Program Slicing

MARK WEISER

Abstract—Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. 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 represent faithfully the original program within the domain of the specified subset of behavior.

Some properties of slices are presented. In particular, finding *statement-minimal* slices is in general unsolvable, but using data flow analysis is sufficient to find approximate slices. Potential applications include automatic slicing tools for debugging and parallel processing of slices.

Index Terms—Data flow analysis, debugging, human factors, parallel processing, program maintenance, program metrics, slicing, software tools.

Introduction

LARGE computer programs must be decomposed for understanding and manipulation by people. Not just any decomposition is useful to people, but some—such as decomposition into procedures and abstract data types—are very useful. Program slicing is a decomposition based on data flow and control flow analysis.

A useful program decomposition must provide pieces with predictable properties. For instance, block-structured languages [17] are powerful in part because their scope and control flow rules permit understanding procedures independent of context. Similarly, abstract data type languages [12], [15], [25] make further control and scope restrictions for even greater context independence. Therefore, the pieces of a program decomposed by dataflow, i.e., the "slices," should be related to one another and the original program in well defined and predictable ways.

As we will see, slices have a very clear semantics based on projections of behavior from the program being decomposed. Unlike procedures and data abstractions, slices are designed to be found automatically after a program is coded. Their usefulness shows up in testing, parallel processor distribution, maintenance, and especially debugging. A previous study showed experienced programmers mentally slicing while debugging, based on an informal definition of slice [22]. Our concern here is with 1) a formal definition of slices and their abstract properties, 2) a practical algorithm for slicing, and 3) some experience slicing real programs.

Manuscript received August 27, 1982; revised June 28, 1983. This work was supported in part by the National Science Foundation under Grant MCS-80-18294 and by the U.S. Air Force Office of Scientific Research under Grant F49620-80-C-001. A previous version of this paper was presented at the 5th International Conference on Software Engineering, San Diego, CA, 1981.

The author is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

Definitions

This section considers programs without procedure calls. Procedures are discussed later. The first few definitions review the standard definitions of digraph, flowgraph, and computation in terms of state trajectory. Finally, a slice is defined as preserving certain projections from state trajectories.

The next few definitions simply establish a terminology for graphs, and restrict attention to programs whose control structure is single-entry single-exit ("hammock graphs").

Definition: A *digraph* is a structure $\langle N, E\rangle$, where N is a set of nodes and E is a set of edges in $N \times N$. If (n, m) is in E, then n is an *immediate predecessor* of m and m is an *immediate successor* of n. A *path* from n to m of length k is a list of nodes $p_0, p_1, \cdots, p_k$ such that $p_0 = n$, $p_k = m$, and for all i, $1 \leqslant i \leqslant k - 1$, (p_i, p_{i+1}) is in E.

Definition: A *flowgraph* is a structure $\langle N, E, n_0\rangle$, where $\langle N, E\rangle$ is a digraph and n_0 is a member of N such that there is a path from n_0 to all other nodes in N. n_0 is sometimes called the initial node. If m and n are two nodes in N, m *dominates* n if m is on every path from n_0 to n.

Definition: A *hammock graph* is a structure $\langle N, E, n_0, n_e\rangle$ with the property that $\langle N, E, n_0\rangle$ and $\langle N, E^{-1}, n_e\rangle$ are both flowgraphs. Note that, as usual, $E^{-1} = \{(a, b) | (b, a) \text{ is in } E\}$. If m and n are two nodes in N, m *inverse dominates* n if m is on every path from n to n_e.

In the remainder of the paper, all flowgraphs will be assumed to be hammock graphs. In addition to its flowgraph, every program is assumed to provide the following information.

Definition: Let V be the set of variable names which appear in a program P. Then for each statement n in P (i.e., node in the flowgraph of P) we have the following two sets, each a subset of V: $REF(n)$ is the set of variables whose values are used at n, and $DEF(n)$ is the set of variables whose values are changed at n.

A state trajectory of a program is just a trace of its execution which snapshots all the variable values just before executing each statement.

Definition: A *state trajectory* of length k of a program P is a finite list of ordered pairs

$$(n_1, s_1)(n_2, s_2) \cdots (n_k, s_k)$$

where each n is in N (the set of nodes in P) and each s is a function mapping the variables in V to their values. Each (n, s) gives the values of V immediately before the execution of n. Our attention will be on programs which halt, so infinite state trajectories are specifically excluded.

Slices reproduce a projection from the behavior of the original program. This projection must be the values of certain variables as seen at certain statements.

0098-5589/84/0700-0352$01.00

```
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 criterion <12,{Z}>.
           BEGIN
           READ(X,Y)
           IF X <= 1
              THEN
              ELSE READ(Z)
           END.
Slice on criterion <9,{X}>.
           BEGIN
           READ(X,Y)
           END.
Slice on criterion <12,{TOTAL}>.
           BEGIN
           READ(X,Y)
           TOTAL := 0.0
           IF X <= 1
              THEN
              ELSE TOTAL := X*Y
           END.
```

Fig. 1. Examples of slices.

Definition: A *slicing criterion* of a program P is a tuple $\langle i, V\rangle$, where i is a statement in P and V is a subset of the variables in P.

A slicing criterion $C = \langle i, V\rangle$ determines a projection function Proj_C which throws out of the state trajectory all ordered pairs except those starting with i, and from the remaining pairs throws out everything except values of variables in V.

Definition: Let $T = (t_1, t_2, \cdots, t_n)$ be a state trajectory, n any node in N and s any function from variable names to values. Then

$$\text{Proj}'_{\langle i,V\rangle}((n,s)) = \begin{cases} \lambda & \text{if } n \neq i \\ (n, s|V) & \text{if } n = i \end{cases}$$

where $s|V$ is s restricted to domain V, and λ is the empty string. Proj′ is now extended to entire trajectories:

$$\text{Proj}_{\langle i,V\rangle}(T) = \text{Proj}'_{\langle i,V\rangle}(t_1) \cdots \text{Proj}'_{\langle i,V\rangle}(t_n).$$

A slice is now defined behaviorally as any subset of a program which preserves a specified projection of its behavior.

Definition: A slice S of a program P on a slicing criterion $C = \langle i, V\rangle$ is any executable program with the following two properties.

1) S can be obtained from P by deleting zero or more statements from P.

2) Whenever P halts[1] on an input I with state trajectory T, then S also halts on input I with state trajectory T', and $\text{Proj}_C(T) = \text{Proj}_{C'}(T')$, where $C' = \langle \text{succ}(i), V\rangle$, and succ($i$) is the nearest successor to i in the original program which is also in the slice, or i itself if i is in the slice.

There can be many different slices for a given program and slicing criterion. There is always at least one slice for a given slicing criterion—the program itself. Fig. 1 gives some examples of slices.

Finding Slices

The above definition of a slice does not say how to find one. The smaller the slice the better, but the following argument shows that finding minimal slices is equivalent to solving the halting problem—it is impossible.

Definition: Let C be a slicing criterion on a program P. A slice S of P on C is *statement-minimal* if no other slice of P on C has fewer statements than S.

Theorem: There does not exist an algorithm to find statement-minimal slices for arbitrary programs.

Informal Proof: Consider the following program fragment:

```
1 read (X)
2 if (X)
        then
        ...
        perform any function not involving x here
        ...
3       X := 1
4 else  X := 2 endif
5 write (X)
```

Imagine slicing on the value of x at line 5. An algorithm to find a statement-minimal slice would include line 3 if and only if the function before line 3 did halt. Thus such an algorithm could determine if an arbitrary program could halt, which is impossible.

A similar argument demonstrates that statement-minimal slices are not unique.

More interesting are slices that *can* be found. Data flow analysis can be used to construct conservative slices, guaranteed to have the slice properties but with possibly too many statements. The remainder of this section outlines how this is done. To avoid repetition, an arbitrary program P with nodes N and variables V is assumed.

In general, for each statement in P there will be some set of variables whose values can affect 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. X is said to be directly "relevant" to the slice at statement n. (See Fig. 2.) The set of all such relevant variables is denoted R_C^0, and defined below.

[1] Extending this definition of slice to inputs on which the original program does not halt causes many new problems. For example, the proof of Theorem 1 below demonstrates that there is no way to guarantee that a slice will fail to halt whenever the original program fails to halt.

(from definition of R^0_C):

```
1       Y := X
2       A := B
3       Z := Y
```

$R^0_{\langle 3,\{Y\}\rangle}(3)$ = {Y} by rule 1.

$R^0_{\langle 3,\{Y\}\rangle}(2)$ = {Y} by rule 2b.

$R^0_{\langle 3,\{Y\}\rangle}(1)$ = {X} by rule 2a.

Fig. 2. Definition of direct influence.

The superscript 0 indicates how indirect the relevance is; higher valued superscripts are defined later.

Definition: Let $C = \langle i, V\rangle$ be a slicing criterion. Then

$R^0_C(n)$ = 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 the length of paths to reach node i, where (1) is the base case. Case (2a) says that if w is a relevant variable at the node following n and w is given a new value at n, then w is no longer relevant and all the variables used to define w's value are relevant. Case (2b) says that if a relevant variable at the next node is not given a value at node n, then it is still relevant at node n. This is a simplification of the usual data flow information which would use a PRE set to represent preservation of variable values.

The author has previously proved [20] that the computation of R^0_C can be imbedded in a fast monotone information propagation space [11], and so can be computed in time $0(e \log e)$ for arbitrary programs and time $0(e)$ for structured programs where e is the number of edges in the flowgraph.

The statements included in the slice by R^0_C are denoted S^0_C. S^0_C is defined by

$$S^0_C = \text{all nodes } n \text{ s.t. } R^0_C(n+1) \cap \text{DEF}(n) \neq \phi.$$

Note that R^0_C is a function mapping statements to sets of variables, but S^0_C is just a set of statements.

S^0_C does not include indirect effects on the slicing criterion, and therefore is a sufficient but not necessary condition for including statements in the slice. For instance, in the following program statement 2 obviously has an affect on the value of Z at statement 5, yet 2 is not in $S^0_{\langle 5,\{Z\}\rangle}$.

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

Generally any branch statement which can choose to execute or not execute some statement in S^0_C should also be in the slice. Denning and Denning [8] use the nearest inverse dominator of a branch to define its range of influence.

Definition: INFL (b) is the set of statements which are on a path P from b to its nearest inverse dominator d, excluding the endpoints of P.

INFL(b) will be empty unless b has more than one immediate successor (i.e., is a branch statement).

INFL allows the following definition of branch statements with indirect relevance to a slice.

Definition:

$$B^0_C = \bigcup_{n \in S^0_C} \text{INFL}(n).$$

To include *all* indirect influences, the statements with direct influence on B^0_C must now be considered, and then the branch statements influencing those new statements, etc. The full definition of the influence at level n is the following.

Definition: For all $i \geq 0$:

$$R^{i+1}_C(n) = R^i_C(n) \bigcup_{b \in B^i_C} R^0_{BC(b)}(n)$$

$$B^{i+1}_C = \bigcup_{n \in S^{i+1}_C} \text{INFL}(n)$$

$$S^{i+1}_C = \text{all nodes } n \text{ s.t. } n \in B^i_C \text{ or } R^{i+1}_C(n+1) \cap \text{DEF}(n) \neq \phi$$

where $BC(b)$ is the branch statement criterion, defined as $\langle b, \text{REF}(b)\rangle$.

Considered as a function of i for fixed n and C, R^i_C and S^i_C define nondecreasing subsets and are bounded above by the set of program variables and set of program statements, respectively. Therefore, each has a least fixed point denoted R_C and S_C, respectively.

It is easy to see that S_C and R_C have the following combining property:

$$S_{\langle i,A\rangle} \cup S_{\langle i,B\rangle} = S_{\langle i, A \cup B\rangle}$$

$$R_{\langle i,A\rangle} \cup R_{\langle i,B\rangle} = R_{\langle i, A \cup B\rangle}.$$

An upper bound on the complexity of computing S is estimated as follows: each computation of S^{i+1}_C from S^i_C requires an initial $0(e \log e)$ step to compute R. Followed by a computation of B^{i+1}_C. Finding B^{i+1}_C is primarily finding dominators, an almost linear task [14]. Hence each step takes $0(e \log e)$ time. Since one statement must be added each iteration, the total number of steps is at most n. Hence the total complexity is $0(n\ e \log e)$. This bound is probably not tight, since practical times seem much faster.

S_C is not always the "smallest" slice which can found using only dataflow analysis. Fig. 3 gives a counter example. However, the author has proven that only anomolous cases like Fig. 3 will make S give a less than data flow smallest slice [20].

```
1    A := constant
2    WHILE P(k) DO
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)

Slicing criterion c=<10,{Z}>.
```

Fig. 3. A special case for data flow slicing. Statement 1 will be in S_c, but cannot affect the values of Z at statement 10. It cannot because any path by which A at 1 can influence C at 3 will also execute both statements 5 and 7, resulting in a constant value for Z at line 10. Hence, statement 1 can have no effect on the value of Z at 10, and should not be in the slice. Note that this argument is not semantic, but requires knowledge only of the flowgraph, REF, and DEF sets for each statement.

Interprocedural Slicing

If a slice originates in a procedure which calls or is called by other procedures, then the slice may need to preserve statements in the calling or called procedures. Our method of slicing across procedure boundaries requires two steps. First, a single slice is made of the procedure P containing the slicing criterion. Summary data flow information about calls to other procedures is used [5], but no attempt is made to slice the other procedures. In the second step slicing criteria are generated for each procedure calling or called by P. Steps one and two are then repeated for each of these new slicing criteria. The process stops when no new slicing criteria are seen, and this must happen eventually since a program has only a finite number of slicing criteria.

The generation of new criteria is straightforward. In each case (caller or callee) the hard work is translating the set of variables computed by R_C into the scope of the new procedure. Suppose procedure P is being sliced, and P has a call at statement i to procedure Q. The criterion for extending the slice to Q is

$$\langle n_e^Q, \text{ROUT}(i)_{F \to A} \cap \text{SCOPE}_Q \rangle$$

where n_e^Q is the last statement in Q, $F \to A$ means substitute formal for actual parameters, SCOPE_Q is the set of variables accessible from the scope of Q, and

$$\text{ROUT}(i) = \bigcup_{j \in \text{Succ}(i)} R_C(j).$$

Alternatively, again suppose P is being sliced, and now suppose P is called at statement i from procedure Q. The new criterion is then

$$\langle i, R_C(f_p)_{A \to F} \cap \text{SCOPE}_Q \rangle$$

where f_P is the first statement in P, $A \to F$ means substitute actual for formal parameters, and SCOPE_Q is as before.

For each criterion C for a procedure P, there is a set of criteria

```
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
6       RETURN


DOWN(<3,{Z}>) = {<6,{X,Y}>}

UP(<4,{Y}> = {<2,{B}>}
```

Fig. 4. Extending slices to called and calling routines.

$\text{UP}_0(C)$ which are those needed to slice callers of P, and a set of criteria $\text{DOWN}_0(C)$ which are those needed to slice procedures called by P. $\text{UP}_0(C)$ and $\text{DOWN}_0(C)$ are computed by the methods outlined above (see Fig. 4). UP_0 and DOWN_0 can be extended to functions UP and DOWN which map sets of criteria into sets of criteria. Let CC be any set of criteria. Then

$$\text{UP}(CC) = \bigcup_{C \in CC} \text{UP}_0(C)$$

$$\text{DOWN}(CC) = \bigcup_{C \in CC} \text{DOWN}_0(C).$$

The union and transitive closure of UP and DOWN are defined in the usual way for relations. (UP ∪ 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 interprocedural slice for a criterion C is then just the union of the intraprocedural slices for each criterion in (UP ∪ DOWN)*(C).

This algorithm could possibly be improved by using the properties of slices mentioned above. For instance, before slicing on a criterion $\langle a, v \rangle$, the list of criteria could be checked to see if there were already criteria $\langle a, v1 \rangle$, $\langle a, v2 \rangle$ 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 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.

No speed-up tricks have been implemented in a current slicer. It remains to be seen if slow slicing speeds will compel the use of speed-up heuristics.

Separate Compilation

Slicing a program which calls external procedures or which can be called externally creates special problems for computing slices. Assuming the actual external code is unavailable, worst case assumptions must be made. First, calls on external routines must be assumed to both reference and change any external variable. This worst case assumption ensures that slices are at least as large as necessary.

The worst case assumption for procedures called externally (sometimes called "entry" procedures) is that the calling program calls them in every possible order, and between each call

references and changes all variables used as parameters and all external variables. The worst case assumption therefore implies a certain data flow between entry procedures. As with called and calling procedures, this data flow causes a slice for one entry procedure to generate slicing criteria for other entry procedures.

Let ENT_0 be a function which maps a criterion into the set of criteria possible under the above worst case assumption. Specifically, $ENT_0(C)$ is empty unless C is a criterion for an entry procedure P, in which case ENT_0 is computed as follows: let n_0 be the unique initial statement in P, let EE be the set of all entry procedures, let OUT be the set of all external variables, and for each E in EE let n_e^E be the unique final statement in E and F^E be the set of ref parameters to E. Then

$$ENT_0(C) = \{\langle n_e^E, R_C(i) \cup \text{OUT} \cup F^E\rangle | \text{for all } E \text{ in } EE\}.$$

ENT_0 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. (UP ∪ DOWN ∪ ENT)*(C) is the total set of criteria needed to slice from an initial criterion C. Notice that computing this set requires slicing the program.

A Sampling of Slices

Program slicers have been built at the University of Maryland for several different languages, including Fortran and the abstract data type language Simpl-D [9]. To look at typical slices of programs larger than toy examples, slices were made of the 19 load-and-go compilers used in the Basili and Reiter study [6]. These compilers were student projects, written by both individuals and teams, between 500 and 900 executable statements long, with between 20 and 80 subroutines.

The compilers were sliced as follows. For each write statement i which output the values of a set of variables V, a slice was taken on the criterion $\langle i, V\rangle$. Slices that differed by less than 30 statements were then merged into a new slightly larger slice. Merging continued until all slices differed by at least 30 statements.

Slicing was done automatically by a system using an abstract data type for flow analysis [23]. Finding all the output related slices for all compilers took approximately 36 hours of CPU time on a VAX-11/780.[2]

Some basic statistics about the slices are shown in Table I.

Useless:	Number of statements not in any slice.
Common:	Number of statements in all slices.
Slices:	Number of slices per program (this is also the number of output statements).
Clusters:	Number of slices after merging slices with less than 30 statements difference.
Contig:	Length of a run of contiguous statements in a cluster which were contiguous in the original program.
% Size:	Size of cluster as a percentage of total program size, as measured by counting statements.
% Unique:	Number of cluster statements which are in *no* other cluster, expressed as a percentage of cluster size.
% Overlap:	Pairwise sharing of statements between clusters, expressed as a percentage of cluster size.

[2]The slicer was compiled by an early compiler which generated particularly bad object code. A better compiler could probably cut the slicing time by a factor of 10.

TABLE I
STATISTICS ON SLICES

Measure	Mean	Median	Min	Max
Per program measures $N = 19$				
Useless	9.16	6	1	23
Common	14.32	0	0	86
Slices	37.26	32	7	74
Clusters	9.74	7	3	25
Per cluster measures $N = 185$				
Contig	11.78	9.10	0	65.4
% Size	44	40	0	97
% Unique	6	1	0	100
% Overlap	52	51	0	93

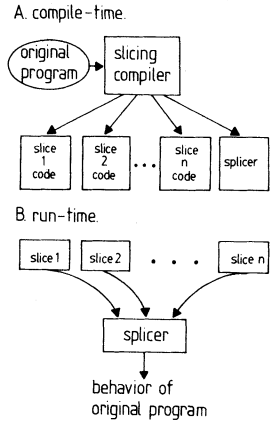


Fig. 5. Parallel execution of slices.

The useless statements were usually either subroutine stubs which immediately returned or loops which computed statistics never written out. The number of statements in a contiguous run is a measure of the scattering of slices through the code. The average of 11.8 shows that components of the original program show up fairly often in slices. The low uniqueness of slices reflects the high degree of interrelatedness of compilers, as does the pairwise overlap.

Parallel Execution of Slices

Because slices execute independently they are suitable for parallel execution on multiprocessors without synchronization or shared memory. Each slice will produce its projection of the final behavior, and one or more "splicing" programs will fit these projections back together into the original program's total behavior (see Fig. 5).

Splicers work in real time (i.e., produce immediate output for every input without delay), so introduce only communications overhead. Splicers require occasional additional output from each slice, and use knowledge of the path expressions corresponding to each slice to properly piece together the slices' output. Splicers can be cascaded, with a few splicers merging the slice output and then a splicer merging splicer output. Details on splicers are described elsewhere [24].

Slices avoid the need for shared memory or synchronization by duplicating in each slice any computation needed by that slice. Although total CPU cycles among all processors are wasted this way, the time to receive an answer is not delayed.

If no computation was duplicated, processors could not proceed until some other processor produced needed immediate results.

Parallel execution of slices might be particularly appropriate for distributed systems, where shared memory is impossible and synchronization requires excessive handshaking. The one way flow of data from slices to splicers mean interprocessor communication is a tree, simplifying VLSI multiprocessor design for parallel execution of slices.

Previous Work

Isolating portions of programs according to their behavior has has been discussed previously. Schwartz [18] hints at such a possibility for a debugging system. Brown and Johnson [7] describe a database for Fortran programs 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 [37]), and this information is a kind of "dynamic slice."

Slicing is a source-to-source transformation of a program. Previous work in program transformation has concentrated on preserving program correctness while improving some desirable property of programs. Baker [4] and Ashcroft and Manna [2] try to add "good structure" to programs. Wegbreit [19], Arsac [1], Gerhart [10], and Loveman [16] try to improve program performance. King [13] suggests using input domain restrictions to eliminate statements from a program. This is close in spirit to slicing, which uses projections from the output domain to eliminate statements.

Future Directions

The power of slices comes from four facts: 1) they can be found *automatically*, 2) slices are generally *smaller* than the program from which they originated, 3) they execute *independently* of one another, and 4) each *reproduces exactly* a projection of the original program's behavior. The independence of slices suggests their use in loosely coupled multiprocessors. The simple relationship between a slice's semantics and the original program's semantics makes slices useful for decomposing any semantical operation, such as program verification or testing. The automatic nature of slicing and its data flow origin suggest basing program complexity metrics on slices. Finally, the small size of slices means people may find them directly understandable and useful.

The problems with slices are: 1) they can be expensive to find, 2) a program may have no significant slices, and 3) their total independence may cause additional complexity in each slice that could be cleaned up if simple dependencies could be represented. However, large classes of programs have significant, easy to find, and revealing slices.

Acknowledgment

B. Rounds encouraged the initial formalization of these ideas. B. Riddle guided the initial application of slicing to real problems.

References

[1] J. J. Arsac, "Syntactic source to source transformations and program manipulation," *Commun. ACM*, vol. 22, pp. 43–53, Jan. 1979.

[2] E. A. Ashcroft and Z. Manna, *The Translation of Goto Programs into While Programs*. Amsterdam, The Netherlands: North-Holland, 1973. Information Processing 71.

[3] B. O. Aygun, "Dynamic analysis of execution–possibilities, techniques, and problems," Ph.D. dissertation, Comput. Sci., Dep. Carnegie-Mellon Univ., Tech Rep. CMU-083, Sept. 1973.

[4] B. Baker, "An algorithm for structuring flowgraphs," *J. Ass. Comput. Mach.*, vol. 24, pp. 98–120, Jan. 1977.

[5] J. M. Barth, "A practical interprocedural dataflow analysis algorithm," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 724–736, Sept. 1978.

[6] V. R. Basili and R. W. Reiter, "An investigation of human factors in software development," *Comput.*, vol. 12, pp. 21–38, Dec. 1979.

[7] J. C. Browne and D. B. Johnson, "FAST: A second generation program analysis system," in *Proc. Third Int. Conf. Software Eng.*, pp. 142–148, May 1978. IEEE Catalog 78CH1317-7C.

[8] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 504–513, July 1977.

[9] J. D. Gannon and J. Rosenberg, "Implementing data abstraction features in a stack-based language," *Software-Practice and Experience 9*, pp. 547–560, 1979.

[10] S. Gerhart, "Correctness preserving program transformations," in *Proc. ACM Second Conf. Principles of Programming Languages*, pp. 54–66, Jan. 1975.

[11] S. L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," *J. Ass. Comput. Mach.*, vol. 23, pp. 172–202, Jan. 1976.

[12] J. D. Ichbiah *et al.*, "Preliminary Ada reference manual and rationale," *Sigplan Notices*, vol. 14, no. 6, 1979.

[13] J. King, "Program reduction using symbolic evaluation," *ACM SIGSOFT, Software Engineering Notes 6*, Jan. 1, 1981.

[14] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Programming Languages and Systems*, vol. 1, pp. 121–141, July 1979.

[15] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Commun. ACM*, vol. 20, pp. 564–576, Aug. 1977.

[16] D. B. Loveman, "Program improvement by source to source transformation," *J. Ass. Comput. Mach.*, vol. 24, pp. 121–145, Jan. 1977.

[17] T. W. Pratt, *Programming Languages: Design and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[18] J. T. Schwartz, "An overview of bugs," in *Debugging Techniques in Large Systems*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1971.

[19] B. Wegbreit, "Goal-directed program transformation," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 69–80, June 1976.

[20] M. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, Univ. Michigan, Ann Arbor, MI, 1979.

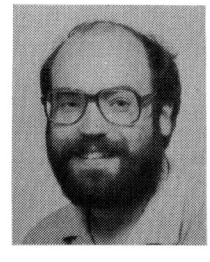
[21] —, "Program slicing," in *Proc. Fifth Int. Conf. Software Eng.*, San Diego, CA, Mar. 1981.

[22] —, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, pp. 446–452, July, 1982.

[23] —, "Experience with a data flow datatype," *J. Comput. Languages*, to be published, 1983.

[24] —, "Reconstructing sequential behavior from parallel behavior projections," *Inform. Processing Lett.*, vol. 17, pp. 129–135, Oct. 5, 1983.

[25] W. A. Wulf, R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 253–265, 1976.



Mark Weiser received the M.S. and Ph.D. degrees in computer and communication sciences from the University of Michigan, Ann Arbor, in 1976 and 1979, respectively.

He worked as a Systems Programmer and Programming Manager before attending graduate school. He is presently an Assistant Professor with the Department of Computer Science, University of Maryland, College Park. His research interests include human factors of software systems, programming environments, and parallel processing architectures.