Using Program Slicing in Software Maintenance

Keith Brian Gallagher and James R. Lyle, *Member, IEEE*

Abstract—**Program slicing, introduced by Weiser, is known to help programmers in understanding foreign code and in debugging. We apply program slicing to the maintenance problem by extending the notion of a program slice (that originally required both a variable and line number) to a *decomposition slice*, one that captures all computation on a given variable; i.e., is independent of line numbers. Using the lattice of single variable decomposition slices ordered by set inclusion, we demonstrate how to form a slice-based decomposition for programs. We are then able to delineate the effects of a proposed change by isolating those effects in a single component of the decomposition. This gives maintainers a straightforward technique for determining those statements and variables which may be modified in a component and those which may not. Using the decomposition, we provide a set of principles to prohibit changes which will interfere with unmodified components. These semantically consistent changes can then be merged back into the original program in linear time. Moreover, the maintainer can test the changes in the component with the assurance that there are no linkages into other components. Thus decomposition slicing induces a new software maintenance process model which eliminates the need for regression testing.**

Index Terms—**Software maintenance, program slicing, decompostion slicing, software process models, software testing, software tools, impact analysis.**

I. Introduction

In "Kill that Code!" [32], G. Weinberg alludes to his private list of the world's most expensive program errors. The top three disasters were caused by a change to exactly one line of code: "Each one involved the change of a *single digit* in a previously correct program." The argument goes that since the change was to only one line, the usual mechanisms for change control could be circumvented. And, of course, the results were catastrophic. Weinberg offers a partial explanation: "Unexpected linkages," i.e., the value of the modified variable was used in some other place in the program. The top three of this list of ignominy are attributed to linkage. More recently, Schneidewind [30] notes that one of the reasons that maintenance is difficult is that it is hard to determine when a code change will affect some other piece of code. We present herein a method for maintainers to use that addresses this issue.

While some may view software maintenance as a less intellectually demanding activity than development, the central premise of this work is that software maintenance is *more* demanding. The added difficulty is due in large part to the semantic constraints that are placed on the maintainer. These constraints can be loosely characterized as the attempt to avoid unexpected linkages. Some [4], [14] have addressed this problem by attempting to eliminate these semantic constraints and then providing the maintainer with a tool which will pinpoint potential inconsistencies after changes have been implemented. This makes maintenance appear to be more like development, since the programmer does not need to worry about linkages: Once the change is made, the tool is invoked and the inconsistencies (if any) are located. One would expect that the tool would proceed to resolve these inconsistencies, but it has been shown that this problem is NP-hard [14]. Thus the maintainer can be presented with a problem which is more difficult to resolve than the original change.

We take the opposite view: Present the maintainer with a semantically constrained problem and let him construct the solution which implements the change within these constraints. The semantic context with which we propose to constrain the maintainer is one that will *prohibit* linkages into the portions of the code that the maintainer does not want to change. This approach uncovers potential problems earlier than the aforementioned methods, and, we believe, is worth any inconvenience that may be encountered due to the imposition of the constraints.

Our program slicing-based techniques give an assessment of the impact of proposed modifications, ease the problems associated with revalidation, and reduce the resources required for maintenance activities. They work on unstructured programs, so they are usable on older systems. They may be used for white-box, spare-parts, and backbone maintenance without regard to whether the maintenance is corrective, adaptive, perfective, or preventive.

II. Background

Program slicing, introduced by Weiser [33], [36], is a technique for restricting the behavior of a program to some specified subset of interest. A slice $S(v, n)$ (of program P) on variable v, or set of variables, at statement n yields the portions of the program that contributed to the value of v just before statement n is executed. $S(v, n)$ is called a *slicing criteria.* Slices can be computed automatically on source programs by analyzing data flow and control flow. A program slice has the added advantage of being an executable program. Slicing is done implicitly by programmers while debugging [33], [35]; slices can be combined to isolate sections of code likely to contain program faults and significantly reduce debugging times [23]–[25].

There has been a flurry of recent activities where slicing plays a significant role. Horwitz *et al.* [15], [16], [28] use slices in integrating programs. Their results are built on the seminal

Manuscript received March 17, 1989; revised April 5, 1991. Recommended by R. A. DeMille.

K. B. Gallagher is with the Computer Science Department, Loyola College in Maryland, 4501 N. Charles Street, Baltimore, MD 21210.

J. R. Lyle is with the University of Maryland, Baltimore Campus, 5401 Wilkens Avenue, Baltimore, MD 21228.

IEEE Log Number 9101139.

work of Ottenstein and Ottenstein [7], [27], combining slicing with the robust representation afforded by program dependence graphs. Korel and Laski [20]–[22] use slices combined with execution traces for program debugging and testing. Choi *et al.* [6] use slices and traces in debugging parallel programs. Reps and Wang [29] have investigated termination conditions for program slices. Hausler [13] has developed a denotational approach to program slicing. Gallagher [8] has improved Lyle's [23] algorithm for slicing in the presence of GOTO's and developed techniques for capturing arbitrarily placed output statements. We will not discuss slicing techniques in this paper and instead refer the interested reader to these works.

Since we want to avoid getting bogged down in the details of a particular language, we will identify a program with its flowgraph. Each node in the graph will correspond to a single-source language statement. Henceforth the term statement will mean a node in the flowgraph. Using a common representation scheme makes the presentation clear, although it is clear that any tool based on these techniques will need to account for the nuances of the particular language. In this paper we also ignore problems introduced by having dead code in the source program, and declare that the programs under consideration will not have any dead code. See [8] for slicing-based techniques to eliminate dead code.

Figs. 2–6 illustrate slicing on the program of Fig. 1, a bare bones version of the Unix utility **wc**, word count, taken from [19]. The program counts the number of characters, words, and lines in a text file. It has been slightly modified to illustrate more clearly the slicing principles. The slices of Figs. 2–4 are complete programs which compute a restriction of the specification. The slice on **nw** (Fig. 2) will output the number of words in a file; the slice on **nc** (Fig. 3) will count the number of characters in the input text file; and the slice on **nl** (Fig. 4) will count the number of lines in the file.

```
1   #define YES 1
2   #define NO 0
3   main()
4   {
5       int c, nl, nw, nc, inword ;
6       inword = NO ;
7       nl = 0;
8       nw = 0;
9       nc = 0;
10      c = getchar();
11      while ( c != EOF ) {
12          nc = nc + 1;
13          if ( c == '\n')
14              nl = nl + 1;
15          if ( c == ' ' || c == '\n' || c == '\t')
16              inword = NO;
17          else if (inword == NO) {
18              inword = YES ;
19              nw = nw + 1;
20          }
21          c = getchar();
22      }
23      printf("%d \n",nl);
24      printf("%d \n",nw);
25      printf("%d \n",nc);
26  }
```

Fig. 1. Program to be sliced.

```
1   #define YES 1
2   #define NO 0
3   main()
4   {
5       int c, nw, inword ;
6       inword = NO ;
8       nw = 0;
10      c = getchar();
11      while ( c != EOF ) {
15          if ( c == ' ' || c == '\n' || c == '\t')
16              inword = NO;
17          else if (inword == NO) {
18              inword = YES ;
19              nw = nw + 1;
20          }
21          c = getchar();
22      }
24      printf("%d \n",nw);
26  }
```

Fig. 2. Slice on **nw**: word counter.

```
3   main()
4   {
5       int c, nc ;
9       nc = 0;
10      c = getchar();
11      while ( c != EOF ) {
12          nc = nc + 1;
21          c = getchar();
22      }
25      printf("%d \n",nc);
26  }
```

Fig. 3. Slice on **nc**: character counter.

III. Using Slices for Decomposition

This section presents a method for using slices to obtain a decomposition of the program. Our objective is to use slicing to decompose a program "directly" into two (or more) components. A program slice will be one of the components. The construction is a two step process. The first step is to build, for one variable, a *decomposition slice,* which is the union of certain slices taken at certain line numbers on the given variable. Then the other component of the decomposition, called the *complement,* will also be obtained from the original program. The complement is constructed in such a way that when certain statements of the decomposition slice are removed from the original program, the program that remains is the slice that corresponds to the *complement* (in a sense to be defined) of the given criteria with respect to the variables defined in the program. Thus the complement is also a program slice.

The decomposition slice is used to guide the removal of statements in a systematic fashion to construct the complement. It is insufficient to merely remove the slice statements from the original program. Since we require that a slice be executable, there will be certain crucial statements that are necessary in both the slice and its complement. For example, if we start with the slice of Fig. 2 and remove *all* its statements from the original program, the resulting object will not even compile!

```
3    main()
4    {
5         int c, nl, ;
7         nl = 0;
10        c = getchar();
11        while ( c != EOF ) {
13             if ( c == '\n')
14                  nl = nl + 1;
21             c = getchar();
22        }
23        printf("%d \n",nl);
26   }
```

Fig. 4. Slice on **nl**: line counter.

```
1    #define YES 1
2    #define NO 0
3    main()
4    {
5         int c, inword ;
6         inword = NO ;
10        c = getchar();
11        while ( c != EOF ) {
15             if ( c == ' ' || c == '\n' || c == '\t')
16                  inword = NO;
17             else if (inword == NO) {
18                  inword = YES ;
20             }
21             c = getchar();
22        }
26   }
```

Fig. 5. Slice on **inword.**

```
3    main()
4    {
5         int c ;
10        c = getchar();
11        while ( c != EOF ) {
21             c = getchar();
22        }
26   }
```

Fig. 6. Slice on **c**.

We use this decomposition to break the program into manageable pieces and automatically assist the maintainer in guaranteeing that there are no ripple effects induced by modifications in a component. We use the complement to provide a semantic context for modifications in the decomposition slice; the complement must remain fixed after any change.

The decomposition ideas presented in this section are independent of a particular slicing method. Once a slice is obtained by any slicing algorithm, a program decomposition may be computed. Clearly, the quality of the decomposition will be affected by the quality of the slice, in the sense that more refined slices give a finer granularity and also deliver more semantic information to the maintainer.

A program slice is dependent on a variable and a statement number. A *decomposition slice* does not depend on statement numbers. The motivation for this concept is easily explained using the example of Fig. 7. The slice $S(t,4)$ is statements 1, 2, 3, 4, while the slice $S(t,6)$ is statements 1, 2, 5, 6. Slicing at statement *last* (in this case 6) of a program is insufficient to get all computations involving the slice variable

```
1         input a
2         input b
3         t = a + b
4         print t
5         t = a - b
6         print t
```

Fig. 7. Requires a decomposition slice.

t. A decomposition slice captures all relevant computations involving a given variable.

To construct a decomposition slice, we borrow the concept of *critical instructions* from an algorithm for dead code elimination as presented in Kennedy [18]. A brief reprise follows. The usual method for dead code elimination is to first locate all instructions that are useful in some sense. These are declared to be the critical instructions. Typically, dead code elimination algorithms start by marking output instructions to be critical. Then the *use-definition* [18] chains are traced to mark the instructions which impact the output statements. Any code that is left unmarked is useless to the given computation.

Definition 1: Let $Output(P,v)$ be the set of statements in program P that output variable v, let $last$ be the last statement of P, and let $N = Output(P,v) \cup \{last\}$. The statements in $\bigcup_{n \in N} S(v)$ form the decomposition slice on v, denoted $S(v)$.

The decomposition slice is the union of a collection of slices, which is still a program slice [36]. We include statement $last$ so that a variable which is not output may still be used as a decomposition criteria; this will also capture any defining computation on the decomposition variable after the last statement that displays its value. To successfully take a slice at statement last, we invoke one of the crucial differences between the slicing definitions of Reps [29], with those of Weiser [36], Lyle [23], and this work. A Reps slice must be taken at a point p with respect to a variable which is defined or referenced at p. Weiser's slices can be taken at an arbitrary variable at an arbitrary line number. This difference prohibits Reps's slicing techniques from being applicable in the current context, since we want to slice on every variable in the program at the last statement.

We now begin to examine the relationship between decomposition slices. Once we have this in place, we can use the decomposition slices to perform the actual decompositions. To determine the relationships, we take the decomposition slice for each variable in the program and form a lattice of these decomposition slices, ordered by set inclusion. It is easier to gain a clear understanding of the relationship between decomposition slices if we regard them *without* output statements. This may seem unusual in light of the above definition, since we used output statements in obtaining relevant computations. We view output statements as windows into the current state of computation, which do not contribute to the realization of the state. This coincides with the informal definition of a slice: the statements which yield the portions of the program that contributed to the value of v just before statement n is executed. Assuming that output statements do not contribute to the value of a variable precludes from our discussion output statements (and therefore programs) in which the output values are reused, as is the case with random

access files or output to files which are later reopened for input. Moreover, we are describing a decomposition technique which is not dependent on any particular slicing technique; we have no way of knowing whether or not the slicing technique includes output statements or not. We say a slice is *output-restricted* if all its output statements are removed.

Definition 2: Output restricted decomposition slices $S(v)$ and $S(w)$ are *independent* if $S(v) \cap S(w) = \emptyset$.

It would be a peculiar program that had independent decomposition slices; they would share neither control flow or data flow. In effect, there would be two programs with nonintersecting computations on disjoint domains that were merged together. The lattice would have two components. In Ott's slice metric terminology [26], independence corresponds to low (coincidental or temporal) cohesion.

Output-restricted decomposition slices that are not independent are said to be (*weakly*) *dependent.* Subsequently, when we speak of independence and dependence *of slices,* it will always be in the context of output-restricted decomposition slices.

Definition 3: Let $S(v)$ and $S(w)$ be output-restricted decomposition slices, $w \neq v$, and let $S(v) \subset S(w)$. $S(v)$ is said to be *strongly dependent* on $S(w)$.

Thus output-restricted decomposition slices strongly dependent on independent slices are independent. The definitions of independence and dependence presented herein are themselves dependent on the notion of a slice. The analogous definitions are used by Bergeretti and Carré [3] to *define* slices. In Ott's metric terminology [26], strong dependence corresponds to high (sequential or functional) cohesion.

Strong dependence of decomposition slices is a binary relation; in most cases, however, we will not always need an explicit reference to the containing slice. Henceforth we will write "$S(v)$ is strongly dependent" as a shorthand for "$S(v)$ is strongly dependent on some other slice $S(w)$" when the context permits it.

Definition 4: An output-restricted slice $S(v)$ that is *not* strongly dependent on any other slice is said to be *maximal.*

Maximal decomposition slices are at the "ends" of the lattice. This definition gives the motivation for output restriction; we do not want to be concerned with the possible effects of output statements on the maximality of slices or decomposition slices. This can be observed by considering the decomposition slices on **nw** and **inword**, of Figs. 2 and 5. If we regarded output statements in defining maximal, we could force the slice on **inword** to be maximal by the addition of a *print* statement referencing **inword** along with the others at the end of the program. Such a statement would not be collected into the slice on **nw**. Since this added statement is not in any other slice, the slice on **inword** would be maximal and it should not be.

Fig. 8 gives the lattice we desire. $S(nc), S(nl)$, and $S(nw)$ are the maximal decomposition slices. $S(inword)$ is strongly dependent on $S(nw)$; $S(c)$ is strongly dependent on all the other decomposition slices. The decomposition slices on $S(nw), S(nc)$, and $S(nl)$ (Figs. 2–4) are weakly dependent and maximal when the output statements are removed. There

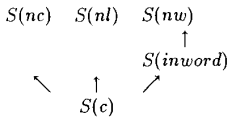


Fig. 8. Lattice of decomposition slices.

are no independent decomposition slices in the example. Recall that independent decomposition slices cannot share any control flow: the surrounding control statements would make them dependent.

We now begin to classify the individual statements in decomposition slices.

Definition 5: Let $S(v)$ and $S(w)$ be output-restricted decomposition slices of program P. Statements in $S(v) \cap S(w)$ are called *slice dependent statements.*

Slice independent statements are statements which are not slice dependent. We will refer to slice dependent statements and slice independent statements as *dependent statements* and *independent statements.* Dependent statements are those contained in decomposition slices which are interior points of the lattice; independent statements are those in a maximal decomposition slice which are not in the union of the decomposition slices which are properly contained in the maximal slice. The terms arise from the fact that two or more slices *depend* on the computation performed by dependent statements. Independent statements do not contribute to the computation of any other slice. When modifying a program, dependent statements cannot be changed or the effect will ripple out of the focus of interest.

For example, statement 12 of the slice on **nc** (Fig. 3) is a slice independent statement with respect to any other decomposition slice. Statements 13 and 14 of the slice on **nl** (Fig. 4) are also slice independent statements with respect to any other decomposition slice. The decomposition slice on **c** (Fig. 6) is strongly dependent on all the other slices; thus all its statements are slice dependent statements with respect to any other decomposition slice. Statements 6 and 15–20 of the slice on **nw** (Fig. 2) are slice independent statements with respect to decomposition slices $S(nc), S(nl)$, and $S(c)$; only statement 19 is slice independent when compared with $S(inword)$. Statements 6, 15–18, and 20 of the decomposition slice on **inword** (Fig. 5) are slice independent statements with respect to decomposition slices $S(nc), S(nl)$, and $S(c)$; no statements are slice independent when compared with $S(nw)$.

We have a relationship between maximal slices and independent statements. This proposition permits us to apply the terms "(slice) independent statement" and "(slice) dependent statement" in a sensible way to a particular statement in a given maximal decomposition slice without reference to the binary relation between decomposition slices which is required in definition 5.

Proposition 1: Let 1) $\mathrm{Varset}(P)$ be the set of variables in program P; 2) $S(v)$ be an output-restricted decomposition slice of P; 3) Let $M = \{m \in \mathrm{Varset}(P) | S(m) \text{ is maximal}\}$; 4) Let $U = M - \{v\}$. The statements in $S(v) - \bigcup_{u \in U} S(u)$ are independent.

Proof: Let $U = \{u_1, \ldots, u_m\}$. $S(v) - \bigcup_{u \in U} S(u) = S(v) - S(u_1) \ldots - S(u_m)$. ◊

There is a relationship between the maximal slices and program. (Recall that dead code has been excluded from our discussions.)

Proposition 2: Let $M = \{m \in \text{Varset}(P) | S(m)$ is maximal. Then $\bigcup_{m \in M} S(m) = P$.

Proof: Since $S(m) \in P$, $\bigcup_{m \in M} S(m) \subset P$. If $P \not\subseteq \bigcup_{m \in M} S(m)$, then the statements in P that are not in $\bigcup_{m \in M} S(m)$ are dead code. ◊

Maximal slices capture the computation performed by the program. Maximal slices and their respective independent statements also are related.

Proposition 3: An output-restricted decomposition slice is maximal if it has at least one independent statement.

Proof: Suppose $S(v)$ is maximal. By definition, $S(v)$ has at least one statement that no other slice has. This statement is an independent statement.

Now suppose that $S(v)$ has an independent statement s. Then s is not in any other slice, and the slice that contains s is maximal. ◊

Conversely, a slice with no independent statements is strongly dependent.

We also have another characterization of strongly dependent slices.

Proposition 4: Let 1) $\text{Varset}(P)$ be the set of variables in program P; 2) $S(v)$ be an output-restricted decomposition slice of P; 3) Let $D = \{w \in \text{Varset}(P) | S(v)$ is strongly dependent on $S(w)\}$; 4) Let $M = \{m \in \text{Varset}(P) | S(m)$ is maximal$\}$; 5) Let $U = M - \{v\}$. An output-restricted decomposition slice $S(v)$ is strongly dependent (on some $S(d)$) if $\bigcup_{u \in U} S(u) = P$.

Proof: Suppose $S(v)$ is strongly dependent. We need to show that D has a maximal slice. Partially order D by set inclusion. Let d be one of the maximal elements of D. The element d is maximal; if it is not, then it is properly contained in another slice d_1, which is in D and contains $S(v)$. Then $d \in M, d \neq v$, and $S(v)$ makes no contribution to the union.

Suppose $\bigcup_{u \in U} S(u) = P$. Since $U \subseteq M, S(v)$ makes no contribution to the union. By proposition 3, $S(v)$ is strongly dependent. ◊

We are now in a position to state the decomposition principles. Given a maximal output-restricted decomposition slice $S(v)$ of program P, delete the independent and output statements of S and P. We will denote this program $\sum(v)$ and call it the *complement of decomposition slice* $S(v)$ (with respect to P). Henceforth, when we speak of complements, it will always be in the context of decomposition slices. The decomposition slice is the subset of the program that computes a subset of the specification; the complement computes the rest of the specification.

Figs. 9–11 give the complements of the slices on **nw**, **nc**, and **nl** of Figs. 2–4. Using proposition 4, we obtain that the complement of both the slice on **inword** and the slice on **c** is the entire program.

This yields the approximation of a direct sum decomposition of a program which preserves the computational integrity

```
3    main()
4    {
5         int c, nl, nw, nc, inword ;
7         nl = 0;
9         nc = 0;
10        c = getchar();
11        while ( c != EOF ) {
12             nc = nc + 1;
13             if ( c == '\n')
14                  nl = nl + 1;
21             c = getchar();
22        }
23        printf("%d \n",nl);
25        printf("%d \n",nc);
26   }
```

Fig. 9. $\sum(nw)$. Complement of slice on **nw**: computes line count and character count.

```
1    #define YES 1
2    #define NO 0
3    main()
4    {
5         int c, nl, nw, nc, inword ;
6         inword = NO ;
7         nl = 0;
8         nw = 0;
9         nc = 0;
10        c = getchar();
11        while ( c != EOF ) {
12             nc = nc + 1;
13             if ( c == '\n')
14                  nl = nl + 1;
15             if ( c == ' ' || c == '\n' || c == '\t')
16                  inword = NO;
17             else if (inword == NO) {
18                  inword = YES ;
19                  nw = nw + 1;
20             }
21             c = getchar();
22        }
23        printf("%d \n",nl);
24        printf("%d \n",nw);
26   }
```

Fig. 10. $\sum(nc)$. Complement of slice on **nc**: computes word count and line count.

of the constituent parts. This also indicates that the only useful decompositions are done with maximal decomposition slices. A complement $\sum$ of a maximal slice can be further decomposed, so the decomposition may be continued until all slices with independent statements (i.e., the maximal ones) are obtained.

In practice, a maintainer may find a strongly dependent slice as a starting point for a proposed change. Our method will permit such changes. Such a change may be viewed as properly *extending* the domain of the partial function that the program computes, while preserving the partial function on its original domain.

IV. Application to Modification and Testing

Statement independence can be used to build a set of guidelines for software modification. To do this, we need to make one more set of definitions regarding variables which appear in independent and dependent statements. With these

```
1   #define YES 1
2   #define NO 0
3   main()
4   {
5       int c, nl, nw, nc, inword ;
6       inword = NO ;
8       nw = 0;
9       nc = 0;
10      c = getchar();
11      while ( c != EOF ) {
12          nc = nc + 1;
15          if ( c == ' ' || c == '\n' || c == '\t')
16              inword = NO;
17          else if (inword == NO) {
18              inword = YES ;
19              nw = nw + 1;
20          }
21          c = getchar();
22      }
24      printf("%d \n",nw);
25      printf("%d \n",nc);
26  }
```

Fig. 11. $\sum(nl)$. Complement of slice on **nl**: computes character count and word count.

definitions we give a set of rules which maintainers must obey in order to make modifications without ripple effects and unexpected linkages. When these rules are obeyed, we have an algorithm to merge the modified slice back into the complement and effect a change. The driving motivation for the following development is: "What restrictions must be placed on modifications in a decomposition slice so that the *complement* remains intact?"

Definition 6: A *variable* that is the target of a dependent assignment statement is called a *dependent variable.* Alternatively and equivalently, if *all* assignments to a variable are in independent statements, then the variable is called an *independent variable.*

An assignment statement can be an independent statement while its target is not an independent variable. In the program of Fig. 12, the two maximal decomposition slices are $S(a)$ and $S(e)$ (Figs. 13 and 14). Slice $S(b)$ (Fig. 15) is strongly dependent on $S(a)$, and $S(f)$ (Fig. 16) is strongly dependent on $S(b)$ and $S(a)$. $S(d)$ and $S(c)$ (not shown) are strongly dependent on both maximal slices. In $S(a)$, statements 8, 10, and 11 are independent, by the proposition. But *variables* **a** and **b** are targets of assignment statements 6 and 5, respectively. So, in the decomposition slice $S(a)$, only variable **f** is an independent variable.

A similar argument applies for independent control flow statements which reference dependent variables. A dependent variable in an independent statement corresponds to the situation where the *variable* in question is required for the compilation of the complement, but the *statement* in question does not contribute to complement. If a variable is referenced in a dependent statement, it is necessary to the complement and cannot be independent.

If a decomposing on a single variable yields a strongly dependent slice, we are able to construct a slice where the original slice *variable* is an independent variable.

Proposition 5: Let 1) Varset(P) be the set of variables in program P; 2) $S(v)$ be a strongly dependent output restricted decomposition slice of P; 3) Let $D = \{w \in \text{Varset}(P) | S(v)$ is strongly dependent on $S(w)\}$; 4) Let $M = \{m \in \text{Varset}(P) | S(m)$ is maximal$\}$; 5) Let $U = D \cap M$; 6) Let $T = \bigcup_{u \in U} S(u)$. The variable v is an independent variable in T.

In other words, when $S(v)$ is a strongly dependent slice and T is the union of all the maximal slices upon which $S(v)$ is strongly dependent, then v is an independent variable in T.

Proof: We show that the complement of T, $P-T$ has no references to v: if variable v is in the complement of T, then there is a maximal slice in the complement upon which

```
1  main()
2  {
3     int a, b, c, d, e, f;
4     c = 4;
5     b = c;
6     a = b + c;
7     d = a + c;
8     f = d + b;
9     e = d + 8;
10    b = 30 + f;
11    a = b + c;
12 }
```

Fig. 12. Dependent variable sample program.

```
1  main()
2  {
3     int a, b, c, d, e, f;
4     c = 4;
5     b = c;
6     a = b + c;
7     d = a + c;
8     f = d + b;
10    b = 30 + f;
11    a = b + c;
12 }
```

Fig. 13. Slice on **a**.

```
1  main()
2  {
3     int a, b, c, d, e, f;
4     c = 4;
5     b = c;
6     a = b + c;
7     d = a + c;
9     e = d + 8;
12 }
```

Fig. 14. Slice on **e**.

```
1  main()
2  {
3     int a, b, c, d, e, f;
4     c = 4;
5     b = c;
6     a = b + c;
7     d = a + c;
8     f = d + b;
10    b = 30 + f;
12 }
```

Fig. 15. Slice on **b**.

```
1  main()
2  {
3     int a, b, c, d, e, f;
4     c = 4;
5     b = c;
6     a = b + c;
7     d = a + c;
8     f = d + b;
12 }
```

Fig. 16. Slice on **f**.

$S(v)$ is strongly dependent. This contradicts the hypotheses, so the complement if T has no references to v and the variable v is independent in T. ◊

This can be interpreted as the variable version of Proposition 1, which refers to statements.

This has not addressed the problem that is presented when the decomposition slice on variable is maximal, but the variable itself remains dependent. This is the situation that occurred in the example at the beginning of the chapter; the slice on variable *a* (Fig. 13) is maximal, but the variable is dependent. The solution is straightforward: we construct the slice that is the union of all slices in which the variable is dependent.

Proposition 6: Let 1) $\text{Varset}(P)$ be the set of variables in program P; 2) $S(v)$ be an output-restricted decomposition slice of P; 3) Let $E = \{w \in \text{Varset}(P) | v$ is a dependent variable in $S(w)\}$; 4) Let $T = \bigcup_{e \in E} S(e)$.

We have two cases: 1) $E = \varnothing$ (and thus T is empty also), in which case v is an independent variable; 2) $E \neq \varnothing$, so T is not empty and the variable v is an independent variable in T.

Proof:

Case 1: $E = \varnothing$

$S(v)$ contains all references to v. In particular, $S(v)$ contains all assignments to v. So v is an independent variable in $S(v)$. *End Case 1*

Case 2: $E \neq \varnothing$

T contains references to v. In particular, T contains all assignments to v. So v is an independent variable in T. *End Case 2* ◊

This proposition is about *variables.*

A. Modifying Decomposition Slices

We are now in a position to answer the question posed at the beginning of this section. We present the restrictions as a collection of rules with justifications.

Modifications take three forms: additions, deletions, and changes. A change may be viewed as a deletion followed by an addition. We will use this second approach and determine only those statements in a decomposition slice that can be deleted, and the forms of statements that can be added. Again, we must rely on the fact that the union of decomposition slices is a slice, since the complementary criteria will usually involve more than one maximal variable. We also assume that the maintainer has kept the modified program compilable and has obtained the decomposition slice of the portion of the software that needs to be changed. (Locating the code may be a highly nontrivial activity; for the sake of the current discussion, we assume its completion.)

Since independent statements do not affect data flow or control flow in the complement, we have:

Rule 1: Independent statements may be deleted from a decomposition slice.

Reason: Independent statements do not affect the computations of the complement. Deleting an independent statement from a slice will have no impact on the complement. ◊

This result applies to control flow statements and assignment statements. The statement may be deleted, even if it is an assignment statement which targets a dependent variable or a control statement which references a dependent variable. The point to keep in mind is that if the statement is independent, it does *not* affect the complement. If an independent statement is deleted, there will certainly be an effect in the slice. But the purpose of this methodology is to keep the complement intact.

There are a number of situations to consider when statements are to be added. We progress from simple to complex. Also note that for additions, new variables may be introduced *as long as the variable name does not clash with any name in the complement.* In this instance the new variable is independent in the decomposition slice. In the following, *independent variable* means an independent variable or a new variable.

Rule 2: Assignment statements that target independent variables may be added anywhere in a decomposition slice.

Reason: Independent variables are unknown to the complement. Thus changes to them cannot affect the computations of the complement. ◊

This type of change is permissible even if the changed value flows into a dependent variable. In Fig. 13, changes are permitted to the assignment statement at line 8, which targets **f**. A change here would propagate into the values of dependent variables **a** and **b** at lines 10 and 11. The maintainer would then be responsible for the changes which would occur to these variables. If lines 10 and 11 were dependent (i.e., contained in another decomposition slice), line 8 would also be contained in this slice and variable **f** would be dependent.

Adding control flow statements requires a little more care. This is required because control statements have two parts: the logical expression, which determines the flow of control, and the *actions* taken for each value of the expression. (We assume no side effects in the evaluation of logical expressions.) We discuss only the addition of **if-then-else** and **while** statements, since all other language constructs can be realized by them [5].

Rule 3: Logical expressions (and output statements) may be added anywhere in a decomposition slice.

Reason: We can inspect the state of the computation anywhere. Evaluation of logical expressions (or the inclusion of an output statement) will not even affect the computation of the slice. Thus the complement remains intact. ◊

We must guarantee that the statements which are controlled by newly added control flow do not interfere with the complement.

Rule 4: New control statements that surround (i.e., control) any dependent statement will cause the complement to change.

Reason: Suppose that newly added code controls a dependent statement.

Let C be the criteria which yield the complement. When using this criteria on the modified program, the newly added control code will be included in this complementary slice. This is due to the fact that the dependent statements are in both the slice and the complement. Thus any control statements which control dependent statements will also be in the slice and the complement. ◊

By making such a change we have violated the principle that the complement remain fixed. Thus new control statements may not surround any dependent statement.

This short list is necessary and sufficient to keep the slice complement intact. This also has an impact on testing the change that will be discussed later.

Changes may be required to computations involving a dependent variable *v* in the extracted slice. the maintainer can choose one of the following two approaches:

1) Use the techniques of the previous section to extend the slice so that *v* is independent in the slice.
2) Add a new local variable (to the slice), copy the value to the new variable, and manipulate the new name only. Of course, the new name must not clash with any name in the complement. This technique may also be used if the slice has no independent statements; i.e., it is strongly dependent.

B. Merging the Modifications into the Complement

Merging the modified slice back into the complement is straightforward. A key to understanding the merge operation comes from the observation that through the technique, the maintainer is editing the *entire program.* The method gives a view of the program with the unneeded statements deleted and with the dependent statements restricted from modification. The slice gives a smaller piece of code for the maintainer to focus on, while the rules of the previous subsection provide the means by which the deleted and restricted parts cannot be changed accidentally.

We now present the merge algorithm.

1) Order the statements in the original program. (In the following examples we have one statement per line, so that the ordering is merely the line numbering.) A program slice and its complement can now be identified with the subsequence of statement numbers from the original program. We call the sequence numbering from the slice the *slice sequence,* and the numbering of the complement, the *complement sequence.* We now view the editing process as the addition and deletion of the associated sequence numbers.
2) For deleted statements, delete the sequence number from the slice sequence. Observe that since only independent statements are deleted, this number is not in the complement sequence.
3) For statements inserted into the slice, a new sequence number needs to be generated. Let P be the sequence number of the statement preceding the statement to be inserted. Let M be the least value in the slice sequence greater than P. Let $F = \min(\text{int}(P+1), M)$. Insert the new statement at sequence number $(F+P)/2$. (Although this works in principle, in practice, more care needs to be taken in the generation of the insertion sequence numbers to avoid floating point errors after 10 inserts.)
4) The merged program is obtained by merging the modified slice sequence values (i.e., statements) into the complement sequence.

Thus the unchanged dependent statements are used to guide the reconstruction of the modified program. The placement of the changed statements within a given control flow is arbitrary. Again, this becomes clearer when the editing process is viewed as a modification to the entire program. The following example will help clarify this.

C. Testing the Change

Since the maintainer must restrict all changes to independent or newly created variables, testing is reduced to testing the modified slice. Thus the need for regression testing in the complement is eliminated. There are two alternative approaches to verifying that only the change needs testing. The first is to slice on the original criteria, plus any new variables, minus any eliminated variables, and compare its complement with the complement of the original: they should match exactly. The second approach is to preserve the criteria which produced the original complement. Slicing out on this must produce the modified slice exactly.

An axiomatic consideration illumines this idea. The slice and its complement perform a subset of the computation; where the computations meet are the dependencies. Modifying the code in the independent part of the slice leaves the independent part of the complement as an invariant of the slice (and vice versa).

If the required change is "merely" a module replacement, the preceding techniques are still applicable. The slice will provide a harness for the replaced module. A complete independent program supporting the module is obtained. One of the principal benefits of slicing is highlighted in this context: any side effects of the module to be replaced will also be in the slice. Thus the full impact of change is brought to the attention of the modifier.

As an example, we make some changes to $S(nw)$, the slice on **nw**, the word counter of Fig. 2. The changed slice is shown in Fig. 17. The original program determined a word to be any string of "nonwhite" symbols terminated by a "white" symbol (space, tab, or newline). The modification changes this to the requirement to be alphabetical characters terminated by white space. (The example is illustrating a change, not advocating it.) Note the changes. We have deleted the independent "variables" YES and NO; added a new, totally independent variable **ch**; and revamped the independent statements. The addition of the C macros **isspace** and **isalpha** is safe, since the results are only referenced. We test this program independently of the complement. Fig. 18 shows the reconstructed, modified program. Taking the decomposition slice on **nw** generates the program of Fig. 17. Its complement is already given in Fig. 9.

```
3    main()
4    {
*         int ch;
5         int c, nw ;
*         ch = 0;
8         nw = 0;
10        c = getchar();
11        while ( c != EOF ) {
*              if (isspace(c) && isalpha(ch))
*                   nw = nw + 1;
*              ch = c ;
21             c = getchar();
22        }
24        printf("%d \n",nw);
26   }
```

Fig. 17. Modified slice on **nw**, the word counter.

```
3    main()
4    {
*         int ch;
5         int c, nl, nw, nc ;
*         ch = 0;
7         nl = 0;
8         nw = 0;
9         nc = 0;
10        c = getchar();
11        while ( c != EOF ) {
*              if (isspace(c) && isalpha(ch))
*                   nw = nw + 1;
*              ch = c ;
12             nc = nc + 1;
13             if ( c == '\n')
14                  nl = nl + 1;
21             c = getchar();
22        }
23        printf("%d \n",nl);
24        printf("%d \n",nw);
25        printf("%d \n",nc);
26   }
```

Fig. 18. Modified program.

the starred (*) statements indicate where the new statements would be placed using the line-number-generation technique above.

V. A New Software Maintenance Process Model

The usual Software Maintenance Process Model is depicted in Fig. 19. A request for change arrives. It may be adaptive, perfective, corrective, or preventive. In making the change, we wish to minimize defects, effort, and cost, while maximizing customer satisfaction [12]. The software is changed, subject to pending priorities. The change is composed of two parts: Understanding the code, which may require documentation, code reading, and execution. Then the program is modified. The maintainer must first design the change (which may be subject to peer review), then alter the code itself, while trying to minimize side effects. The change is then validated. The altered code itself is verified to assure conformance with the specification. Then the new code is integrated with the existing system to insure conformance with the system specifications. This task involves regression testing.

The new model is depicted in Fig. 20. The software is changed, subject to pending priorities. The change is composed of two parts: Understanding the code will now require documentation, code reading, execution, and the use of decomposition slices. The decomposition slices may be read and executed (a decided advantage of having executable program slices). The code is then modified, subject to the strictures outlined. Using those guidelines, no side effects or unintended linkages can be induced in the code, even by accident. This lifts a substantial burden from the maintainer.

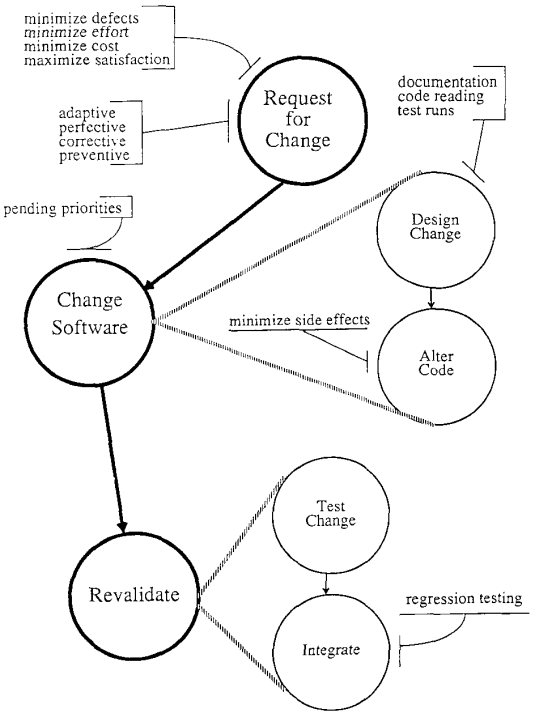


Fig. 19. A software maintenance process model.

The change is tested in the decomposition slice. Since the change cannot ripple out into other modules, regression testing is unnecessary. The maintainer need only verify that the change is correct. After applying the merge algorithm, the change (of the code) is complete.

VI. Future Directions

The underlying method and the tool based on it [9] need to be empirically evaluated. This is underway using the Goal-Question-Metric paradigm of Basili *et al.* [2]. Naturally, we are also addressing questions of scale, to determine if existing software systems decompose sufficiently via these techniques, in order to effect a technology transfer. We are also evaluating decomposition slices as candidates for components in a reuse library.

Although they seem to do well in practice, the slicing algorithms have relatively bad worst-case running times of $O(n\ e\ \log(e))$, where n is the number of variables and e is the number of edges in the flowgraph. To obtain all the slices, this running time becomes $O(n^2 e\ \log(e))$. These worst-case times would seem to make an interactive slicer

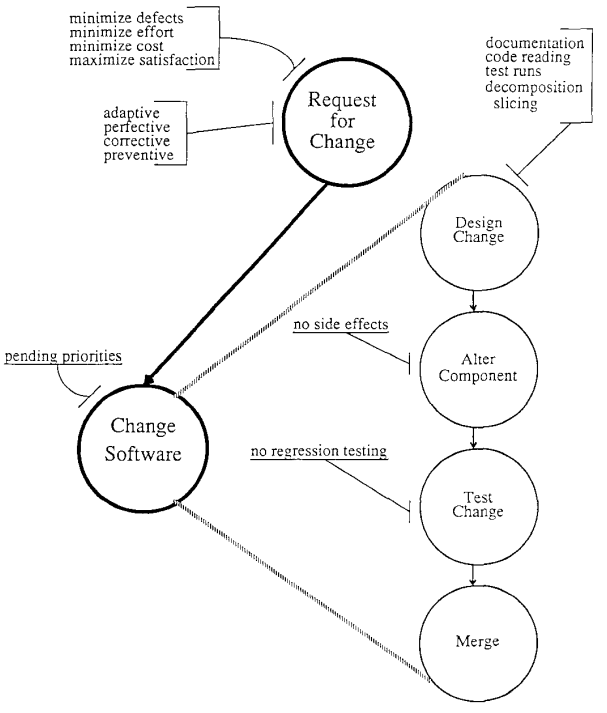


Fig. 20. A new software maintenance process model.

for large (i.e., real) programs impractical. This difficulty can be assuaged by making the data-flow analysis one component of the deliverable products which are handed off from the development team to the maintenance team. An interactive tool could then be built using these products. Then as changes are made by the maintainers, the data flow data can be updated, using the incremental techniques of Keables [17].

Interprocedural slices can be attacked using the techniques in Weiser [36] and Barth [1]. The interprocedural slicing algorithms of Horwitz *et al.* [16] cannot be used, since they require that the slice be taken at a point where the slice variable id **def**ed or **ref**ed; we require that all slices be taken at the last statement of the program. For separate compilation, a worst-case assumption must be made about the external variables if the source is not available. If the source is available, one proceeds as with procedures.

Berzins [4] has attacked the problem of software merges for *extensions* of programs. To quote him:

> "An extension extends the domain of the partial function without altering any of the initially defined values, while a modification redefines values that were defined initially."

We have addressed the modification problem by first *restricting* the domain of the partial function to the slice complement, modifying the function on the values defined by the independent variables in the slice, then merging these two disjoint domains.

Horwitz *et al.* [15] have addressed the modification problem. They start with a *base* program and two modifications it, A and B:

> "Whenever the changes made to *base* to create A and B do not 'interfere' (in a sense defined in the paper), the algorithm produces a program M that integrates A and B. The algorithm is predicated on the assumption that differences in the *behavior* of the variant programs from that of *base,* rather than the differences in *text,* are significant and must be preserved in M."

Horwitz *et al.* do not restrict the changes that can be made to *base;* thus their algorithm produces an approximation to the undecidable problem of determining whether or not the behaviors interfere. We have side-stepped this unsolvable problem by constraining the modifications that are made. Our technique is more akin to the limits placed on software maintainers. Changes must be done in a context: independence and dependence provides the context. It is interesting to note, however, that their work uses program slicing to determine potential interferences in the merge.

They do note that program *variants,* as they name them, are easily embedded in the change control system, such as RCS [31]. Moreover, the direct sum nature of the components can be exploited to build related families of software. That is, components can be "summed" as long as their dependent code sections match exactly, and there is no intersection of the independent domains. We also follow this approach for component construction.

Weiser [34] discusses some slice-based metrics. *Overlap* is a measure of how many statements in a slice are found only in that slice, measured as a mean ratio of nonunique-to-unique statements in each slice. *Parallelism* is the number of slices that has few statements in common, computed as the number of slices that have pairwise overlap below a certain threshold. *Tightness* is the number of statements in every slice, expressed as a ratio over program length. Programs with high overlap and parallelism but with low tightness would decompose nicely: the lattice would not get too deep or too tangled.

We have shown how a data flow technique, program slicing, can be used to form a decomposition for software systems. The decomposition yields a method for maintainers to use. The maintainer is able to modify existing code cleanly, in the sense that the changes can be assured to be completely contained in the modules under consideration and that no unseen linkages with the modified code is infecting other modules.

REFERENCES

[1] J. M. Barth, "A practical interprocedural dataflow analysis algorithm," *Comm. Assoc. Computing Machinery,* vol. 21, no. 9, pp. 724–726, Sept. 1978.

[2] V. Basili, R. Selby, and D. Hutchens, "Experimentation in software engineering," *IEEE Trans. Software Eng.,* vol. 12, pp. 352–357, July 1984.

[3] J.-F. Bergeretti and B. Carré, "Information-flow and data-flow analysis of **while**-programs," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 1, pp. 37–61, Jan. 1985.

[4] V. Berzins, "On merging software extensions," *Acta Informatica,* vol. 23, pp. 607–619, 1985.

[5] C. Bohm and G. Jacopini, "Flow diagrams and languages with only two formation rules," *CACM,* vol. 9, no. 5, pp. 366–371, May 1966.

[6] J.-D. Choi, B. Miller, and P. Netzer, "Techniques for debugging parallel programs with flowback analysis," Univ. Wisconsin–Madison, Tech. Rep. 786, Aug. 1988.

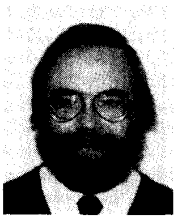
[7] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Languages*

and Systems, vol. 9, no. 3, pp. 319–349, July 1987.
[8] K. B. Gallagher, "Using program slicing in software maintenance," Ph.D. thesis, Univ. Maryland, Baltimore, Dec. 1989.
[9] K. B. Gallagher, "Surgeon's assistant limits side effects," *IEEE Software,* vol. 7, p. 64, May 1990.
[10] K. B. Gallagher and J. R. Lyle, "Using program decomposition to guide modifications," in *Proc. Conf. Software Maintenance—1988,* Oct. 1988, pp. 265–268.
[11] K. B. Gallagher and J. R. Lyle, "A program decomposition scheme with applications to software modification and testing," in *Proc. 22nd Int. Conf. System Sciences* (Hawaii), Jan. 1989, vol. II, pp. 479–485.
[12] R. Grady, "Measuring and managing software maintenance," *IEEE Software,* vol. 4, Sept. 1987.
[13] P. Hausler, "Denotational program slicing," in *Proc. 22nd Hawaii Int. Conf. System Sciences,* Jan. 1989, vol. II (Software Track), pp. 486–494.
[14] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," in *Proc. SIGPLAN'88 Symp. Principles of Programming Languages,* Jan. 1988.
[15] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 3, pp. 345–387, July 1989.
[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 1, pp. 35–46, Jan. 1990.
[17] J. Keables, K. Robertson, and A. von Mayrhauser, "Data flow analysis and its application to software maintenance," in *Proc. Conf. Software Maintenance—1988,* Oct. 1988, pp. 335–347.
[18] K. Kennedy, "A survey of data flow analysis techniques," in *Program Flow Analysis: Theory and Applications,* S. S. Muchnick and N. D. Jones, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[19] B. Kernighan and D. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.
[20] B. Korel and J. Laski, "Dynamic program slicing," *Inform. Process. Lett.,* vol. 29, no. 3, pp. 155–163, Oct. 1988.
[21] B. Korel and J. Laski, "STAD—a system for testing and debugging: User perspective," in *Proc. 2nd Workshop on Software Testing, Verification and Analysis* (Banff, Alberta, Can.), July 1988, pp. 13–20.
[22] J. Laski, "Data flow testing in STAD" *Systems and Software,* to be published.
[23] J. R. Lyle, "Evaluating variations of program slicing for debugging," Ph.D. thesis, Univ. of Maryland, College Park, Dec. 1984.
[24] J. R. Lyle and M. D. Weiser, "Experiments in slicing-based debugging aids," in *Empirical Studies of Programmers,* E. Soloway and S. Iyengar, Eds. Norwood, NJ: Ablex, 1986.
[25] J. R. Lyle and M. D. Weiser, "Automatic program bug location by program slicing," in *Proc. 2nd Int. Conf. Computers and Applications* (Peking, China), June 1987, pp. 877–882.
[26] L. Ott and J. Thuss, "The relationship between slices and module cohesion," in *Proc. 11th Int. Conf. Software Eng.,* May 1989, pp. 198–204.
[27] K. Ottenstein and L. Ottenstein, "The program dependence graph in software development environments," *ACM SIGPLAN Notices,* vol. 19, no. 5, pp. 177–184, May 1984; see also, *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp.* Practical Software Development Environments.
[28] T. Reps and S. Horwitz, "Semantics-based program integration," in *Proc. 2nd European Symp. Programming (ESOP '88)* (Nancy, France), Mar. 1988, pp. 133–145.
[29] T. Reps and W. Yang, "The semantics of program slicing," Univ. Wisconsin–Madison, Tech. Rep. 777, June 1988.
[30] N. Schneidewind, "The state of software maintenance," *IEEE Trans. Software Eng.,* vol. SE-13, pp. 303–310, Mar. 1987.
[31] W. Tichy, "RCS: A system for version control," *Software—Practice and Experience,* vol. 15, no. 7, pp. 637–654, July 1985.
[32] G. Weinberg, "Kill that code!," *Infosystems,* pp. 48–49, Aug. 1983.
[33] M. Weiser, "Program slicing: Formal, psychological and practical investigations of an automatic program abstraction method," Ph.D. thesis, Univ. Michigan, Ann Arbor, 1979.
[34] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Software Eng.,* May 1981, pp. 439–449.
[35] M. Weiser, "Programmers use slicing when debugging," *CACM,* vol. 25, no. 7, pp. 446–452, July 1982.
[36] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.,* vol. SE-10, pp. 352–357, July 1984.

Keith Brian Gallagher received the B.A. degree in mathematics from Bucknell University, the M.S. degrees in both mathematics and computer and communication sciences from the University of Michigan, and the Ph.D. degree from the University of Maryland Graduate School in Baltimore.

He is an Assistant Professor of Computer Science at Loyola College in Baltimore, MD, where his research interests are in the software maintenance process and practice and computing for the disabled.

Dr. Gallagher is a member of the IEEE Computer Society and the Association for Computing Machinery. He has received a Research Initiation Award from the National Science Foundation to continue exploring issues in software maintenance.



James R. Lyle (S'80–M'84) received the B.S. and M.S. degrees in mathematics from East Tennessee State University, and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park.

He is an Assistant Professor in the Department of Computer Science at the University of Maryland, Baltimore, and a Faculty Researcher at the National Institute of Standards and Technology Computer Systems Laboratory in Gaithersburg, MD. His research area is in software tools for testing and debugging supported by graphics workstations.

Dr. Lyle is a member of the IEEE Computer Society and the Association for Computing Machinery.