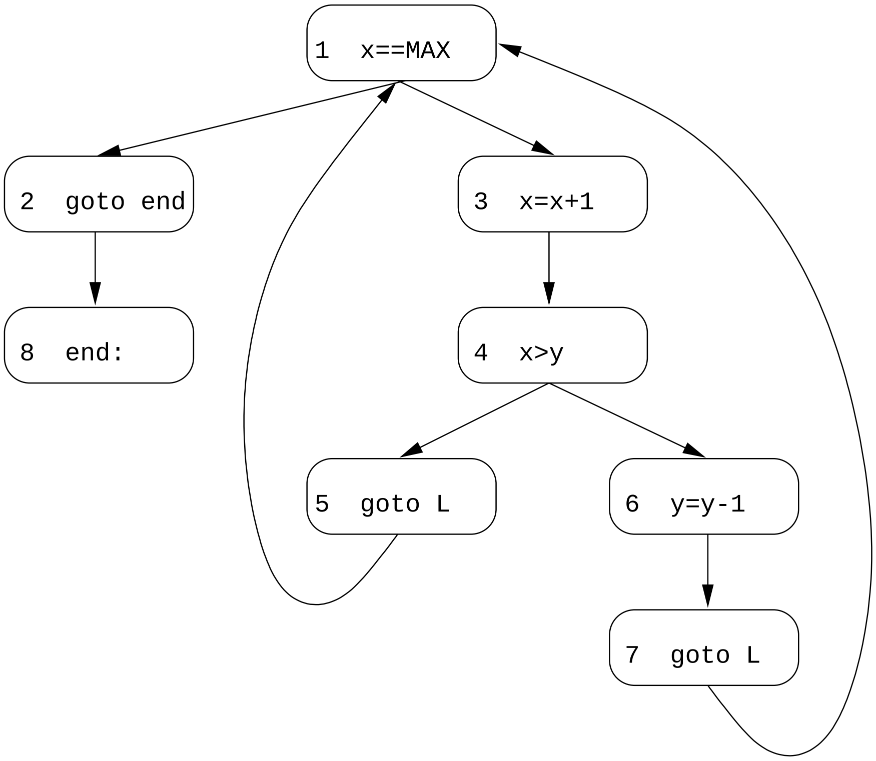

Figure 6: `goto` as a `continue` Statement

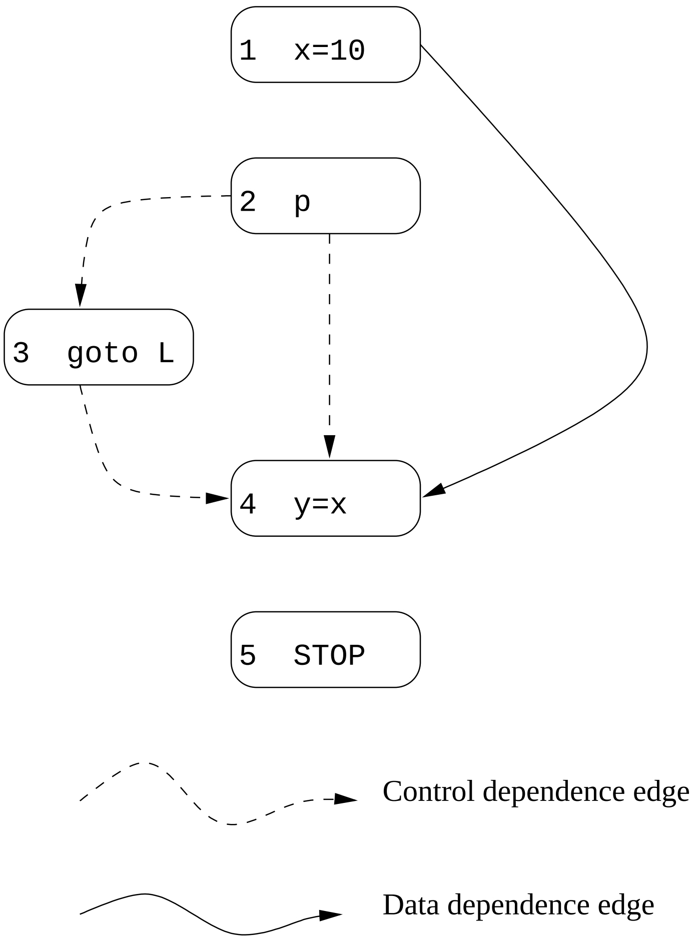

Figure 7: The PDG Constructed from the ACFG in Figure 5

Γ = the set of `goto` nodes in the original program;
Slice = the set of nodes in the PDG slice;
Org = the CFG of the original program;

do
begin
  Traverse the postdominator tree using *preorder* traversal, and for each jump statement, J, encountered that is not in *Slice* and for which Target(J) $\neq$ ILS(J) in *Slice*.
  if all nodes which control J are in $Slice \cup \Gamma$
    then
    Add J to *Slice*;
    else (J is controlled by a predicate p not in *Slice*)
    for each predicate p which controls J and is not in *Slice*
    if ILS(p) = IPD(p) in $Org \rhd (Slice \cup \Gamma \cup \{p\})$
      then
      take no action (i.e. include neither J nor p)
      else
      if[a] there exists a cycle free path π, from the consequent of p to IPD(p) in $Org \rhd (Slice \cup \Gamma \cup \{p\})$
        then
        begin
        Include the consequent of p in *Slice*.
        Include the `goto` statements on π in *Slice*.
        If p has an alternative branch, do not include it.
        end

        else
        (there must be a cycle free path π, from the alternative node of p to IPD(p) in $Org \rhd (Slice \cup \Gamma \cup \{p\})$)
        begin
        Include the alternative node of p in *Slice*.
        Include the `goto` statements on π in *Slice*.
        end
end

Remove from *Slice* any `goto` nodes which target their immediate lexical successor.
For each `goto` statement, *Goto L*, in *Slice*, if the statement labelled L is not in *Slice* then associate the label L with its nearest postdominator in *Slice*;
return (*Slice*);

[a] If there is a cycle free path from *both* the consequent *and* the alternative branch of p, then clearly there is a choice as to which to use. However, if path x is already contained in *Slice* but path y is not, then path x should be chosen in preference to path y.

Figure 8: The New Algorithm for Slicing Unstructured Programs

Figure 9: The Two Cases (edges denote paths containing zero or more `goto` nodes)

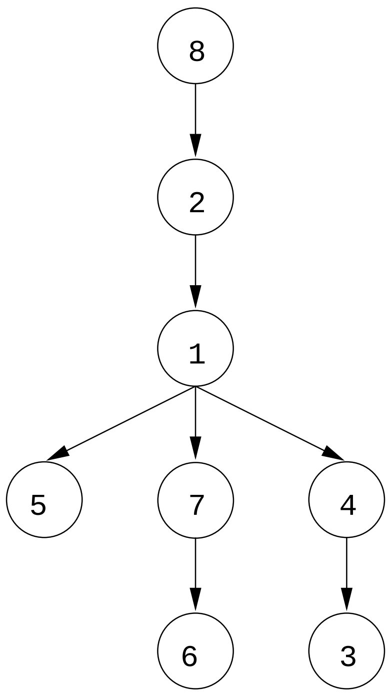

Figure 10: The post dominator tree for the program in Figure 6

1	`if (p)`	1	`if (p)`
2	`{ x=x+1;`	2	`{ x=x+1;`
3	`y=10;`		
4	`goto L; }`	4	`goto L; }`
5	`x=x+2;`	5	`x=x+2;`
6	`goto s2;`	6	`goto s2;`
7	`s0: if (y>0)`		
8	`{ L: y=y-1;`		`L:`
9	`goto s0; }`		
10	`else goto s3;`	10	`goto s3;`
11	`s2: x=x+1;`	11	`s2: x=x+1;`
12	`goto s4;`	12	`goto s4;`
13	`s3: x=x+2;`	13	`s3: x=x+2;`
14	`x=x+5;`	14	`x=x+5;`
Original Program		Slice on $(\{x\}, 15)$	

Figure 11: A More Complex Example

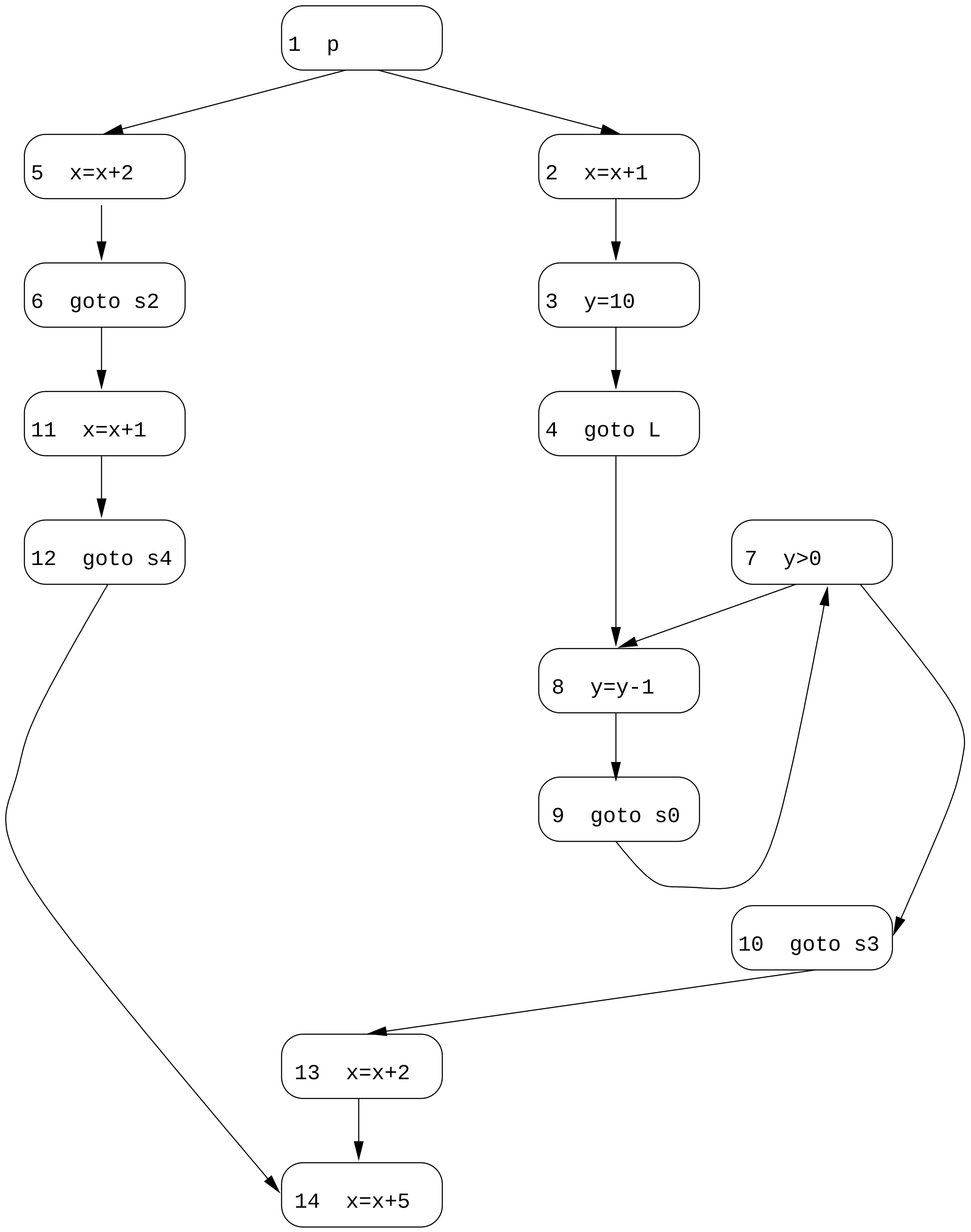

Figure 12: CFG for the program fragment in Figure 11

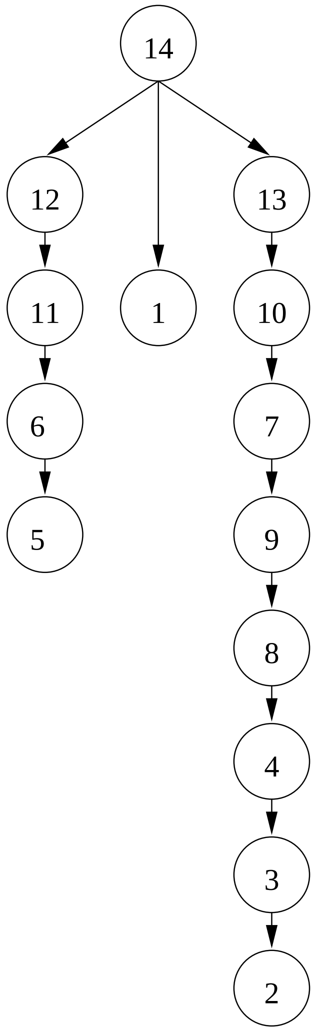

Figure 13: Post Dominator Tree for the program fragment in Figure 11