PROGRAM SLICING*

Mark Weiser

Computer Science Department
University of Maryland
College Park, MD 20742

Abstract

Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.

Finding a slice is in general unsolvable. A dataflow algorithm is presented for approximating slices when the behavior subset is specified as the values of a set of variables at a statement. Experimental evidence is presented that these slices are used by programmers during debugging. Experience with two automatic slicing tools is summarized. New measures of program complexity are suggested based on the organization of a program's slices.

KEYWORDS: debugging, program maintenance, software tools, program metrics, human factors, dataflow analysis

Introduction

A large computer program is more easily constructed, understood, and maintained when broken into smaller pieces. Several different methods decompose programs during program design, such as information hiding (Parnas 1972), data abstraction (Liskov and Zilles 1975), and HIPO (Stay 1976). These methods are not mutally exclusive, but rather complement one another. Proposed here is another complementary method of program decomposition: program slicing. Unlike most other methods (but see Tarjan and Valdes, 1980), slicing is applied to programs after they are written, and is therefore useful in maintenance rather than design. Unlike design methodologies, working on actual program text allows slicing to be specified precisely and performed automatically.

Slicing starts with the observation that these are times when only a portion of a program's behavior is of interest. For instance, during debugging a subset of behavior is being corrected, and in program modification or maintenance a subset of behavior is being improved or replaced. In these cases, a programmer starts from the program behavior and proceeds to find and modify the corresponding portions of program code. Code not having to do with behavior of interest is ignored. Gould and Dronkowski (1974) report programmers behaving this way during debugging, and a further confirming experiment is presented below.

A programmer maintaining a large, unfamiliar program would almost have to use this behavior-first approach to the code. Understanding an entire system to change only a small piece would take too much time. Since most program maintenance is done by persons other than the program designers, and since 67 percent of programming effort goes into maintenance (Zelkowitz, Shaw, and Gannon 1979), decomposing programs by behavior must be a common occurence.

Automatic slicing requires that behavior be specified in a certain form. If the behavior of interest can be expressed as the values of some sets of variables at some set of statements, then this specification is said to be a slicing criterion. Dataflow analysis (Hecht 1977) can find all the program code which might have influenced the specified behavior, and this code is called a slice of the program. A slice is itself an executable program, whose behavior must be identical to the specified subset of the original program's behavior.

Figure 1 gives examples of some slicing criteria and their corresponding slices.

There are usually many different slices for a given program and slicing criterion, depending on how minimal a slice is desired. The issue of minimality is discussed further below. There is always at least one slice--the program itself. The interesting slices are the ones which, compared to the original program, are significantly smaller and simpler.

The idea of isolating portions of programs according to their behavior has appeared previously. Schwartz (1971) hints at such a possibility for a debugging system. Browne and Johnson (1978) describe a question-answerer for Fortran programs

*This research was supported in part by the Computer Science Center at the University of Maryland, and by grants from the Air Force Office of Scientific Research and the General Research Board at the University of Maryland.

CH1627-9/81/0000/0439$00.75 © 1981 IEEE

Examples of Slices

The original program:

```
 1   BEGIN
 2   READ(X,Y)
 3   TOTAL := 0.0
 4   SUM := 0.0
 5   IF X <= 1
 6      THEN SUM := Y
 7      ELSE BEGIN
 8         READ(Z)
 9         TOTAL := X*Y
10         END
11   WRITE(TOTAL,SUM)
12   END.
```

Slice on the value of Z at statement 12.

```
BEGIN
READ(X,Y)
IF X < 1
   THEN
   ELSE READ(Z)
END.
```

Slice on the value of X at statement 9.

```
BEGIN
READ(X,Y)
END.
```

Slice on the value of TOTAL at statement 12.

```
BEGIN
READ(X,Y)
TOTAL := 0
IF X <= 1
   THEN
   ELSE TOTAL := X*Y
END.
```

Figure 1

which, through a succession of questions, could be made to reveal the slices of a program although very slowly. Several on-line debuggers permit a limited traceback of the location of variable references (e.g. Aygun, 1973), and this information is a kind of "dynamic slice".

Slicing is a source code transformation of a program. Previous work in program transformation has concentrated on preserving program correctness while improving some desirable property of programs. For instance, Baker (1977) and Ashcroft and Manna (1973) both are concerned with adding "good structure" to programs. Wegbreit (1976), Arsac (1979), Gerhart (1975), and Loveman (1977), are more oriented to improving a program's performance.

Slicing Algorithms

This section more formally discusses the ideas of a slicing criterion and a slicing algorithm, using the reader's intuitive understanding of machine execution. All proofs have been carried out in an abstract operational model (Weiser 1979).

This paper considers the slicing of block-structured, possibly recursive programs written in a Pascal-like language. All variables are assumed to be uniquely named, and all procedures are assumed to be single-entry, single exit.

The following notation is used throughout this paper. Due to typesetting limitations, square brackets ([...]) are used to enclose superscripted and subscripted quantities. Set notation is expressed as follows. Let A and B denote sets, let f and g be functions whose values are sets, and let C(i) be a finite family of sets indexed by i. Then:

A union B	denotes the set union of A and B.
A intersect B	denotes the set intersection of A and B.
f union g	denotes the function whose value is f(n) union g(n) for each n in the domain of f and g, and is undefined elsewhere.
UNION C(i)	denotes the union of all C(i) for each i.

A slicing criterion for a program specifies a window for observing its behavior. A window is specified as a statement and a set of variables. The window allows the observation of the values of the specified variables just before executing the specified statement. If the statement specified by the slicing criterion is executed several times while the program is running, then a sequence of variable values will be observed.

Identifying statements by numbers and variables by name, a slicing criterion is a pair $<i,v>$, where i is the number of the statement at which to observe and v is the set of variable values to be observed.

There are two properties intuitively desirable in a slice. First, the slice must have been obtained from the original program by statement deletion. Second, the behavior of the slice must correspond to the behavior of the original program as observed through the window of the slicing criterion. Both of these informal properties allow several interpretations. The interpretation used here is derived and justified in the next several paragraphs.

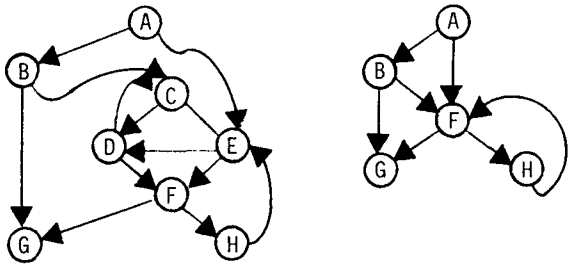
The problem with obtaining a slice by statement deletion is that the source code of a program may become ungrammatical. For instance, removing the THEN clause from an IF-THEN-ELSE statement leaves an ungrammatical fragment if the "null" statement is not permitted between THEN and ELSE. Because of their language dependence, detailed consideration of these issues is beyond the scope of this paper. See Arsac (1979) for some approaches. Instead, a flowgraph will be used to represent a program, with each node in the graph corresponding to a single source language statement. The terms "node" and "statement" will be used interchangably.

A flowgraph is a structure $G = <N,E,n0>$, where N is the set of nodes, E is a set of edges in NxN, and n0 is the distinguished initial node. If (n,m) is an edge in E then n is an immediate predecessor of m, and m is an immediate successor of n. A path of length k from n to m is a set of nodes p(0),p(1),

...,p(k) such that p(0) = n, p(k) = m, and (p(i), p(i+1)) is in E for all i, $0 \leq i < k-1$. There is a path from n0 to every other node in N. A node n is nearer than a node m to some node q if the shortest path from n to q has length less than the shortest path from m to q. A node m is dominated by a node n if n is on every path from n0 to m. An inverse dominator is a dominator on the flowgraph obtained by reversing the direction of all edges and making the final node the initial node.

Deleting statements in a flowgraph produces a meaningful new flowgraph so long as any group of statements deleted have only a single successor (see figure 2). This restriction ensures that no statement increases its number of immediate successors as a result of statement deletion. The graph transformation following statement deletion is just: All predecessors of any member of a deleted group of statements have the deleted group's unique successor as their new successor.

Group of statements with a single successor



Nodes C,D, and E form a set with a single successor, F, not in the set. The flowgraph is shown before and after removing this set.

Figure 2

The second desirable property of slices is that they duplicate the behavior observable through the window of the slicing criterion. This means observing original program and slice through the "same" window, and not being able to distinguish between them. But how can a slicing criterion for one program (the original) be used to specify a window for a different program (the slice)? A slicing criterion has the form <i,v>. v can be used in both the slice and the original program, of course. However statement number "i" may not even exist in the slice. Therefore, the window for observation of the slice is specified as <SUCC(i),v>. SUCC(i) is the nearest successor to "i" in the original program which is also in the slice, or "i" itself if present. It is easy to prove that SUCC(i) is unique.

The program and its slice now have corresponding windows for observing behavior. A reasonable requirement for a slice might be that the trajectories of behavior observable through the slice window must be identical to that observable through the original program window for all inputs. Unfortunately this condition is too strong, because it implies the unsolvability of finding slices. Consider the following program skeleton:

```
1    BEGIN
2    READ(X)
3    IF X = 0
4      THEN BEGIN
           .
         perform infinite loop
         without changing X.
           .
5          X := 1
6          END
7      ELSE X := 2
8    END.
```

Let the slicing criterion be the value of X at line 8. A slicing algorithm based on equivalent behavior trajectories for all inputs would necessarily include line 5 unless there were some assurance that for all input line 5 was never reached. Such a slicing algorithm could be used to determine the termination of an arbitrary procedure by suitably inserting that procedure between lines 4 and 5, and then noticing whether or not line 5 appeared in the slice. But there can be no algorithm to determine if an arbitrary procedure must terminate, and hence no such slicer can exist.

To fix this problem, the requirement of equivalent projected behaviors can be weakened to be: projected behaviors must be equivalent whenever the original program terminates. This definition is the one intended in the remainder of this paper whenever the phrase "equivalent behavior" is used.

A similar problem now arises with finding the smallest possible slice. The reader can easily generalize the above example to show that no algorithm can always find the slice with the minimum number of statements, because of the impossibility of evaluating the functional equivalence of two different pieces of code. This problem suggests that a practical definition of a minimal slice must avoid exact knowledge of the functions computed by pieces of code. Dataflow information about a program is of this type, and it permits an exact slicing algorithm. The remainder of this section considers the computation of slices from dataflow information alone.

Before going on, the objection can be raised that dataflow analysis provides information that is too imprecise to be of any use. Programmers know more about their programs than dataflow analysis can reveal, and so dataflow slices may not be sufficiently smaller than the original program to justify their use. For instance, arrays and pointer variables usually require worst-case assumptions about variable influence (Aho and Ullman 1977). For more recent work see Reynolds (1979), Luckham and Suzuki (1979), and Weihl (1980). However, dataflow approximations have the advantage of being computable in polynomial time, and in practice are often good enough. Sections 3 and 4 below present empirical evidence for the adequacy of dataflow slices.

Dataflow algorithms

Finding slices using dataflow analysis begins by tracing backwards possible influences of vari-

ables. This process is similar to "reaching definitions" (e.g. Hecht 1978). In effect, slicing means knowing which variable assignments can "reach" (i.e. have an effect on) the variables observed through the window of the slicing criterion.

In general, for each statement in the program there will be some set of variables whose values can affect, through some chain of assignments, a variable observable at the slicing criterion. For instance, if the statement "Y := X" is followed by the statement "Z := Y", then the value of X before the first statement can affect the value of Z after the second statement. Let R[0,C](n) be the variables at statement n whose values can directly affect what is observed through the window defined by criterion C. The "0" (zero) refers to the direct effect. R[1,C], R[2,C], etc. will be defined later for increasing levels of indirect effect.

To define R[0,C] formally, consider the dataflow information known about each statement. It is a convenient simplification to allow only two kinds of dataflow information: variables altered (called DEF), and variables referenced (called REF). DEF(n) is the set of variables whose values may be changed at node n. REF(n) is the set of variables whose values may be referenced at node n. The propagation of influence from variable to variable is deduced from the assumption that if w and x are variable names, with w in DEF(n) and x in REF(n) for some statement n, then the value of x can influence the value of w.

This assumption may be inaccurate for cases of multiple assignment or procedure calls. A more accurate formulation which uses precise within-statement influences can be found in Weiser (1979). All the results in this section hold for that more precise formulation.

R[0,C](n), where C = <i,v>, is defined recursively as follows.

R[0,C](n) = all variables v such that either:
1. n=i and v is in V,
or 2. n is an immediate predecessor of a node m, such that either:
   a) v is in REF(n) and there is a w in both DEF(n) and R[0,C](m),
   or
   b) v is not in DEF(n) and v is in R[0,C](m).

The reader can check that the recursion is over length of paths to reach node i, and that case 2) above corresponds to observing behavior just before executing a statement. Notice that 2b) assumes that all statements not altering a variable will preserve its value. This is a simplification of the usual dataflow information, which separately uses "PRE" information for this.

The definition of R[0] can also be specified by a pair of equations giving values for R[0] coming into and going out of a statement (Aho and Ullman 1977). For those readers more familiar with this notation, here is the definition:

(1) RIN(n) = ROUT(n) - DEF(n) union REF(n)
union {if n=i then all v in V}
(2) ROUT(n) = UNION RIN(m), for all m imm. successors of n.

It happens that R[0] can be imbedded in a fast monotone information propagation space, in the sense of Graham and Wegman (1976). This means R[0] can be found in worst-case time O(e log e) for a flowgraph with e edges, and in time O(e) for structured programs. The proof is immediate from noticing that except for the constant union when n = i, equation (1) above corresponds to Graham and Wegman's example on the top of page 176. A constant union is irrelevent in defining the information propagation space, so the Graham and Wegman result applies.

The REF and DEF information about statements can usually be obtained by inspection of the source code. When it can't because the statement is a procedure call, dataflow information about the call must be obtained by an interprocedural dataflow method. Barth (178) is good in practice, and has been used in the implementation of the slicing algorithm described in section 4. Additional interprocedural slicing issues are also discussed there.

R[0] provides a sufficient condition for including statements in a slice. Any statement n which changes a variable in R[0,C](n) must be in the slice on C. Excluding such a statement from the slice would require additional information about the function computed by that statement. For instance, a statement n with REF(n) = DEF(n) = {x} need not be in the slice if it computes the identity function on x. But this is just the sort of information excluded from dataflow analysis, although more sophisticated methods could take it partially into account.

The statements included in the slice by R[0,C] are denoted S[0,C]. S[0,C] is defined by:

S[0,C] = all nodes n such that R[0,C](n) intersect DEF(n) is non-empty.

R[0] does not allow for the indirect effects of conditional control-flow, and therefore is not a necessary condition for including statements in a slice (see figure 3).

Slicing Criterion C = <5,{Z}>

```
1    READ(X)
2    IF X < 1
3       THEN Z := 1
4       ELSE Z := 2
5    WRITE(Z)
```

Even though statement 2 changes no variables and hence is not in S[0,C], it obviously has an effect on the value of Z and should be in the slice for C.

Figure 3

Generally, any branch statement which can choose to execute or not execute some statement in S[0] may cause a change in behavior observable through the slicing criterion.

Finding such branch statements can be done in several ways. For instance Denning and Denning (1977), in computing secure information flow, use the nearest inverse dominator of a branch statement to define its range of influence. An inverse dominator D(n) of a statement n is on every path from n to the final statement of the flowgraph. Therefore, for any statement x on a path from a statement n to its nearest inverse dominator d, there is another path from n to d which excludes x. Presumably by choosing paths which did or did not execute x, b could exert an indirect influence over any variable directly influenced by x.

Another method which is more exact but more expensive is given in Weiser (1980). This is the method used in the automatic slicers described in section 4. In what follows Denning's approach is used for ease of presentation.

Let ND(b) be the set of statements which are on a path from b to its nearest inverse dominator d, excluding b and d themselves. ND(b) will be empty unless b has more than one immediate successor. If S[0] is the set of statements with direct influence, then a statement b has indirect influence just if S[0] intersect ND(b) is non-empty. This prompts the following definition:

For any flowgraph G, and any set of statements P,
CS[G](P) = all nodes n such that either:
a) n is in P,
or
b) ND(n) intersect P is non-empty.

The subscript G will be dropped when obvious from context.

CS has the "semi-linear" property that:
CS[G](A) union CS[G](B) = CS[G](A union B)
for any two sets of statements A and B. In most of what follows only the monotonicity of CS is required, namely that:
$A \subseteq B \Rightarrow CS(A) \subseteq CS(B)$
This follows immediately from the previous property.

The set of statements in CS(S[0]) will include those branch statements which can influence the execution of a statement in S[0]. For convenience the branch statements included by CS(S[0]) will be denoted by B[0]. In other words, B[0,C] = CS(S[0,C]0 - S[0,C]).

To include all indirect influences, the statements with direct influence on B[0] must now be considered, and then the branch statement influencing those new statements, etc.

For convenience, let BC(b) denote the slicing criterion defined as <b,REF(b)>. If b is a branch statement, then REF(b) is assumed to be the set of variables which can influence the choice of paths from b. The variables at a statement n which directly influence a branch statement b are just R[0,BC(b)](n). The next level of influence, R[1], is therefore defined as:

R[1,C](n) = R[0,C](n) union {s in R[0,C](n), for all b in B[0,C]}.

Similarily for S[1]:

S[1,C] = {n: DEF(n) intersect R[1,C](n) is non-empty,
or
n is in B[0,C]. }

More generally, R,B, and S are defined for all levels:

R[i+1,C](n) = R[i,C](n) union {s in R[0,BC(b)](n), such that b is in B[i,C]. }

B[i+1,C] = CS(S[i+1,C])

S[i+1,C] = {n: DEF(n) intersect R[i+1,C](n) is non-empty,
or
n is in B[i,C]. }

The recursion starts with the S[0], R[0], and B[0] as previously defined.

Considered as a function of i, for fixed n, these definitions are non-decreasing. E.g., R[i,C](n) is a subset of R[i+1,C](n) for all n, and similarily for B[i,C] and S[i,C].

Dropping the superscript, we let R[C] denote the least fixed-point of R[i,C], and S[C] the least fixed-point of S[i,C]. Obviously,
R[C] = UNION R[i,C], for all $i \geq 0$.
S[C] = UNION S[i,C], for all $i \geq 0$.

The time to compute R and S is no worse than $O(n \cdot e \cdot \log e)$ for a flowgraph with n nodes and e edges. This bound is probably not tight, since practical times seem much faster. The worst case analysis is as follows: Each computation of S[i+1,C] from S[i,C] requires an initial $O(e \cdot \log e)$ step to compute influence. The remaining computation of B[i,C] is largely finding the dominators, which takes at most $O(e \cdot \log n)$ time (Lengauer and Tarjan 1979). The value of i can be at most n, hence the total complexity is at worst $O(n \cdot (e \cdot \log e + e \cdot \log n))$, or approximately $O(n\ e \log e)$.

The algorithm above is conservative, because the statements in S[C] are sufficient to display all the behavior observable from window C. Intuitively, this is true because all possible sequences of dataflow and control flow influence have been accounted for, and there are no other sources of influence. A more formal proof requires a detailed model of possible computations, and is too long to reproduce here.

Slices have the following properties.

Properties of Slices:
Let C = <i,v>, D = <i,w>, E = <i,v union w>, and F = <j,w>.
C,D,E, and F are all slicing criteria.

Property A:
S[C] union S[D] = S[E]

R[C] union R[D] = R[E]

Property B:
Let F = <j,w>
W ⊆ R[C](j) => S[W].⊆ S[C]

Is S[C] always the "smallest" slice that can be found using only dataflow and control flow information? No. If the code being sliced does strange things, then the iterating of R and B can produce anomalies in the analysis, as shown in figure 4. Such cases are probably rare in practice.

```
1     A := constant
2     WHILE P(K)
         DO
         BEGIN
3        IF Q(C)
            THEN BEGIN
4              B := A
5              X := 1
               END
            ELSE BEGIN
6              C := B
7              Y := 2
               END
8        K := K + 1
         END
9     Z := X + Y
10    WRITE(Z)
```

Criterion = <10,{Z}>
The first level of anlaysis gives:
S[0] = {5,7,9}
B[0] = {2,3}
Because statement 3 uses C, the next level shows:
S[1] = {1,2,3,4,5,6,7,8,9}
Statement 1 is included because it can influence the value of C via statements 4 and 6. But the value of C is only important to choose between statements 5 and 7. By the time A can influence C, all possible successors to statement 3 must have already been executed, so statement 1 actually can have no effect, and should not be in the slice. Notice that this argument can be carried out with dataflow information alone.

Figure 4

Practical Slicing

Slicing has been empirically investigated in two ways. Preliminary indications that slicing is useful were obtained by showing that experienced programmers already use slicing during debugging. This led to the construction of a series of slicing tools. Preliminary results from using these tools show that in practice slicing is fairly fast, and can often eliminate large numbers of unnecessary statements from slices of programs.

The details of the experiment on slicing are being reported elsewhere, so just an overview and the important conclusions are given here. The participants were 21 experienced programmers drawn from the academic computing community at the University of Michigan in Ann Arbor. Counting multiple roles, 12 individuals had taught programming, 10 were working as counselors to users of the University computing center, and 6 were professional programmers of several years experience. There were no significant correlations between type of experience and slicing.

Each participant in the experiment was given three programs to debug. Supplied with each program was a brief description of its purpose, and a sample of output which clearly showed the bug. The bugs were deliberately kept simple, and were found in 59 out of the 63 possible opportunities.

The three programs being debugged had lengths of 75, 121, and 150 lines each of ALGOLW code. All the participants were familiar with ALGOLW. Each program consisted of a main program which did all input and output, and at least one major subroutine which contained the bug. Comments were few and high-level.

After finding all three bugs, each participant was shown fragments of "algorithms" that had been present in the three programs. Participants indicated on a one to four scale (see figure 5) whether or not they thought each "algorithm" had been used in one of the three programs.

An example "algorithm" and rating scale

```
T1 := X1:
FOR B1 := X2 UNTIL N1 DO
   BEGIN
   R1(N2,H1,B1,E2):
   IF H1 > X3 THEN
      BEGIN
      IF E2 THEN
         E3 := (H1-X3)
      END:
T1 := T1 + E3:
END:
```

☐ almost certainly used.
☐ probably used.
☐ probably not used.
☐ almost certainly not used.

Figure 5

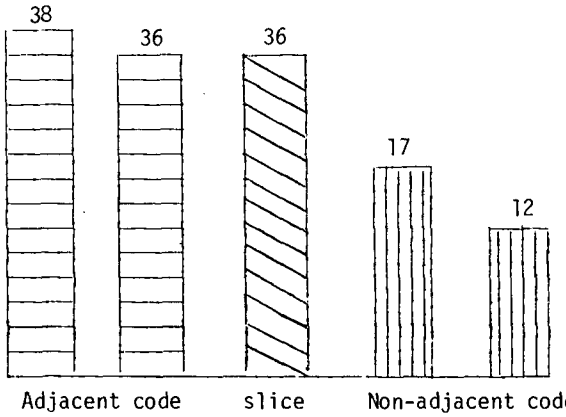
The algorithms participants rated were two locally adjacent segments of code and three imbedded non-adjacent segments of code from each original program. Adjacency and non-adjacency refer to whether the statements in the code segment were next to one another in the original program: e.g. a slice is usually non-adjacent. Five "algorithms" were taken from each program. Each was truncated at top and bottom to a length of about 10 statements, and had all of its variable names changed.

For each program, one of the two adjacent algorithms consisted of the 10 statements nearest (and including) the statement causing the bug, and the other consisted of 10 adjacent statements from elsewhere in the program. Of the non-adjacent algorithms, one was a slice based on the output

statement and variable at which the error became visible. This slice had relatively few statements in common with either of the two adjacent algorithms. One of the other two non-adjacent algorithms was a slice not relevant to the bug, and the other was constructed by doctoring the segment of code consisting of every third or fourth statement to look like a possible foreshortened algorithm.

In all three programs, the slice relevant to the bug was remembered as having been used or probably used in almost half of the 63 cases. This was not significantly worse than how well the two adjacent code segments were remembered. Of course, the adjacent code was expected to be remembered since Shneiderman (1976) has shown that experienced programmers can reproduce functionally equivalent programs from memory. But no previous work would have led to the expection that imbedded non-adjacent algorithms would be remembered.

Of non-adjacent code, the relevant slice did much better than the other two non-adjacent "algorithms". A summary of the results pooled for all three programs is shown in figure 6.



Graph shows the number of times 21 programmers, after debugging three programs each, rated imbedded algorithms as "probably" or "possibly" used. All algorithms were actually present. See text for details.

Figure 6

Since programmers remembered the relevant slices from programs they had just debugged, they probably were mentally constructing and using those slices while debugging. The results for the irrelevant slices show that not just any imbedded algorithm was remembered. Presumably, each programmer had in his or her career independently developed the slicing method, indicating that slicing must have been a useful technique for each of them.

All the participants in this experiment were experienced programmers. It would be interesting to look at novice/expert differences in the use of slicing, and also at differences in debugging performance between novice programmers taught and not taught about slicing as a debugging technique.

Slicing Real Programs

The first program slicer was implemented as a post-processor to the DAVE program for analyzing FORTRAN source code (Fosdick and Osterweil, 1976). This arrangement could slice only main programs, although calls from the main program to other subroutines were permitted.

Several programs were sliced by this prototype system (see table 1). Slicing criteria were based on variables whose values were printed near the end of the program. In all, three different slices were taken of each of four programs. For the larger programs, the slices were considerably smaller than the original code. This was what was hoped for.

Results from Prototype Slicer

PROGRAM	Number of Lines	Average Slice Size
MMGS	15	15
PH2B	60	20
TALLY	67	25
MAIN	380	100

Table 1

The small program identified as "MMGS" is a matrix multiplication subroutine in the IBM Scientific Subroutine Library (IBM 1968). Almost all its slices included every statement. The single-mindedness of its mathematical code made it difficult to slice, since all its statements directly or indirectly affected the matrix product. Only by choosing a trivial slicing criterion, such as the value of a loop control variable, could any statements be eliminated. But this does not generalize to all mathematical software, since the routine TALLY in table 1 is from the same subroutine library as MMGS.

A second slicer is now being implemented to slice programs written in the SIMPL family of languages (Basili 1976). The slice is done on a language independent representation of the program flowgraph, and extensions to other source languages are planned. Among the languages in the SIMPL family is SIMPL-D (Gannon 1979), which has abstract datatype facilities similar to CLU (Liskov 1976), MESA (Geschke, Morris, and Satterthwaite 1977), or ADA (U.S.D.O.D 1979). The slicer itself, like the SIMPL-D compiler, is written in SIMPL-D.

Incorporated into the SIMPL compiler is a program which writes to a file all of the information necessary to slice the program. The programmer need not be aware of this file until the program is to be sliced.

Interprocedural slicing

Unlike the DAVE slicer, the SIMPL slicer performs full interprocedural dataflow analysis. The most accurate interprocedural slicing algorithm requires slicing called and calling procedures repeatedly until convergence. This was actually implemented, and was very slow. The algorithms of Barth (1978) are slightly less accurate because

they do not distinguish separate calls on a procedure. But they are very fast, and are now being used. See Rosen (1979) for another discussion of interprocedural dataflow analysis.

To slice across systems of subroutines requires two steps. First, a single slice is made of the procedure containing the slicing criterion. Summary dataflow information about calls to other procedures is used, but no attempt is made to slice the other procedures. In the second step, for each procedure call which could influence variables relevant to the slice generated in step one, a new slicing criterion is created in the called procedure. Steps one and two are then repeated for each of these new criteria until no new criteria are generated.

There are two basic ways in which slicing can cross a procedure boundary. The first occurs when a procedure being sliced contains a call to another procedure. Summary information about the possible effects of the call is sufficient to continue slicing within the calling routine, but not within the called routine. The second kind of influence is going in the opposite direction--that is, when the procedure sliced is called by another procedure.

Extending a slice from a calling procedure P to a called procedure Q is done as follows. Suppose the call to procedure Q is statement number i in procedure P. Then there is some set of variables (namely ROUT(i)) relevant to the current slice of P. Recall that this set is just UNION R[C](j), for all j successors to i. C is the slicing criterion for P. ROUT(i) is easily transformed into a slicing criterion for Q by simply changing actual parameters in the call to Q which are also in ROUT(i) to their corresponding formal parameters in Q (See figure 7). Any variables not in the scope of Q are also removed from the criterion. The statement number in the new slicing criterion for Q is simply the final statement in Q.

Extending slices to called and calling routines

```
1    READ(A,B)
2    CALL Q(A,B)
3    Z := A + B

     procedure Q(var x,y : integer)
4    X := 0
5    Y := X + 3
```

Slicing on a criterion C=<3,{Z}> causes a new criterion C'=<6,{X,Y}> to be generated. The complete slice on C is {2,3,4,5}.
Slicing on a criterion D=<4,{Y}> causes a new criterion D'=<2,{B}> to be generated. The complete slice on D is {1}.

Figure 7

Extending a slice from a called procedure Q to a calling procedure P is done as follows. Let i be the initial statement of Q. Then the variables in RIN(i) form the basis of the slicing criterion for P. Local variables are removed from RIN(i), and formal parameters are replaced by actual parameters. If an actual parameter is an expression, all the variables in the expression are added to the criterion for P. The statement for the slicing criterion of P is the statement which calls Q.

Generating these additional slicing criteria and slicing on them could be very time consuming if several slices are to be found. This is especially true since the SIMPL slicer can only keep in memory enough information to slice a single subroutine at a time, and therefore has to make successive passes over the file of dataflow information. What is needed is a way of computing as many slicing criteria as possible at one time, so as not to miss any opportunities. This is done as follows.

Let PC be the set of all possible slicing criteria. For each criterion C for a procedure P, there is a set of criteria UPO(C) which are those needed to slice callers of P, and a set of criteria DOWNO(C) which are those needed to slice procedures called by P. UPO(C) and DOWNO(C) are computed by the methods outlined above. UPO and DOWNO can be extended to functions UP and DOWN which map sets of criteria into sets of criteria. Let CC be any subset of PC. Then:

UP(CC) = UNION UPO(C), for all C in CC,

DOWN(CC) = UNION DOWNO(C), for all C in CC.

The transitive closure of UP and DOWN, denoted (UP union DOWN)*, will map any set of criteria into all those criteria necessary to complete the corresponding slices through all calling and called routines. The complete inter-procedural slice for a criterion C is then just the union of the intra-procedural slices for each criterion in (UP union DOWN)*(C).

In implementing this, lists are kept of all the criteria generated so far, the intra-procedural slice for each criterion, and (UP(C) union DOWN(C)) for each criterion C on the list. Bitmaps are used for representing (UP(C) UNION DOWN(C)) and the intra-procedural slices.

This algorithm could possibly be improved by using the properties of slices mentioned in section 3. For instance, before slicing on a criterion <a,v>, the list of criteria could be checked to see if there were already criteria <a,v1>, <a,v2> such that v1 union v2 = v. Other improvements in speed at the expense of accuracy and memory might make use of the value of R from previous slices, together with property B from section 3, to avoid recomputing slices. This seems to have the potential for eliminating quite a bit of slicing work, at the expense of remembering the value of R for all slices.

None of these tricks have been implemented in the current SIMPL slicer. It remains to be seen if slow slicing speeds will compell the use of such speed-up heuristics.

Separate compilation

SIMPL-D allows separate compilation of modules, and this complicates interprocedural slicing in two

respects. First, calls on separately compiled routines are assumed to both reference and change any variable known outside the current compilation. This worst case assumption ensures that slices are at least as large as necessary. The second complication is from procedures in the current compilation which can be called from some other compilation. These are known as "entry" procedures.

The worst case assumption for entry procedures is that there is an externally compiled program which calls them in every possible order, and between each call references and changes all variables used as parameters and all variables known outside the current compilation. The worst case assumption therefore implies a certain dataflow between entry procedures. As with called and calling procedures, this dataflow causes a slice for one entry procedure to generate slicing criteria for other entry procedures.

Let ENTO be a function which maps a criterion into the set of criteria possible under the above worst case assumption. Specifically, ENTO(C) is empty unless C is a criterion for an entry procedure P, in which case ENTO is computed as follows: Let i be the unique initial statement in P, let EE be the set of all entry procedures, let OUT be the set of all variables known outside the compilation, and for each E in EE let f(E) be the unique final statement in E. Then: ENTO(C) = {<f(E), R[C](i) union OUT> : for all E in EE}.
ENTO can be extended to a function ENT which maps sets of criteria into sets of criteria in the same manner as UP and DOWN.

Of course, it is now a simple matter to include the entry criteria in the interprocedural slicing algorithm. ENT need only be unioned with UP and DOWN when taking the transitive closure of generated slices.

Slicing Based Metrics

There are two investigations now in progress on the practical use of slicing. One is to use slicing in a program debugging and maintenance aid. Programmers will be able to interactively obtain slices of programs, and so use this information in making program changes or to look for bugs. Comparison will be made of programmer performance with and without a slicer. This should help establish whether or not slicing aids are useful programming tools. The second practical use being looked at is slicing-based program metrics. Numbers of slices, their spatial arrangement, etc., may hold significant information about the structuring of a program. Since programmers do already break programs into slices, slicing-based metrics may be particularly meaningful, compared to measures such as McCabe's (1976) or Halstead's (1977).

Some possible slicing-based metrics are:

1. "Coverage" compares the length of slices to the length of the entire program. Coverage might be expressed as the ratio of mean slice length to program length. A low coverage value, indicating a long program with many short slices, may indicate a program which has several distinct conceptual purposes.

2. "Overlap" is a measure of how many statements in a slice are found only in that slice. This could be computed as the mean of the ratios of non-unique to unique statements in each slice. A high overlap might indicate very interdependent code.

3. "Clustering" reveals the degree to which slices are reflected in the original code layout. It could be expressed as the mean of the ratio of statements formerly adjacent to total statements in each slice. A low cluster value indicates slices intertwined like spaghetti, while a high cluster value indicates slices physically reflected in the code by statement grouping.

4. "Parallelism" is the number of slices which have few statements in common. Parallelism could be computed as the number of slices which have a pair wise overlap less than a certain threshold. A high degree of parallelism would suggest that assigning a processor to execute each slice in parallel could give a significant program speed-up.

5. "Tightness" measures the number of statements which are in every slice, expressed as a ratio over the total program length. The presence of relatively high tightness might indicate that all the slices in a subroutine really belonged together because they all shared certain activities.

Slicing-based metrics are now being applied to several student-written load-and-go compilers. Much more work is needed, but the following are some initial results and conclusions.

Slicing on every output statement leads to a great many similar slices. Clustering together slices which differ by only a few statements gives a more meaningful set of slices on which to apply metrics.

For instance, the 48 output statements in one compiler could be clustered into seven categories:

1) object code interpretation messages and errors
2) source code scanning messages and errors
3) miscellaneous fixed messages (e.g. "EXECUTION BEGINS")
4) global error messages (e.g. "NO PROCEDURES")
5) symbol table listing
6) object code listing
7) symbol table error messages

Slices within a cluster differed by less than seven statements, while inter-cluster differences were between 30 and 400 statements, with most more than 100. There were about 500 executable statements altogether, divided among 21 procedures and functions.

The cost of finding these 48 slices is interesting. Together, the 48 original slicing criteria generated 217 additional slicing criteria as a

result of called and calling procedures. (There were no entry procedures.) The entire process of finding the criteria, propogating the slices interprocedurally, and calculating the 217 intra-procedural slices took about 10 minutes of CPU time on a Univac 1100/40. Six passes over the file of dataflow information were necessary.

There were certain core groups of statements which showed up in many different slices. For instance, in the same compiler as above, a set of 115 statements showed up in every non-trivial slice. These statements were drawn mostly from the scanner, with a few from the parser and symbol table sections of code. The 115 statements are not themselves a slice, because they do not constitute an independently executable program. But they form the core of a stripped down compiler, since any non-trivial output requires that they be executed.

Conclusion

Slicing is a new way of decomposing programs automatically. Limited to code already written, it may prove useful during the debugging, testing, and maintenance portions of the software life-cycle. Unlike design methodologies which decompose a program in just one way, many different slicing decompositions can be chosen by selecting appropriate slicing criteria. This paper concentrated on the basic methods for slicing programs and their embodiment in automatic slicers. Future work on slicing-based programming aids and slicing-based program metrics is necessary before the implications of this kind of decomposition are fully known.

References

1. Aho, A.V. and Ullman, J.D. 1977 Principles of Compiler Design. Addison-Wesley, 1977.
2. Arsac, J.J. 1979 Syntactic Source to Source TransformaHs and Program Manipulation. CACM 22, 1 (Jan. 1979) pp. 43-53.
3. Ashcroft and Manna 1973 The Translation of GOTO Programs into WHILE programs. Information Processing 71, North Holland Pub. Co. Amsterdam, pp. 250-255.
4. Aygun, B.O. 1973 Dynamic analysis of execution: possibilities, techniques, and problems. PhD thesis, Carnegie-Mellon University Sept. 1973.
5. Baker, B. 1977 An Algorithm for Structuring Flowgraphs. JACM 24, 1 (Jan. 1977) pp. 98-120.
6. Barth, J.M. 1978 A Practical Interprocedural Dataflow Analysis Algorithm. CACM 21, 9 (Sept. 1978) pp. 724-736.
7. Basili, V.R. 1976 The design and implementation of a family of application-oriented languages. Fifth Texas Conference on Computing Systems. pp. 6-12.
8. Browne, J.C. and Johnson, D.B. 1978 FAST: A second generation program analysis system. Third int'l conference on software engineering. IEEE catalog no. 78CH1317-7C. May 1978. pp. 142-148.
9. Denning, D.E. and Denning, P.J. 1977 Certification of programs for secure information flow. CACM 20, 7 (July 1977) pp. 504-513.
10. Fosdick. L.D. and Osterweil, L.J. 1976 Data Flow Analysis in Software Reliability. ACM Computing Surveys 8, 3 (Sept. 1976) pp. 305-330.
11. Gannon, J.D. and Rosenberg, J. 1979 Implementing data abstraction features in a stack-based language. Software-Practice and Experience, Vol. 9, pp. 547-560.
12. Gerhart, S. 1975 Correctness Preserving Program Transformations. Second Conference on the Principles of Programming Languages. ACM (Jan. 1975) pp. 54-66.
13. Gould, J.D. and Drongowski, P. 1974 An Exploratory Study of Computer Program Debugging. Human Factors 1, 6. pp. 258-277.
14. Graham, S.L., and Wegman, M. 1976 A fast and usually linear algorithm for global flow analysis. JACM 23, 1. January 1976, pp. 172-202.
15. Halstead, M. 1977 Elements of Software Science. Elsevier Computer Science Library. 1977.
16. Hecht, M.S. 1977 Flow Analysis of Computer Programs. North-Holland (1977).
17. IBM 1968 Scientific Subroutine Package (PL/I). 360-ACM-07X. Program Description and Operations Manual. Form 6H20-0586-0.
18. Lengauer, T. and Tarjan, R.E. 1979 A fast algorithm for finding dominators in a flowgraph. ACM T. on Prog. Lan. and Systems, Vol 1, no. 1 July 1979, pp. 121-141.
19. Liskov, B.H. and Zilles, S.N. 1975 Specification techniques for data abstractions. IEEE Trans of Software Engineering. March 1975.
20. Loveman, D.B. 1977 Program Improvement by Source to Source Transformation. JACM 24, 1 (Jan. 1977) pp. 121-145.
21. Luckham, D.C. and Suzuki, N. 1979 Verification of array, record, and pointer operations in Pascal. ACM T. on Prog. Lan. and Systems, Vol. 1,

no. 2 Oct. 1979, pp. 226-244.

21. McCabe, Thomas J.
1976 A Complexity Measure. IEEE Transactions on Software Engineering. SE-2,

22. Parnas, D.L.
1972 On the criteria used in decomposing systems into modules. CACM 15, 12 (Dec. 1972) pp. 1053-1058.

23. Schwartz, J.T.
1971 An overview of bugs. in Debugging techniques in large systems. Rustin, Randall, ed. Prentice-Hall.

24. Shneiderman, B.
1976a Exploratory Experiments in Programmer Behavior. International J. of Computer and Information Sciences, 5, 2.

25. Stay, J.F.
1976 HIPO and integrated program design. IBM systems journal.

26. Tarjan, R.E. and Valdes, J.
1980 Prime subprogram parsing of a program. Seventh annual ACM symposium on the principles of programming languages. Jan. 1980, pp. 95-105.

27. U.S.D.O.D.
1979 Preliminary Ada reference manual and rationale. Sigplan notices 14, 6.

28. Wegbreit, Ben
1976 Goal-directed program transformation IEEE Transactions on Software Engineering. Vol SE-2, 2 (June 1976) pp. 69-80.

29. Weihl, W.E.
1980 Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. Seventh ACM Symposium on the Principles of Programming Languages pp. 83-94.

30. Weiser, M.D.
1979 Program Slices: Foraml Psychological, and Practical Investigations of an Automatic Program Abstraction Method. Ph.D. Thesis, Computer and Communication Sciences Dept., University of Michigan.

31. Weiser, M.D.
1980 Color dominance: a new graph coloring program with applications to computer program optimization. In preparation.

32. Zelkowitz, M.W., Shaw, A.C., and Gannon, J.D.
1979 Principles of software engineering and design. Prentice-Hall.