Abstraction Mechanisms for Pictorial Slicing

Daniel Jackson and Eugene J. Rollins
School of Computer Science
Carnegie Mellon University

Abstract

Big programs tend to have big slices, so reverse engineering tools based on slicing must apply additional abstractions to make slices intelligible. We have built a tool that displays slices as diagrams. By confining the slice to the statements of a single procedure, by eliding all primitive statements, and by merging different calls of the same procedure, we eliminate local information that is easily seen in the code without the help of tools. And by labelling edges with the variables responsible for flows between procedure calls, global information about called procedures is represented locally. The resulting diagram gives a compact but rich summary of the role of called procedures in the slice.

Keywords

Reverse engineering, program comprehension, program slicing, program dependence graph, dataflow diagram, interprocedural analysis, modularity, abstraction.

1 Introduction

Understanding a program means at least knowing how variables at critical points in the program acquire their values. It is not surprising, then, that slicing—a mechanical analysis that marks all the statements that might influence the value of a variable at a given point in the program text [Wei84]—is widely viewed as a promising basis for reverse engineering tools.

In practice, though, slicing is not as useful as one might imagine. A slice is itself a program, after all, and, unless dramatically smaller than the original program, is unlikely to be much easier to understand. Unfortunately, big programs tend to have big slices, so some further form of abstraction is essential.

Address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Phone: (412) 268-5143. Fax: (412) 268-5576. 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.

We have developed a tool called Chopshop that slices C programs. In addition to highlighting the code to show traditional slices, it generates diagrams that illustrate how called procedures might affect the chosen variable. The abstractions use to form these diagrams were motivated by our intuitions of how programmers understand code, and their appropriateness has yet to be determined. Our prelimary experiments are encouraging, however; in examining a Unix utility we have discovered features of the code by looking at these diagrams in minutes that we had overlooked in several hours of code reading.

The diagram is a directed graph with labelled arcs. The nodes represent program statements and the arcs dataflow dependences between them. The labels on the arcs indicate which variables are responsible for the dataflow. This diagram is derived from a representation similar to the program dependence graph [FOW87], but its spirit is closer to the dataflow diagram used (albeit with a variety of differing, informal interpretations) in a number of development methods [DeM78]. Indeed, our work may be viewed as an attempt to bridge the gap between representations that can be generated easily from code and architectural descriptions that appeal to developers.

A number of abstractions are applied in the creation of the diagram. Most vital is the modular treatment of procedures. Standard interprocedural slicing [HRB90] pays no respect to procedure call boundaries; a slice on a variable that appears in some procedure *p* typically includes statements occurring both in procedures that call *p* and in procedures called by *p*. This approach was designed for applications that use slicing internally (for pruning regression test suites, e.g., or integrating different versions of a program); for reverse engineering, more structure is called for. Programmers tend to confine their analysis of a program to one procedure at a time, so it seems desirable that a reverse engineering tool should be capable of this too. A Chopshop user first selects a procedure; any subsequent slices treat this procedure as the entire program. Since the resulting slices mark relevant variables at the entry to this procedure and at the return of called procedures, the programmer can easily follow the slice into a calling or called procedure by slicing again on these variables .

Merely confining the slice to a single procedure does not, of course, solve the problem. Such a slice is rarely comprehensible; some summary of the role of called procedures is needed to explain why each call was included in the slice. The diagram shows what cannot easily be expressed textually: which statements affect which, and how they do so (that is, which variables carry the dataflow).

The abstraction of procedure calls summarizes global information so that it may be presented locally. The remaining abstractions eliminate local information that can easily be gleaned by reading the code (of the procedure being sliced, not the called procedures). Control dependences are eliminated first. They are usually evident from the syntactic nesting of the code, and increase the size of the diagram enormously because of their transitive effects. Second, calls of the same procedure at different sites are folded into a single node. Third, primitive statements (excluding the statement at the slicing point itself) are elided. Suppose, for example, that procedure p writes x and that, following the assignment $y = x$, procedure q reads y. The diagram would omit the assignment, but would connect node p to node q with an arc labelled y/x to indicate that the reading of y by q depends on the writing of x by p. All of these abstractions may be turned off by the user.

Presenting slices in this pictorial form appears to be novel. Most slicers, such as Andersen's Cobol/SRE [NEK94], are purely textual. A slicer being built at Microsoft Research uses an intermediate representation (the value dependence graph) that would allow similar kinds of abstraction [Ern94], and its developers have considered laying unlabelled arrows over text; so far, though, only text highlighting is used.

Many reverse engineering tools generate diagrams, but these tend to be derived from shallow semantic analyses. Refine/Cobol's set-use analysis [M+94], for example, can produce a diagram superficially similar to ours, but derived from a traditional cross-reference listing: an arc is shown connecting two procedures if there is a global variable written by one that is read by the other, whether or not a dataflow path is present. Rigi [M+92] lets the user impose structure on a program by applying various syntactic aggregation mechanisms to call-graphs; Chopshop, in contrast, is designed to expose semantic relationships between components chosen by the user. The idea of folding different calls of the same procedure comes from the star diagram of [BG94], which uses an analysis a bit like slicing to present candidate statements for encapsulation in an operation of an abstract type.

The next section gives an example of Chopshop's output and compares it to a traditional slice. The remaining sections explain the underlying dependence model, the slice computation and the abstraction mechanisms.

2 An Example

One of the procedures from the *more* utility of Berkeley Unix 5.22 is shown, without emendation, in Figure 1*. This procedure, *screen*, controls the basic cycle of *more*: displaying lines and prompting the user for input. From reading the code, we can guess what some of the called procedures do. Presumably *getline* reads a line from the file and *prbuf* displays it; the prompting of the user perhaps happens in *command*. But discovering the details is not easy. How does the line read by *getline* get passed to *prbuf*? Which file is being read? Indeed, is it even the same file for all executions of *getline*?

This procedure is hard to understand for many reasons, but two stand out. First, global variables are used pervasively; *screen* reads or writes more than 20, of which several appear only in procedures it calls. The variable *line*, for example, is a pointer to the string that is passed from *getline* to *prbuf*, but it appears nowhere in the call to *getline*. Second, the called procedures are big, and since their functionality is often as obscure as *screen*'s, attempting to understand *screen* by examining their code raises more questions than it answers. *Getline*, for example, is 109 lines long; *prbuf* is 38 lines and *command* a debilitating 235.

Let's now consider slicing the *screen* procedure. We notice that the variable *dlines* appears only once, about 10 lines from the bottom, so we slice on it to see where its value comes from. The resulting slice is shown underlined. Unfortunately, this does not help much: it is too large to assimilate and it conveys no explanation of why those statements were chosen. Following the slice into called procedures just makes matters worse.

Chopshop generates, in addition, the diagram of Figure 2. Each node represents a procedure call appear-

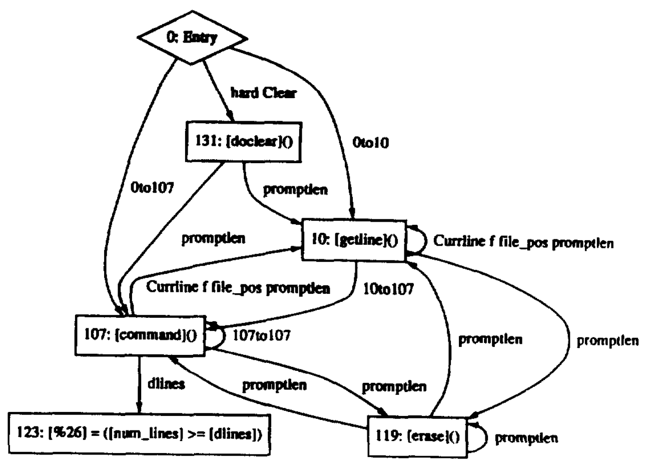
*Copyright © 1980 Regents of the University of California. All rights reserved. Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. This software is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranties of merchantibility and fitness for a particular purpose.

```
screen (f, num_lines)
register FILE *f;
register int num_lines;
{
   register int c;
   register int nchars;
   int length;              /* length of current line */
   static int prev_len = 1;           /* length of previous line */

   for (;;) {
    while (num_lines > 0 && !Pause) {
       if ((nchars = getline (f, &length)) == EOF)
       {
        if (clreol)
           clreos();
        return;
       }
       if (ssp_opt && length == 0 && prev_len == 0)
        continue;
       prev_len = length;
       if (bad_so || (Senter && *Senter == ' ') && promptlen > 0)
        erase (0);
       /* must clear before drawing line since tabs on some terminals
        * do not erase what they tab over.
        */
       if (clreol)
        cleareol ();
       prbuf (Line, length);
       if (nchars < promptlen)
        erase (nchars);  /* erase () sets promptlen to 0 */
       else promptlen = 0;
       /* is this needed?
        * if (clreol)
       · *     cleareol();  /* must clear again in case we wrapped *
        */
       if (nchars < Mcol || !fold_opt)
        prbuf("\n", 1);  /* will turn off UL if necessary */
       if (nchars == STOP)
        break;
       num_lines--;
    }
    if (pstate) {
        tputs(ULexit, 1, putch);
        pstate = 0;
    }
    (void) fflush(stdout);
    if ((c = Getc(f)) == EOF)
    {
       if (clreol)
        clreos ();
       return;
    }

    if (Pause && clreol)
       clreos ();
    Ungetc (c, f);
    (void) setjmp (restore);
    Pause = 0; startup = 0;
    if ((num_lines = command ((char *)NULL, f)) == 0)
       return;
    if (hard && promptlen > 0)
        erase (0);
    if (noscroll && num_lines >= dlines)
    {
       if (clreol)
        home();
       else
        doclear ();
    }
    screen_start.line = Currline;
    screen_start.chrctr = Ftell (f);
  }
}
```

Figure 1: Code of a procedure taken from Unix *more*, with slice on *dlines* (10 lines from the bottom) shown underlined.



0to10 = Currline Mcol dumb eraseln f file_pos fold_opt hard hardtabs pass_ctl promptlen stop_opt
0to107 = Clear Currline Senter Sexit clreol context dlines dum_opt eraseln errors f file_pos
    file_pos/f hard lastarg lastcmd no_intty noscroll otty promptlen screen_start shellp soglitch
10to107 = Currline f file_pos file_pos/f promptlen
107to107 = Currline dlines errors f file_pos file_pos/f lastarg lastcmd promptlen shellp

Figure 2: Slice diagram corresponding to textual slice of Figure 1

ing in *screen* that might be relevant to the slice criterion. Multiple call sites of the same procedure are folded together; *erase*, for instance, appears 3 times in the textual slice but only once in the diagram. No primitive statements are shown, except for the statement in which *dlines* itself appears. The arcs denote dataflow dependences; although control dependences within called procedures are accounted for, control dependences in *screen* itself are omitted. Each label on an arc indicates a variable defined by the procedure at the arc's source that is used by the procedure at the arc's target. Only variables relevant to the slice are shown, so, for example, there is no mention of *stdout* on the arc joining *doclear* to *command* even though *doclear* defines *stdout* and *command* uses it. When two variables appear in the form y/x as a single label, x is defined by the arc's source, y is used by the arc's target, and there are some elided primitive statements that cause the use of y to depend on the definition of x. A label list too long for its arc is given beneath the diagram, with a name giving the end points of the arc.

Examining the diagram reveals much more information than the textual slice about the interactions between the called procedures. We discover that *dlines* is set in *command* (since the arc from *command* carries *dlines* as a label). The initial value of *dlines* on entry to *screen* is relevant too (because of the label on arc *0to107*). The file pointer f (and the variable *file_pos*,

Size of *screen* procedure	74 lines
Sum of sizes of called procedures	1184 lines
Length of slice on *dlines*	21 lines
Size of *screen*'s PDG	138 nodes
Slice diagram, no abstraction	63 nodes
	67 dataflow arcs
	274 control arcs
...with control deps removed	15 nodes, 35 arcs
...and with primitives elided	8 nodes, 19 arcs
...and with calls folded	6 nodes, 15 arcs

Table 1: Effect of abstraction on slice of *screen* procedure

which turns out to be a copy of it) is relevant, and, surprisingly, modified in both *command* and *getline*. Most puzzling is the influence of *promptlen*, which is responsible for bringing in the procedures *erase* and *doclear*.

The variable *dlines* appears to hold the number of lines to be printed on the screen in the next cycle. To understand how it might be affected by *promptlen*, the length of the last prompt displayed by *command*, we applied Chopshop to the bodies of *getline* and *command*. We discovered that, as a fresh line from the file is written over the prompt, when a tab is encountered, and the prompt has not been entirely overwritten, the characters to the tab mark are erased. This seems to affect the length of the line written, and thus (since long lines are folded) the value of *dlines*. It is tempting to regard the dependence as spurious, but this cannot be justified without a detailed verification of *getline*. Frequently, obscure dependences that the programmer believes to be absent are exactly the ones that signal the presence of real bugs.

To illustrate the effects of Chopshop's abstractions, Table 1 shows how each in turn reduces the size of the diagram. The remarkable shrinkage caused by eliminating control dependences might not be typical. It arises here because of conditional exits from the loop that bring in many of the global variables that have no dataflow links to *dlines*.

3 The Dependence Model

Chopshop's underlying model is similar to the program dependence graph (PDG) [FOW87] but accommodates interprocedural dependences more naturally. The standard way to extend a graphical program representation (whether of dependences, control flow, etc.) to account for called procedures is to form a single supergraph for the entire program by joining the graphs of the individual procedures at their call sites. Extending the PDG in this way [HRB90] introduces a host of complications, not only in the construction process but also in the

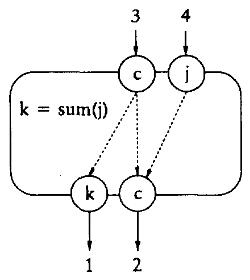


Figure 3: Summary dependences of called procedure. Slicing on the definition of *k* (1) follows through to the dependence edge incident on the use of *c* (3), while slicing on the definition of *c* (2) follows through on both edges (3 and 4).

graph itself. Each call site becomes a complex web of linkage nodes and mock assignments (to model the passing of parameters) that is not conducive to a modular analysis.

Our approach is simpler. A procedure call is modelled as a single node, as if it were a primitive statement. The effects of its internal dependences at the call site are represented by a def-use relation between the result and argument variables of the call. For example, the call

$$k = sum(j)$$

to the procedure

```
int sum (int i) {
    int t;
    t = c;
    c = c + i;
    return (t) }
```

would be summarized by the def-use relation

$$\{(c, j), (c, c), (k, c)\}$$

indicating that the definition of the global *c* depends on the use of the argument *j* and the use of *c* (that is, its old value), and that the definition of *k* depends only on the use of *c*. The local variable *t* is invisible to callers, so it does not appear.

It may be helpful to think of this relation as a set of edges inside the procedure call node. A slice on *c* after the call, for example, is found by following the two internal edges from *c* at the bottom to *j* and *c* at the top; the edges incident on the uses of these two variables are then followed to their definitions at preceding nodes, and so on. Note that a slice on *k* would traverse only a single internal edge to the use of *c*, and from there to its definitions, so that the edge bringing the definition of *j* to the call would be followed in the first case but not the second. This scenario is illustrated in Figure 3.

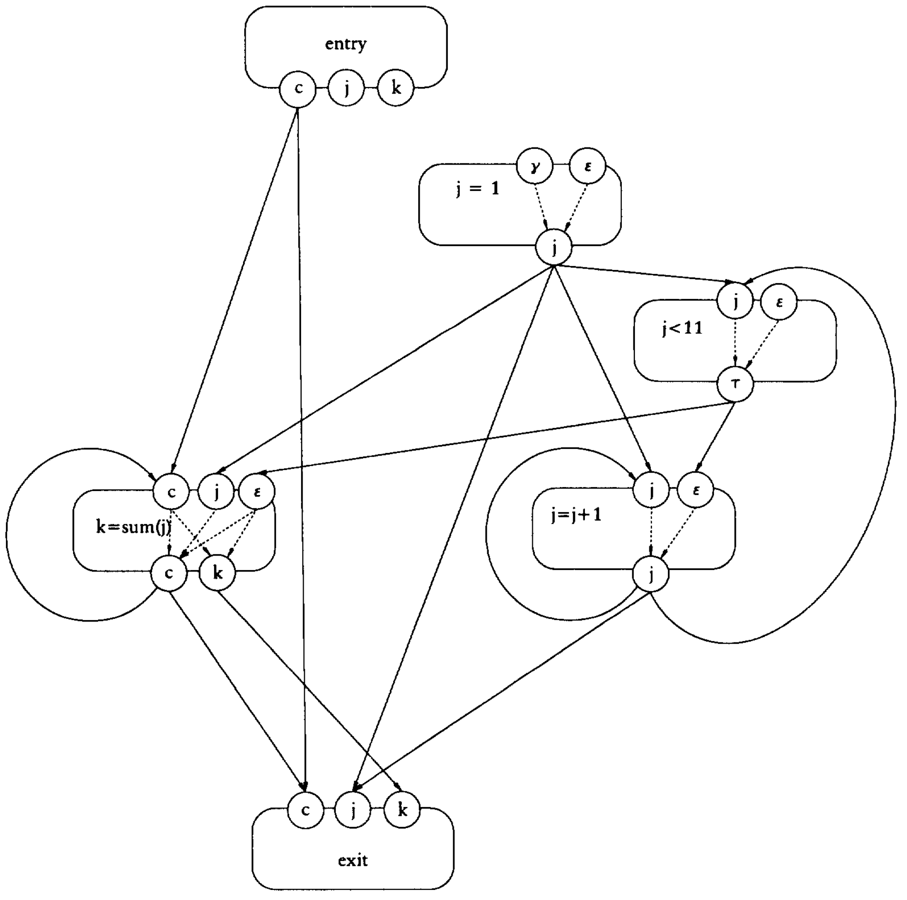


Figure 4: A dependence graph.
The *du* relation is shown by the dotted edges;
the solid edges belong to *ud* (when they link program variables),
and *cd* (when they link an execution ε to a temporary τ).

For uniformity, all nodes are treated in the same way, whether procedure calls or primitive statements. Since the caller of a procedure cannot observe any difference between control and dataflow dependences inside the call, there is no need to distinguish them in the internal edges. Between statements, however, the two kinds of dependence must be separated.

All dependences relate the use of a variable at one point to a definition at another. We call these points "sites" as they do not always correspond to program statements. Nesting of expressions causes a single source statement to be translated into several nodes connected by temporary variables.

Our model thus consists of three binary relations, *ud*, *du* and *cd*, over variable/site pairs (or "instances"). The dataflow dependence relation *ud* relates a use of a variable at one site to a definition at another; it contains the pair $((x, i),(x, j))$ if there is a use of x at site i that has a reaching definition at site j.

The control dependence relation *cd* relates the execution of one site to the evaluation of an expression at another; it contains the pair $((\varepsilon, i),(\tau, j))$ when execution of site i, modelled by the dummy variable ε, depends on the expression at site j, where τ is a temporary variable holding the result of the expression.

The summary relation *du* (which we referred to above as the "internal edges") relates the definition of a variable at a site to a use at the same site; it contains $((x, i),(y, i))$ when the effect of site i is to make x afterwards depends on y before. For every variable x modified at site i the pair $((x, i),(\varepsilon, i))$ is included too, to show that the definition of x depends on the execution of the site. If a variable x is modified but is not dependent on any use, the pair $((x, i),(\gamma, i))$ is inserted to show a dependence on some constant γ.

Formally, then, we have:

$$Var = ProgramVariables \cup \{\gamma, \varepsilon, \tau\}$$
$$Instance = Var \times Site$$
$$du, ud, cd: Instance \leftrightarrow Instance$$

Figure 4 illustrates these relations for this procedure, which calls *sum* to add the numbers from 1 to 10 to the global *c*:

```
total () {
  j = 1;
  while (j < 11) {
    k = sum (j);
    j = j + 1; }}
```

There are two special sites for every procedure: an entry and an exit. All variables are defined at the entry and used at the exit. The instances are drawn, as in Figure 3, as ports of the nodes, although this time the special variables are shown too. To see how control dependences work, follow the path back from *j* at the exit through $j = j + 1$; note how the definition of *j* depends on whether the increment is executed (ε), which depends on the outcome of the loop test (τ), which in turn depends on the value of *j* before the test. (Incidentally, the graph is not complete. One arc is missing: it proved to be beyond the authors' drawing abilities and is left as an exercise for the reader.)

Constructing the dependence relations is straightforward; the only subtleties arise in obtaining *du* from the bodies of called procedures, and handling recursion. The details, along with a more substantial justification of the model, may be found in our technical report [JR94a] and a paper to appear soon [JR94b].

4 Constructing Slice Diagrams

Rather than giving explicit worklist algorithms for constructing slices, we shall define slices as algebraic expressions in terms of the dependence relations. We find these much easier to understand and manipulate. As specifications, the expressions allow a variety of implementations, but the current version of Chopshop actually implements them directly. Altering the slicing algorithm or the abstraction mechanisms thus involves changes to only a few lines of code.

We use six relational operators. Union, composition, transitive closure and projection of a set are standard:

$$p \cup q = \{(a, b) \mid (a, b) \in p \lor (a, b) \in q\}$$
$$p \circ q = \{(a, b) \mid \exists z \in T.\ (a, z) \in p \land (z, b) \in q\}$$
$$p^* = I \cup p \cup (p \circ p) \cup (p \circ p \circ p) \cup \ldots$$
$$p[S] = \{b \mid \exists a \in S.\ (a, b) \in p\}$$

The domain restriction of a relation *p* to a set *S* is the relation containing all pairs in *p* whose first element is in *S*:

$$S \lhd p = \{(a, b) \in p \mid a \in S\}$$

Similarly, the range restriction of *p* to *S* contains pairs whose second elements are in *S*:

$$p \rhd S = \{(a, b) \in p \mid b \in S\}$$

The slice criterion is a set of variables *V* and a site *i* at which they are used, that is, a set of instances

$$C = V \times \{i\}$$

The reaching definitions that affect these uses directly are given by the projection of C under the *ud* relation. The uses affecting these definitions are found by projecting this set under the relation *du*. This process is repeated to find all uses that affect C. The set of relevant uses is thus the projection of C under the transitive closure of $ud \circ du$:

$$relUses = (ud \circ du)^* [C]$$

The basic diagram is now obtained by restricting the *ud* relation to these uses:

$$ud' = relUses \lhd ud$$

Now we apply two abstractions. Nodes corresponding to different calls of the same procedure are merged; this is easy and not worth formalizing. To elide primitive statements, we start by collecting together the instances

$$drop: \mathbb{P}\ Instance$$

associated with the nodes to be dropped, namely all the non-calls except the site of the slice criterion and the entry and exit, and the instances associated with nodes to be kept (all the rest):

$$keep: \mathbb{P}\ Instance$$

The elision happens in two stages. First we add transitive edges that replace dependences brought about by paths passing through dropped nodes:

$$ud'' = ud' \cup (ud' \circ (drop \lhd du \circ ud')^*)$$

Second, we restrict the diagram so that only nodes with instances in *keep* remain, giving

$$ud''' = keep \lhd ud'' \rhd keep$$

This relation, *ud'''* is the final, abstracted slice diagram. A pair $((x, i),(y, j))$ in this relation connects a use of a variable *x* at site *i* to a definition of a variable *y* at site *j*.

Only one edge is drawn between i and j, irrespective of how many pairs connect them. If x and y are the same variable, v say, the edge is labelled v; if x and y differ, the edge is labelled x/y to show that the use of x is indirectly due to the definition of y.

We have illustrated only a limited form of slicing. Most variants of slicing (such as slicing forwards instead of backwards, or slicing on a definition instead of a use) are easily expressed in our model [JR94a,b]. Chopshop implements a variant we call (predictably) "chopping", whose criterion is two sets of instances, *source* and *sink*; the chop shows how the definitions in *source* affect the uses in *sink*.

5 The Chopshop Tool

Chopshop is implemented in Standard ML. It runs as a subprocess under *emacs 19* (a new version that supports colour highlighting). The diagram is written to a file as an adjacency list, which is then converted to postscript by AT&T's *dot* program, and finally displayed by the *ghostview* previewer.

Selecting a procedure of interest initiates the analysis of the code. The def-use abstractions of procedures are cached, so they need be calculated only once. The tool also maintains a variety of closure relations, so that the chopping operations can be performed in linear time.

The performance of the current version is acceptable only for small programs. It takes about 15 minutes to analyze the entire *more* program (approximately 2000 lines), and about ten seconds to perform each chop. We are working on a number of improvements. By hardwiring the analysis (in contrast to direct implementation of the relational expressions), we expect to increase its speed dramatically. The next version will store the def-use abstractions of procedures in a library whose entries can be read and written by the user; this will allow Chopshop to handle calls to procedures with no code, either because they are low-level, or because they have yet to be written. It will also include an alias analysis to account for dependences caused by indirect accesses.

References

[BG94] R.W. Bowdidge and W.G. Griswold. Automated support for encapsulating abstract data types. *Proc. ACM Sigsoft '94 Symp. on Foundations of Software Engineering*. New Orleans, La., December 1994.

[DeM78] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

[Ern94] Michael D. Ernst. *Practical fine-grained static slicing of optimized code*. Technical report MSR-TR-94-14, Microsoft Research, Redmond, Wa., July 1994.

[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.

[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.

[JR94a] Daniel Jackson and Eugene J. Rollins. *Chopping: a generalization of slicing*. Technical report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., July 1994.

[JR94b] Daniel Jackson and Eugene J. Rollins. A new abstraction of the program dependence graph for reverse engineering. *Proc. ACM Sigsoft '94 Symp. on Foundations of Software Engineering*. New Orleans, La., December 1994.

[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.

[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.

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