Chopping: A Generalization of Slicing

Daniel Jackson and Eugene J. Rollins
July 1994
CMU-CS-94-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A new method for extracting partial representations of a program is described. Given two sets of variable instances, *source* and *sink*, a graph is constructed showing the statements that cause definitions of *source* to affect uses of *sink*. This criterion can express a wider range of queries than the various forms of slice criteria, which it subsumes as special cases. On the standard slice criterion (backward slicing from a use or definition) it produces better results than existing algorithms.

The method is modular. By treating all statements abstractly as def-use relations, it can present a procedure call as a simple statement, so that it appears in the graph as a single node whose role may be understood without looking beyond the context of the call.

This report is a copy of a paper entitled "Abstract Program Dependences for Reverse Engineering" submitted to *SIGSOFT'94: Foundations of Software Engineering*.

This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of ARPA, NSF, TRW or the United States Government.

Keywords

Slicing, program dependence graph, dataflow dependence, specification, reverse engineering, program understanding, abstraction.

1 Introduction

No task that involves existing code – whether debugging, adaptation or restructuring – can begin until the developer understands it. Knowing all the details of how and why the code works is almost never necessary; an experienced developer will try to extract only just enough information to perform the task at hand.

Many of the questions that arise when a developer faces unfamiliar code are partial in two respects. First, they concern the *relationships* between program components rather than their values. That a procedure modifies a global counter, for instance, is likely to matter more than whether it increments or decrements it; and, if values do matter, they are usually determined easily once the relationships are clear. Second, questions tend to *focus* on some components and ignore others. A developer might want to see, for instance, only the statements that read or write some variable, or those that cause one variable to acquire its value.

Program slicing is a focusing technique based on dependence relationships that seems well suited to reverse engineering. Very roughly, a slice of a program is a skeleton obtained by deleting all statements that do not affect the value of a given variable at a given line [Wei84]. Slicing was originally devised for debugging, where its utility is easily seen: if the value of that variable at that line is wrong, the code that corrupted it (but not necessarily the bug [WL86]) must be within the slice, which may be much smaller than the original program.

The utility of slicing for reverse engineering, however, is less obvious. Not all questions – even when restricted to the vocabulary of program dependences – can be cast as slice criteria. How a variable affects other variables, for example, is a different question from how it is affected by others, and calls for a different analysis (sometimes referred to as "forward slicing" in contrast to standard "backward" slicing).

Furthermore, when applied to reverse engineering, slicing can give disappointing results. First, slices often turn out to be too large to be useful. Sometimes this is due to the limitations of static analysis and a smaller slice exists even though it cannot be found. But more commonly, the slice's focus is too broad. The user may want to understand only how a procedure's result is obtained from its arguments and not from globals, say, or vice versa. Slicing cannot discriminate origins, so every statement that affects the given variable will be drawn in, whatever the source of its dependence.

Second, procedure call is troublesome. Programs are easier to understand when procedures are examined one at a time. Interprocedural slicing [HDC88, HRB90, Bin93] treats procedure call as a linkage mechanism rather than an abstraction barrier, and includes statements in the slice from within the called procedure. As a result, to understand why a procedure call appears in a slice, the user must look inside to see which of its statements were responsible. Worse, the natural slice criteria associated with a procedure call are not expressible. One cannot, for example, easily specify a slice on the use of a global x by the call since the use of x will occur at a node inside the procedure (which the user will have to find) and not at the calling node.

We have developed an analysis similar to slicing that aims to overcome these problems. In place of the slice criterion, the user specifies two sets of variable instances, *source* and *sink*; the analysis then identifies the statements that cause *source* to affect *sink*. This allows a wider variety of questions to be formulated, of which the various kinds of slicing emerge as special cases.

Our analysis is based on a variant of the program dependence graph that treats nodes in a

```
program Sum
  s = 0
  x = 1
  while x < 11 do
    s = s + x
    x = x + 1
    end
  end
```

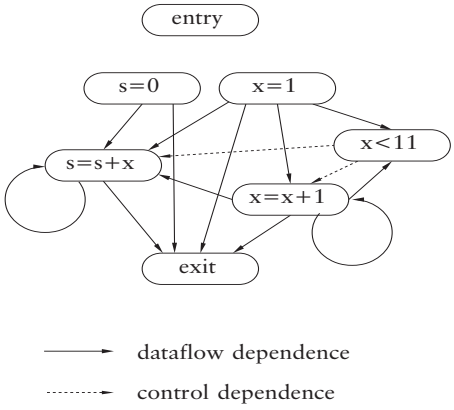


Figure 1: A program and its PDG

more abstract and uniform fashion. Instead of having a node for every simple assignment, and special nodes for procedure call and return, etc., we model each node as a def-use relation. This allows a modular analysis in which a procedure call appears as a statement like any other, but with a role determined by the dependences of its body.

Following the deconstructionist tendency of this line of research, we call our analysis *chopping* and have implemented it in a tool called Chopshop. The paper starts by explaining the foundations of the tool in the abstract program dependence graph and how it differs from the conventional dependence graph. It then shows how the relations of this abstract graph are used to chop a procedure and to compute abstractions of procedure calls.

2 *The Program Dependence Graph: Relating Statements*

The program dependence graph (PDG) is a popular representation of code that is well suited to slicing and a variety of other program manipulations. Its nodes are like those of a flowchart: one for each primitive statement (such as an assignment), one for each predicate (at the head of a loop or if-statement), and special entry and exit nodes for the program as a whole. Its edges, however, do not express control flow. Instead, an edge connects one node to another when the execution of the second is affected directly by the execution of the first.

A simple program (taken from [RY89]) with its PDG is shown in Figure 1. A flowchart for this program would show an edge from $s = 0$ to $x = 1$ because of their syntactic order, but, since their executions are independent, there is no edge connecting them in the PDG.

There are two kinds of edge in the PDG. The solid edges are dataflow dependences; an edge from i to j indicates that some variable is used at j that, on some path through the control-flow graph, was last defined at i. The edges from $x = 1$ to $x = x + 1$ and from $x = x + 1$ to $s = s + x$, for example, are both due to the variable x; the latter is said to be "loop carried", since the connecting path goes round the loop.

The dotted edges are control dependences; an edge links a predicate node i to a node j

when the evaluation of the predicate at i immediately controls execution of the node j. Note that both statements of the loop's body are control dependent on its predicate.

Entry and exit nodes may be treated in a number of ways. We assume that all variables are defined on entry and used on exit. This extends nicely to the PDG of a called procedure, in which the entry node stands for the prior definitions of variables, and the exit node for their subsequent uses. It also seems more natural than the conventional treatment in which the entry node is viewed as a predicate on which other nodes have control dependences [FOW87, RY89].

Slices are easily (but, as we shall see, not accurately) calculated from the PDG. To slice the program on some variable defined or used at a node, one simply walks back over the PDG, marking all the nodes on the way, and then deletes from the program text the statements corresponding to unmarked nodes [OO84]. Weiser's original algorithm [Wei84, LR87] is more general, since it allows slicing on variables not used or defined at the given line, but also less efficient, since it recomputes the dependences for each new slice.

With suitable elaborations (such as def-order edges and true/false labelling of control-dependence edges), the PDG may be regarded as a complete representation of the program, so that two programs with isomorphic PDG's must behave equivalently [HPR88, CF89]. For our purposes, these elaborations are not relevant.

Finally, a note on how the PDG is constructed. The first step is to obtain, for each node of the control-flow graph, a set of reaching definitions [ASU88]: a definition of variable x at node i reaches a node j if there is a path from i to j with no intervening definition of x. For each reaching definition at i of a variable that is actually used at j, a dataflow dependence edge from i to j is inserted. The control dependence edges are a little trickier, requiring the calculation of a post-dominator tree [FOW87].

3 *The Abstract PDG: Relating Variable Instances*

Suppose one of the statements in our program is a call to a procedure whose internal structure is of no interest. We would still like to construct a PDG for the program – in general, itself a procedure – that takes account of the behaviour of the called procedure without explicitly including its statements as a subgraph.

In the construction of the standard PDG, the only details of a statement that matter are the sets of variables that it uses and defines. Usually, only one variable is defined in a primitive statement, and a dataflow edge leaving the corresponding node is clearly due to that variable. For example, the statement

$x = y + z$

has a definition set of $\{x\}$ and a use set of $\{y, z\}$; if there is a dataflow edge from this node to another node, it must be because that node uses x.

A naive extension to handle procedure call would simply calculate definition and use sets for the body of the procedure; these would then (with appropriate renaming of formals to actuals) play the role of definition and use sets for the call, treated as a primitive statement. The *add* procedure in Figure 2, for instance, would be given a definition set of $\{s\}$ and a use set of $\{s, i\}$; after renaming i to x, the call $add(x)$ would have definitions $\{s\}$ and uses $\{s, x\}$, just like the assignment $s = s + x$.

The resulting PDG (to the right in Figure 2) is as before, with $add(x)$ replacing $s = s + x$.

```
program Sum1
   procedure add (i)
      s = s + i
      end
   s = 0
   x = 1
   while x < 11 do
      add (x)
      x = x + 1
      end
   end
```

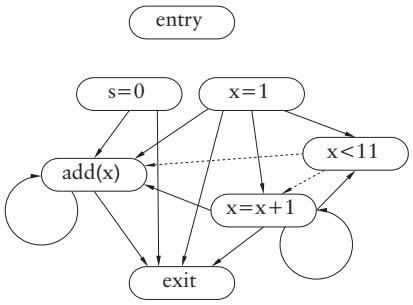


Figure 2: Case in which naive PDG is accurate

```
program Sum2
   procedure add
      s = s + x
      x = x + 1
      end
   s = 0
   x = 1
   while x < 11 do
      add ()
      end
   end
```

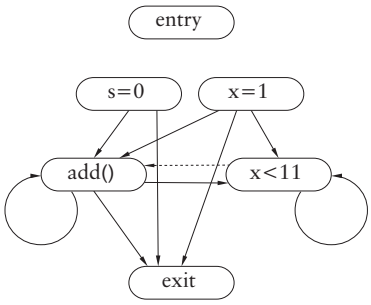


Figure 3: Case in which naive PDG is innaccurate

```
program Sum2
   procedure add
41    s = s + x
42    x = x + 1
      end
1  s = 0
2  x = 1
3  while x < 11 do
4     add ()
      end
   end
```

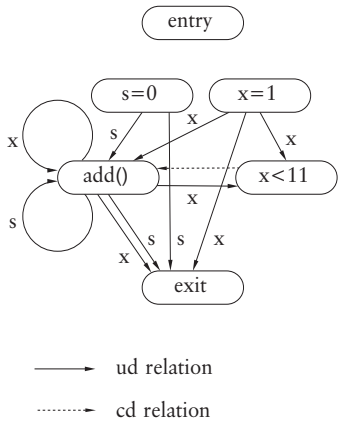


Figure 4: Abstract PDG of body of Sum2

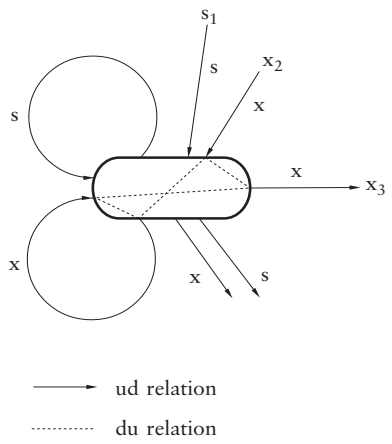


Figure 5: Tracing back a use-def edge through add's def-uses: use of x_3 comes from def of x_2 but not def of s_1

Moreover, any slices of the main program calculated by the standard algorithm will be exactly as if we first inlined the procedure, calculated the slice, and then collapsed the body back to a call.

In general, though, this approach will produce poor results, giving slices that are far larger than necessary. To see why, consider another variant of *Sum* in which both statements of the loop body are moved to a procedure (Figure 3). The resulting PDG correctly relates the statements, there being an edge to or from the procedure call exactly when there was an edge in the original PDG to or from one of its statements. But this time the loss of structure in the procedure call affects the calculation of slices. If we slice on the use of x in the predicate $x<11$, we will obtain all the statements of the original program, including the irrelevant $s = 0$. Treating the call to *add* as a single node spuriously associates the definition of x with the use of s, effectively merging paths through the body of the procedure that should be disjoint.

To avoid this, we must maintain the association between definitions and uses. Every node in the PDG is given, in place of a definition set and a use set, a binary relation on variables that contains the pair (u, v) when u is defined using v. Furthermore, each dataflow dependence edge is labelled with the variable defined at its source.

The *add* procedure of *Sum2* would be given the def-use relation

$\{(s, x), (s, s), (x, x)\}$

from which it is clear that a definition of x cannot come from a use of s. The edge from $s = 0$ to *add*() is labelled s and the edge from $x = 1$ is labelled x (Figure 4).

A new slicing algorithm will now succeed in excluding $s = 0$ from the slice of x at $x<11$. Instead of simply tracing backwards along edges between statement nodes, it follows the uses and definitions of variables. It will determine from the def-use relation of *add*() that the definition of x is due to a use of x alone, and will find the node's predecessors by following only edges marked x. Figure 5 shows the relevant def-use associations of *add*() and how they are connected to the use-def edges; in particular, the outgoing x edges are not connected to the incoming s edges.

The conventional PDG relates statements with edges marked only to distinguish control and dataflow dependences, so it can be modelled as two binary relations on statements. The argument of the last section suggests that, at least for dataflow dependences, we shall need to label each edge with the variable responsible for the dependence.

For uniformity, it is convenient to model all the forms of dependence as relations over the same set. A variable instance is a pair consisting of a variable and a site:

$Instance = Var \times Site$

A site is just a node in the graph, but it also corresponds to a point in the program text. An instance is not the same as a syntactic occurrence, since each primitive statement or procedure call occupies one site, and can have only one instance per variable, however many times the variable appears. We shall write x_i for the instance of variable x at site i. The statement

41 $s = s + x$

for example, would have a definition of s_{41} and uses of s_{41} and x_{41}. The purpose of this scheme is simply to label variables with their sites; it involves no precomputation of dependences (in contrast to single static assignment form [C+91], in which variables are labelled so that uses match their corresponding definitions).

The abstract PDG is modelled as three relations on instances:

$du, ud, cd: Instance \leftrightarrow Instance$

The *du* relation holds the def-use associations of the individual statements. It contains the pair (x_i, y_i) when x is a variable defined at site i by a use of the variable y. If a variable is defined by the use of no variable, it cannot be omitted from the *du* relation, for otherwise it will appear not to be defined at all. So we introduce a dummy variable $\perp$, and a statement such as

2 $x = 1$

will contribute $(x_2, \perp_2)$ to the *du* relation. Whenever a site defines a variable, we add a dependence on another special variable γ; the pair (x_i, γ_i) indicates that the variable x is defined at i (for reasons that will soon be clear). Statement 41 above would thus contribute the pairs

(s_{41}, s_{41}), (s_{41}, x_{41}) and (s_{41}, γ_{41}).

$\perp$ and γ are the only special variables, so

$Var = ProgramVariables \cup \{\perp, \gamma\}$

The *ud* relation models the dataflow dependences between statements; it corresponds to the dataflow dependence relation of the standard PDG. Its pairs are of the form (x_i, x_j), where x is a variable used at site i and defined at site j.

Lastly, the *cd* relation models control dependences. It contains the pair (γ_i, x_j) when site i has a control dependence on site j, and site j determines the flow of control by testing variable x. In contrast to the standard PDG, we shall not distinguish predicate nodes and statement nodes. A node may both have side effects and influence the flow of control. This uniformity simplifies the association between the graph and the program text for languages like C in which side-effecting conditionals are frequent. The fragment

1 *if* $(x{+}{+})$
2 $y = z$

for example, would not require two separate nodes for the conditional expression. The value

of the expression tested is assigned to a temporary variable, e say, so that statement 1 contributes to the *du* relation

(e_1, x_1), (x_1, x_1), (e_1, y_1) and (x_1, y_1)

and statement 2 contributes

(y_2, z_2) and (y_2, y_2).

The control dependence of statement 2 is then expressed by the pair

(y_2, e_1)

so that in the composite if-statement there will be a transitive dependence of the final value of y_2 on the initial value of x_1: y_2 on y_2 from statement 2's contribution to *du*, y_2 on e_1 from its contribution to *cd*, and e_1 on x_1 from statement 1's contribution to *du*.

Like the standard PDG, the abstract PDG has special entry and exit sites. All variables are defined at the entry and used at the exit:

$entry, exit : Site$
$Var \times \{entry\} \subseteq dom\ du$
$Var \times \{exit\} \subseteq ran\ du$

The *ud* and *cd* relations for the program *Sum2* are shown in Figure 4. The *du* relation is not easy to display graphically, but is shown in part for the *add* call in Figure 5. There is no labelling of *cd* edges; this would convey no extra information (and for a compound predicate, as here, would show control dependence on a temporary variable).

The labelling of *ud* edges is implicit in the conventional PDG, since it can be inferred from the plain edges and the definition and use sets of the nodes. The novelty of the abstract PDG is the *du* relation, which, having yielded the definition and use sets of the nodes, plays no role in the construction of the graph. The role of *du* becomes central, however, when the graph is used. The next section explains some useful closure relations based on *ud*, *cd* and *du*; subsequent sections show how they are used.

4 *Closures of the Abstract PDG*

Instead of giving explicit graph traversal algorithms, we shall formulate our analysis of the abstract PDG with the relational operators of Z [Spi89], summarized in Appendix 1. This has several advantages. First, the exposition is terser and, we hope, easier to understand. Second, it suggests an efficient implementation in which the closure relations are calculated only once (being derivable from the abstract PDG), and a variety of partial representations is then produced by restricting their domain and range. Third, the relational expressions are less biased than explicit traversal algorithms, and might be used to justify other implementation strategies.

The closure relations express directly the transitive dependence, due to a path through the graph, of a variable at one site on a variable at another site. Since the difference between a control dependence and a dataflow dependence cannot be observed at the endpoints of a path, we start by merging the *ud* and *cd* relations into a single relation *ucd* expressing all edges between nodes:

$ucd = ud \cup cd$

Four closure relations arise naturally. The first, *UD*, associates a use of some instance with a

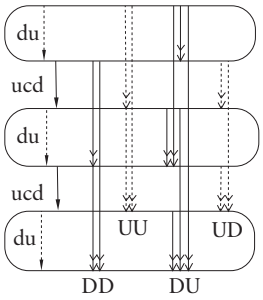


Figure 6: How basic and closure edges are related

definition of another instance, and can be thought of as an abstraction of a subgraph into a set of edges. An instance x_i is related by *UD* to an instance y_j if there is a definition of y at site j that might affect the value of x where it is used at site i. This might come about because of a direct use-def edge (*ucd*), or because of a path with two edges and a mediating node (*ucd* ∘ *du* ∘ *ucd*), or a path of three edges (*ucd* ∘ *du* ∘ *ucd* ∘ *du* ∘ *ucd*), and so on, giving

$UD = ucd \circ (du \circ ucd)^*$

The *UD* relation for *Sum2* (Figure 4), for example, would include the pair (x_3, s_1), indicating that the use of x in the predicate $x<11$ may be affected by the definition of s in $s = 0$.

The second relation, *DU*, abstracts a subgraph into a set of nodes. An instance x_i is related by *DU* to an instance y_j if the definition of x at i can be affected by the value of y when used at j. Again, this might come about directly or along a path, which this time starts and ends with a node rather than an edge:

$DU = du \circ (ucd \circ du)^*$

The *DU* relation for *Sum2* would include the pair (s_4, x_2), since the definition of s in *add*() may be affected by the value of x used in $x = 1$.

The third relation, *UU*, relates uses to uses, and is defined in terms of an even number of hops:

$UU = (ucd \circ du)^*$

Finally, *DD* relates definitions to definitions:

$DD = (du \circ ucd)^*$

The *UU* relation for *Sum2* (Figure 4) would contain (x_4, x_3) since the use of x by *add*() depends on the use by $x<11$; the *DD* relation would include (s_4, x_2) since the definition of s by *add*() depends on the definition of x by $x = 1$. Both of these closures are symmetric, and include pairs like (x_4, x_4).

The relationship between the basic relations and the closures is illustrated in Figure 6.

Chopping is a focusing mechanism like slicing. The user selects two sets of variable instances, *source* and *sink*; chopping then yields the subgraph that shows how the definitions of the instances in *source* can affect the uses of the instances in *sink*.

The subgraph is defined as a subset of the use-def relation *ucd*:

$ucd' = UU(sink) \lhd ucd \rhd DD^{\sim}(source)$

$DD^{\sim}(source)$, the image of *source* under the inverse of *DD*, is the set of definitions that might be affected by the definitions in *source*. Similarly, *UU(sink)*, the image of the set *sink* under *UU*, is the set of uses that the uses in *sink* depend on. The relevant use-def edges are those that connect a use in *UU(sink)* to a definition in $DD^{\sim}(source)$. These are obtained by restricting the domain of *ucd* to the former and its range to the latter.

To distinguish dataflow and control-flow dependences in the subgraph, it may be computed in two parts which are then superimposed; first the dataflow edges

$ud' = UU(sink) \lhd ud \rhd DD^{\sim}(source)$

are displayed as solid lines, and then the control-flow edges

$cd' = UU(sink) \lhd cd \rhd DD^{\sim}(source)$

which are shown dotted. The relevant edges of the def-use relation may be found easily too:

$du' = DD^{\sim}(source) \lhd du \rhd UU(sink)$

and perhaps displayed as in Figure 5 (although our tool does not currently do this).

The various forms of slice criteria may be expressed as special cases of chopping. Reps's criterion [RY89] identifies all the sites that contribute to the use or definition of some variable v at site i. To account for a use we include $\{v_i\}$ in the sink set, and to account for a definition we include $du(\{v_i\})$. The equivalent chopping criterion is thus

$source = Var \times Site,\ \ sink = \{v_i\} \cup\ du\ (\{v_i\}).$

Weiser's criterion [Wei84] identifies all sites that contribute to the values of the variables in a set V just before execution of a statement at some site i. So long as the variables V are used at site i, the criterion may be expressed as

$source = Var \times Site,\ \ sink = V \times \{i\}.$

Forward slicing [YL88] marks the sites that are subsequently affected by a definition of a variable at some site. For a variable v defined at site i, the criterion is

$source = \{v_i\},\ \ sink = Var \times Site.$

Finally, a variant of slicing proposed for use in maintenance [GL91] identifies all the sites that affect the final value of a variable, or a definition at any site. For a variable v, this requires

$sink = du(\{v\} \times Site) \cup \{v_{exit}\},\ \ source = Var \times Site.$

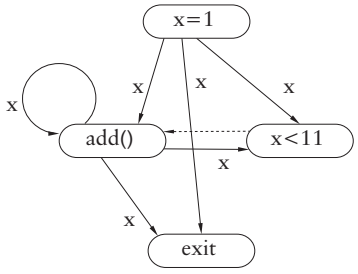
Examples. Figure 7 shows the results of applying four different criteria to the program *Sum2*. To see how the final value of x is determined, we slice on its final use:

$source = Var \times Site, sink = \{x_{exit}\}$

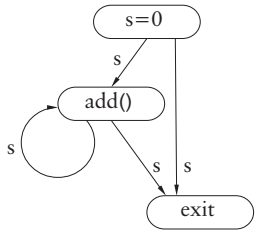
The result (7a) shows not only which sites are included but also why, through the labelling of edges. The absence of edges marked *s*, for instance, shows that *s* is not relevant to the computation of x (even though *add*() uses and defines *s*).

```
program Sum2
      procedure add
41              s = s + x
42              x = x + 1
                end
1     s = 0
2     x = 1
3     while x < 11 do
4               add ()
                end
      end
```

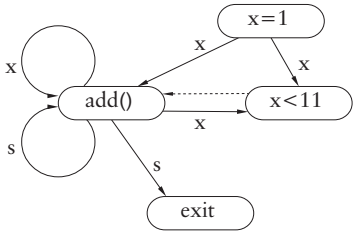
The program



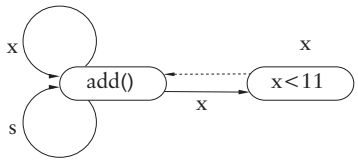
a: Backward slice on final x



b: Forward slice on initial s



c: Effect of initial x *on final* s



d: Role of add()

Figures 7a–d: Chopping Sum2

To see the effect of the initialization of s, we forward slice:

$source = \{s_1\}$, $sink = Var \times Site$

The result (7b) shows *add*() again, but in a different role; this time its use and definition of x are irrelevant. The loop predicate has gone too, since the initial value of s does not affect the number of iterations.

Slicing is not always focused enough. If we want to know how the initial value of x affects the final value of s, we would select

$source = \{x_{entry}\}$, $sink = \{s_{exit}\}$

which gives a subgraph (7c) that is smaller than the slice

$source = Var \times Site$, $sink = \{s_{exit}\}$.

Finally, another example of a query that cannot be cast as a slice criterion: to understand how *add*() behaves in the loop, we might select

$source = Var \times \{4\}$, $sink = Var \times \{4\}$

to show the flows that originate in the definitions of one call to *add* and end in the uses of another (7d).

7 *More Precise Slice Criteria and Smaller Slices*

The discovery that slicing reduces to a simple reachability problem in the PDG [OO84] has made both slicing and the PDG more popular. But it has also led to a reformulation of the slice criterion, weakening its precision considerably. The abstract PDG supports both a precise slice criterion and a simple construction method.

Reps and his colleagues have formalized the relationship between slicing and the PDG [RY89]. They define the slice of a program with respect to a program point p and variable v to be all the statements and predicates that might affect the value of v at point p. Like us, they do not permit a slice to be taken with respect to an arbitrary variable, but require that v is used or defined at p. The slice is constructed by tracing backwards in the PDG from the node corresponding to p. Not surprisingly, the variable v plays no role in this; the conventional PDG, unlike ours, just relates statements. The slice criterion is thus really a node in the PDG, and, indeed, is formalized in exactly this way.

As a result, for a given slice criterion, chopping will often produce better results. In the program *Sum* (Figure 1), for example, a slice on the use of x in the statement $s = s + x$ would spuriously include $s = 0$, which our method correctly eliminates. The PDG algorithm traces back along all the edges from the node at which the variable is used, even those due to uses of different variables. This inaccuracy becomes disastrous if many variables are used at one node, which is why Reps is forced to provide a separate exit node for each program variable. Otherwise, slicing on a final use of any variable would yield the entire program.

Chopping not only gives a more accurate slice, but also allows a more precise criterion. Suppose we want to see the code that affects the value of s coming into statement 4, $s = s + x$. The PDG formulation of slicing does not distinguish uses and definitions, so a slice on s would include the code that determines x too. Our method does not suffer from this problem; the use of s is expressed by

$source = Var \times Site$, $sink = \{s_4\}$

and the definition by

$source = Var \times Site$, $sink = du(\{s_4\})$.

Recall, finally, that chopping – unlike slicing – is modular. It should make sense to slice on a variable at a procedure call if the procedure uses that variable. Interprocedural slicing techniques [HDC88, HRB90, Bin93], however, associate with the call node only the uses of actual parameters. If the variable is a global, the user must find the first use in the procedure body and slice on that. Our abstract PDG gives the procedure call the expected dependences, so that the call node uses a variable when there is a use in the body. We can slice, for example on the use of x in the call *add*() of *Sum2* (Figure 4) by specifying a sink of $\{x_4\}$; interprocedural slicing would require that we identify instead $\{x_{41}\}$, the first use in the procedure body.

8 *Forming a DU Abstraction of a Procedure*

The abstract PDG models a procedure call at a site i as a subset of the du relation consisting of a set of pairs of the form (x_i, y_i), where x is defined (due to the execution of the procedure) by a use of y. We can think of the relation on variables

$Deps \subseteq Var \times Var$

as a kind of dependency specification of the procedure, which we now show how to construct from the abstract PDG of its body.

All variables are used at the exit and defined at the entry. Our task is to determine the def-use relation of the subgraph that lies between these nodes.

First we calculate the def-use associations between variable instances that span the subgraph by restricting the *DU* relation. Its domain is restricted to the instances whose definitions reach the exit node, and its range is restricted to the instances that are reached by definitions of the entry node:

$DefUses = ud(Var \times \{exit\}) \lhd DU \rhd ud^{\sim}(Var \times \{entry\})$

This does not include every relevant definition, however, because a variable set to a constant will have no association to a use reached by the entry node. The instances that represent definitions that reach the exit, excluding the dummy definitions at the entry, are

$Defs = dom\ (ud(Var \times \{exit\}) \lhd DU) \setminus (Var \times \{entry\})$.

These definitions may be ambiguous [ASU88], since there may be paths from entry to exit on which they do not occur. To distinguish ambiguous definitions, we collect the set of instances that might be invariant, whose uses at the exit are reached by their definitions at the entry:

$Invs = dom\ ((Var \times \{exit\}) \lhd UD \rhd (Var \times \{entry\}))$

From the relation *DefUses* and the sets *Defs* and *Invs* we can now determine the dependency specification. Since we do not care which sites are associated with the definitions and uses, we project onto variables:

$DefUses_v = \{(u, v) \mid \exists i, j.\ ((u, i), (v, j)) \in DefUses\}$
$Defs_v = dom\ Defs$
$Invs_v = dom\ Invs$

The variables that are defined but not associated with uses must have been reset to constants:

```
    procedure add
41  s = s + x
42  x = x + 1
    end
```

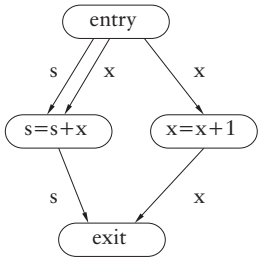


Figure 8: Abstract PDG for add *procedure*

$Resets_v = Defs_v \setminus dom\ DefUses_v$

The ambiguously defined variables are those that are both potentially invariant and potentially defined:

$Maybes_v = Invs_v \cap Defs_v$

Finally, the dependency relation consists of the def-use associations, dependences of resets on the constant symbol, and self-dependences of ambiguously defined variables:

$Deps = DefUses_v \cup Resets_v \times \{\bot\} \cup \{(m,m) \mid m \in Maybes_v\}$

Example. The *add* procedure of Figure 4 has the abstract PDG shown in Figure 8. Note that, being a procedure and not a stand-alone program, it has edges from the entry node representing flows from the calling context into the procedure body. The *DU* closure is in this case identical to *du*, since no paths from a use to a definition span more than one node. The restrictions reduce it to

$DefUses = \{(s_{41}, s_{41}), (s_{41}, x_{41}), (x_{42}, x_{41})\}$.

and the resulting dependence relation is

$Deps = \{(s, s), (s, x), (x, x)\}$.

Were $x = x + 1$ to be replaced by $x = 1$, we would have

$DefUses = \{(s_{41}, s_{41}), (s_{41}, x_{41})\}$

and

$Defs = \{s_{41}, x_{42}\}$

giving

$Resets_v = \{x\}$

and a dependence relation of

$Deps = \{(s, s), (s, x), (x, \bot)\}$.

A dependency relation is constructed, bottom-up, for each procedure in the program. To find the contribution of a procedure call to the *du* relation at some site *i*, the variables of the called procedure's dependence relation are renamed to match formals to actuals; they are

instantiated at i; and, as explained in Section 4, related in addition to the special variable y as a hook for control dependences.

9 *The Chopshop Tool*

We have implemented the abstract dependence graph and chopping algorithm for C programs. The tool is written in Standard ML of New Jersey. It compiles the C code into an intermediate representation from which a conventional PDG is constructed, and then builds the basic and closure relations of the abstract PDG. The def-use contributions of called procedures are computed when required and cached. The tool is connected by a Unix pipe to *emacs*, so that the mouseclicks can be used to select variables and sites. The result of chopping, a graph, is output in adjacency-list form, which is converted to a postscript picture by *dotty* and previewed with *ghostscript*. We have experimented for the last half-year on small programs with no aliasing, but we hope to extend the tool so that it can handle industrial C code by the end of the year. Performance will be a critical issue, but we see no reason that the abstract PDG should be fundamentally less tractable than the standard PDG. Although the closure relations are potentially enormous (varying with the squares of the numbers of sites and variables), they are very sparse in practice, since only a few variables are typically defined or used at a given site.

10 *Related Work*

Tools that analyze relationships between program components have been around for some time. CScope [Ste85] and CIA [CNR90], for example, derive relationships from the abstract syntax tree, such as which procedures call which, and respond to queries in a relational database language. Refine [M+94, Rea92] can present this kind of information in a variety of diagram styles. None of these tools do any serious dataflow analysis. Refine/C, for example, can show when one procedure uses a variable defined by another, but considers a variable to be defined by a procedure whenever it occurs on the left-hand side of any statement in the procedure body.

To trim derived representations of large programs, various mechanisms have been devised. CodeBase [Sel91] organizes CIA output with syntactic rules expressing notions from the application domain. Rigi [MK88, M+92] provides various aggregation and generalization mechanisms. In both cases, the abstractions are syntactic and specified manually. In the "interface slicing" of [BE93], the user selects some subset of the operations of a module, and the tool uses the call graph to eliminate operations that are dead code.

Only a few slicing tools have been built for commercial use, and most of these are for Cobol. Andersen Consulting's Cobol/SRE [NEK93, NEK94] is probably the most complete: it does PDG-based forward and backward slicing, as well as "condition-based slicing" which is really not slicing at all, but rather a kind of partial evaluation. Procedures are rarely used in Cobol, so interprocedural analyses are unnecessary; perform statements are handled by copying the subgraph of the performed paragraph.

Weiser's scheme for interprocedural slicing [Wei84] gives poor results because it fails to distinguish call sites, so that a call at one site appears to be able to return to another. A more accurate analysis based on the PDG has been invented since [HRB90, Bin93]. The essential

idea is to construct a PDG for the whole system with special nodes for procedure call and return, along with an attribute grammar to keep track of the mapping of formals to actuals. The resulting slices are good for many applications but less appropriate for reverse engineering. Unlike our approach, the slices are not *modular*; a procedure call is a link to another part of the graph rather than an abstraction barrier, so that to understand why a procedure call is included in a slice, the user must follow the dependency associations into the procedure body. Chopshop, by labelling the *ud* edges in and out of a procedure call, and by providing its *du* abstraction on request, allows it to be understood abstractly in context. The user can, of course, follow inside the body too, by chopping the body on the variables entering and leaving the call.

This technique, as well as another interprocedural slicing technique [HDC88], can handle recursive programs; we cannot yet do this. More seriously, we do not handle aliasing, a thorny problem in dataflow analysis. We intend to incorporate ideas from recent work such as [HS94] and especially [LR92, LRZ93, PLR94] which address the kinds of pointer aliasing that arise in C programs.

The *du* abstraction originates in the Aspect specifications of our previous work [Jac91, Jac92, Jac93]. Relational models of dependences have been used before, but this paper seems to be the first to separate the use-def relations of the PDG edges from the def-use relations of the nodes. The relations underlying the Spade tool [BC85] are the closest in spirit, since they associate variables and expressions at particular program points. They are also used to extract "partial statements" which seem to be identical to Weiser's slices. Like our previous work, the relations are defined compositionally over the syntax, prohibiting programs with arbitrary jumps. Moreover, a fundamental assumption that each variable occurrence is either a definition or a use prevents the handling of procedure calls with side effects.

Moriconi and Winkler's inference rule system for determining the scope of a program change [MW90] also defines a dependence relation inductively over the syntax. Procedure calls are abstracted, since the proof of a procedure call's dependence can be built from rules applied to its body. The proof of a dependence is a kind of explanation akin to the labelling of our *ud* edges. Apparently this system can support slicing too; it would be interesting to see if it could be extended to more general chopping criteria. Its main disadvantage is performance, since the inference system evaluates the dependence relation pair-by-pair.

Wilde and Huitt discuss "external dependency graphs", which are similar to our *du* abstraction, along with a variety of other dependence relations [WH91], but do not explain how they are constructed and used.

Theories of program dependences [CF89, PC90] and slicing [RY89] have been developed. We hope eventually to develop a theory that explains our dependence relations and justifies an operational interpretation of chopping.

11 *Conclusion and Future Work*

The standard PDG provides no way to abstract the behaviour of a procedure. We have shown that this requires enriching the notion of dependence to relate not just sites but variable/site pairs. Linking the definition of a variable at one site with a subsequent use at another is not enough; we must also make explicit the dependences of a site's outgoing definitions on its incoming uses – the *du* relation of our abstract PDG.

The abstract PDG supports chopping, a novel focusing mechanism that is both more flex-

ible than program slicing and more accurate, even on standard slice criteria. Chopping is modular, so that a procedure call can be treated as a simple statement, whose inclusion can be justified with the *du* relation and explained to the user by the labelling of edges.

Acknowledgments

David Ladd of AT&T Bell Labs wrote the first version of Chopshop's C parser.

Appendix 1: Relational Operators

The paper uses the following operators on sets and relations. Throughout, s and t are sets of elements of type T, and p and q are binary relations on T.

$s \setminus t = \{e{:}s \mid e \notin t\}$
$I = \{(t, t) \mid t{:}T\}$
$dom\ p = \{a{:}T \mid \exists b{:}T.\ (a, b) \in p\}$
$ran\ p = \{b{:}T \mid \exists a{:}T.\ (a, b) \in p\}$
$p^{\sim} = \{(b, a) \mid (a, b) \in p\}$
$p \circ q = \{(a, b) \mid \exists z{:}T.\ (a, z) \in p \wedge (z, b) \in q\}$
$s \lhd p = \{(a, b) \in p \mid a \in s\}$
$p \rhd s = \{(a, b) \in p \mid b \in s\}$
$p(s) = \{b \mid \exists a{:}s.\ (a, b) \in p\}$

The reflexive and transitive closure of p, written p^*, is the smallest relation containing p that is reflexive ($I \subseteq p^*$) and transitive ($p^* \circ p^* \subseteq p^*$). It may be calculated iteratively as:

$p^* = I \cup p \cup (p \circ p) \cup (p \circ p \circ p) \cup \ldots$

Appendix 2: Summary of Definitions

Definition of abstract PDG:

$Var = ProgramVariables \cup \{\bot, \gamma\}$
$Instance = Var \times Site$
$entry, exit: Site$
$du, ud, cd: Instance \leftrightarrow Instance$
$Var \times \{entry\} \subseteq dom\ du$
$Var \times \{exit\} \subseteq ran\ du$
$ucd = ud \cup cd$

Closure relations:

$UD = ucd \circ (du \circ ucd)^*$
$DU = du \circ (ucd \circ du)^*$
$UU = (ucd \circ du)^*$
$DD = (du \circ ucd)^*$

Chop from source to sink:
$ud' = UU(sink) \lhd ud \rhd DD^{\sim}(source)$
$cd' = UU(sink) \lhd cd \rhd DD^{\sim}(source)$
$du' = DD^{\sim}(source) \lhd du \rhd UU(sink)$

Forming DU abstraction of procedure

1. *Determine last defs and first uses*
   $DefUses = ud(Var \times \{exit\}) \lhd DU \rhd ud^{\sim}(Var \times \{entry\}$
2. *Determine instances potentially defined or invariant*:
   $Defs = dom\ (ud(Var \times \{exit\}) \lhd DU) \setminus (Var \times \{entry\})$
   $Invs = dom\ ((Var \times \{exit\}) \lhd UD \rhd (Var \times \{entry\}))$
3. *Project onto variables*
   $DefUses_v = \{(u, v) \mid \exists i, j.\ ((u, i), (v, j)) \in DefUses\}$
   $Defs_v = dom\ Defs$
   $Invs_v = dom\ Invs$
4. *Obtain dependence relation Deps*
   $Resets_v = Defs_v \setminus dom\ DefUses_v$
   $Maybes_v = Invs_v \cap Defs_v$
   $Deps = DefUses_v \cup Resets_v \times \{\bot\} \cup \{(m,m) \mid m \in Maybes_v\}$

References

[ASU88] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, March 1988.

[BC85] Jean-Francois Bergeretti and Bernard A. Carre. Information-flow and dataflow analysis of while-programs. *ACM Trans. on Programming Languages and Systems*, 7(1), January 1985, pp. 37–61.

[BE93] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. *Proc. 15th International Conference on Software Engineering*, Baltimore, May 1993, pp. 509–518.

[Bin93] David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, Vol. 2, Nos. 1–4, March–December 1993, pp. 31–45.

[C+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. Efficiently computing single static assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4), October 1991, pp. 451–490.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. *Proc. ACM Symposium on Programming Language Design and Implementation*, 1989.

[CNR90] Y.-F. Chen, M. Nishimoto and C.V. Ramamoorthy. The C information abstraction system. *IEEE Trans. on Software Engineering*, , March 1990.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3), July 1987, pp. 319–349.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. on Software Engineering*, 17(8), August 1991, pp. 751–761.

[HDC88] J.C. Hwang, M.W. Du and C.R. Chou. Finding slices for recursive procedures. *Proc. IEEE COMPSAC 88*, Chicago, October 1988.

[HPR88] S. Horwitz, J. Prins and T. Reps. On the adequacy of program dependence graphs representing programs. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, pp. 146–157.

[HRB90] Susan Horwitz, Thomas Reps and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1), January 1990, pp. 26–60.

[HS94] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. on Programming Languages and Systems*, 16(2), March 1994, pp. 175–204.

[Jac91] Daniel Jackson. Aspect: An Economical Bug Detector. *Proc. International Conf. Software Engineering*, Austin, Texas, May 1991, pp. 13–22.

[Jac92] Daniel Jackson. *Aspect: A Formal Specification Language For Detecting Bugs*. MIT/LCS/TR-543, MIT Laboratory for Computer Science, Cambridge, Mass. 02139, June 1992.

[Jac93] Daniel Jackson. Abstract Analysis with Aspect. *Proc. International Symposium on Software Testing and Analysis*, Cambridge, Mass., June 1993, pp. 19–27.

[LR87] Hareton K.N. Leung and Hassan K. Reghbati. Comments on program slicing. *IEEE Trans. on Software Engineering*, 13(12), December 1987.

[LR92] William Landi and Barbara Ryder. A safe approximation algorithm for interprocedural pointer aliasing. *Proc. Conf. Programming Language Design and Implementation*, 1992, pp. 235–248.

[LRZ93] William Landi, Barbara Ryder and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. *Proc. Conf. Programming Language Design and Implementation*, 1993.

[M+92] H.A. Muller, S.R. Tilley, M.A. Orgun, B.D. Corrie, N.H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. *Proc. 5th ACM SIGSOFT Symposium on Software Development Environments*, 1992.

[M+94] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson and Ted Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5), May 1994, pp. 58–71.

[MK88] Hausi A. Muller and Karl Klashinsky. Rigi: a system for programming-in-the-large. *Proc. International Conference on Software Engineering*, Singapore, 1988.

[MW90] Mark Moriconi and Timothy C. Winkler. Approximate reasoning about the se-

mantic effects of program changes. *IEEE Trans. on Software Engineering*, 16(9), September 1990, pp. 980–992.

[NEK93] Jim Q. Ning, Andre Engberts and Wojtek Kozaczynski. Recovering reusable components from legacy systems by program segmentation. *Proc. Working Conference on Reverse Engineering*, Baltimore, May 1993, pp. 64–72.

[NEK94] Jim Q. Ning, Andre Engberts and Wojtek Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5), May 1994, pp. 50–57.

[OO84] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984. *ACM SIGPLAN Notices* 19(5), pp. 177–184, May 1984.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE Trans. on Software Engineering*, 16(9), September 1990, pp. 965–979.

[PLR94] Hemant D. Pande, William A. Landi and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. on Software Engineering*, 20(5), May 1994, pp. 385–403.

[Rea92] Reasoning Systems. *Refine User's Guide*, Reasoning Systems, Palo Alto, California, 1992.

[RY89] Thomas Reps and Wuu Yang. The semantics of program slicing and program integration. *Proc. Colloquium on Current Issues in Programming Languages*, Barcelona, March 1989. Lecture Notes in Computer Science 352, pp. 360–374, Springer-Verlag, New York.

[Sel91] Peter G. Selfridge. Knowledge representation support for a software information system. *Proc. 7th IEEE Conf. on Applications of Artificial Intelligence*, pp. 134–140.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., 1989.

[Ste85] J.L. Steffen. Interactive examination of a C program with CScope. *Proc. USENIX Association*, *Winter Conf.*, January 1985.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4), July 1984, pp. 352–357.

[WH91] Norman Wilde and Ross Huitt. A reusable toolset for software dependency analysis. *Journal of Systems and Software*, Vol. 14, 1991, pp. 97–102.

[WL86] Mark Weiser and Jim Lyle. Experiments in slicing-based debugging aids. *Empirical Studies of Programmers*, ed. Elliot Soloway and Sitharama Iyengar, Ablex Publishing Corp., 1986, pp. 187–197.

[YL88] S. Yau and S.S. Liu. *Some approaches to logical ripple-effect analysis*. Software Engineering Research Center, SERC-TR-24F, University of Florida, October 1988.