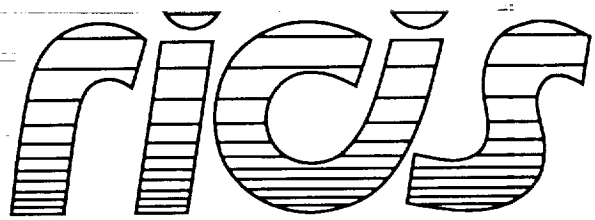
A Survey of Program Slicing for Software Engineering

Jon Beck
West Virginia University Research Corporation

April 5, 1993

Cooperative Agreement NCC 9-16
Research Activity No. RB.10:
RBSE: Component Classification Support

NASA Technology Utilization Program
NASA Headquarters



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

WHITE PAPER

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Jon Beck of West Virginia University. Dr. E. T. Dickerson served as RICIS research coordinator.

Funding was provided by the NASA Technology Utilization Program, NASA Headquarters, Code C, through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Ernest M. Fridge III, Deputy Chief of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

A Survey of Program Slicing for Software Engineering

Jon Beck
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
beck@cs.wvu.edu

5 April, 1993

1 Introduction

1.1 Current Demands in Software Engineering

According to Yourdon [56], software development organizations worldwide are grappling with "staggering" problems of quality and productivity. Demand for quality software is increasing because software of poor quality is very expensive and difficult to maintain; the maintenance portion of the software development process normally consumes from 50% to 80% of total costs [9,42,56]. Improved productivity can reduce risks to schedules and budgets, and reduce the level of resources needed to build new systems. This is significant because the demand for new software systems is increasing faster than the supply of software developers [56].

In both the public and private sectors, there is a recognition that the productivity and quality of the software development process must be increased to meet the ever-increasing need for robust and reliable systems. There is a growing consensus that the incorporation of reuse will be essential for helping to address these needs. For example, the U. S. Department of Defense (DoD) has published a Software Reuse Vision and Strategy document [13] which articulates the DoD vision to drive the software community from its current "re-invent the software" cycle to a process-driven, domain-specific, architecture-centric library-assisted way of constructing software systems. The Software Engineering Institute has developed and applied a domain analysis process (Feature-Oriented Domain Analysis, or FODA) and currently is developing a reuse-based software development methodology that is based on DoD-STD-2167A and focuses on identifying and applying reusable resources [26]. The Fujitsu corporation recognized that in order to remain competitive in today's market, it needed to greatly upgrade the quality and reduce the costs of the software which it produced. Fujitsu developed a domain-specific reuse program for its product line and thereby realized a dramatic increase in productivity and quality. By reusing tested and proven components, the number of switching systems delivered on schedule increased from 20% to 70% [44].

Addressing the productivity and quality issues requires a two-pronged approach, involving improvements in both methodology and tools [48,56]. The methodology component consists of efforts to move the software development process from its historical roots in programming art towards the goal of software engineering. According to Hall [19], current software development practices are not engineering. They lack the discipline, standards, and mathematical principles which characterize true engineering activities. For example, object-oriented development methods, while perhaps good, currently are no more than rules of thumb used by craftsmen. They do not have the predictive power or verifiability of a mathematical specification. More importantly, there is no way of telling whether the object structure is in any sense correct or not. Hall claims that to make software development into an engineering discipline, computer scientists must make sure that the methods used are scientific and produce predictable and verifiable results.[1]

---

1 It is interesting to note that this conflict is not new with Computer Science. In the late 1940s, a leading physicist argued passionately for the need to create a new discipline, Nuclear Engineering, because physicists simply were not equipped by their training for the development of basic scientific

(continued...)

D'Ippolito, et al., state that the primary goal of software developers should be to cast all new work as slight variations of prior work, the qualities of which are predictable by formal analysis before they are implemented. This approach emphasizes "problem setting" over problem solving. In their words:

> The problem setting activity is the dogged insistence of seeing the new in terms of fragments of old paradigms, and appears at first to the non-engineer to be a self-imposed restraint on artistic creativity. There is creativity in engineering, but it is not the sort that allows experimental structures to be tested in public buildings and machines. The creativity is in seeing new uses for old things and in synthesizing new arrangements of them to satisfy human needs. These constraints are what ensure routine designs that are less complex, more maintainable, more predictable in both cost and behavior, and more able to ensure the public safety [14].

The second component in addressing the issues of productivity and quality in software development is the creation and use of specific tools to aid humans by automating various portions of the software development process as prescribed by software engineering methodologies. From the beginnings of the computer age, computer scientists have used tools to help automate the steps in software development. As soon as the first software-programmable machines were available, assemblers, editors, and debuggers were designed and built to help with the process of software creation. As the size and complexity of software systems has increased, the need for advancements in tool technology has increased. Examples of tools which have been developed to meet this need include CASE (Computer Aided Software Engineering) tools, which aid in the activities of software development, from requirements analysis through maintenance. Two other examples are program slicing, used mainly to aid in maintenance activities, and interface slicing [7], developed to facilitate reuse-based software engineering.

1.2 Program Slicing

Program slicing can be viewed as a tool which extracts knowledge from existing systems to aid in their understanding. From the beginning of the computer age, there has been a gulf between the languages that humans and machines understand. The very first computer program existed in the form of hard-wired electrical connections and was not human readable. Even the early written programs were cryptic and undecipherable to all but a few. Modern software systems, given their size and complexity, are equally obscure, their designs and algorithms unknown to any but those who actually had a hand in their creation. It would require a great deal of time and effort for any other computer scientist, even an expert, to understand the design of one of these systems by examining its code. To undertake its maintenance would require a major endeavor.

1(...continued)
phenomena into commercial and industrial applications. Noting that Chemical Engineering had developed from Chemistry a generation earlier, Beck maintained that this was the natural progression from the ad hoc experimentation of an emerging research area to the routine engineering of applications [5,6].

Understanding the design and undertaking the maintenance of a software system, however, is exactly what computer scientists are called to do every day. Because of the complexity of the systems, they use tools to aid in these maintenance efforts. Program slicing is such a tool, developed to decrease the level of effort required to understand complex software systems. From its inception as an aid in debugging, the concept of program slicing has been generalized into a number of different tools, both potential and realized, in the software engineer's kit. Program slicing has been extended to include program comprehension, module cohesion estimation, requirements verification, dead code elimination, and maintenance of various types including reverse engineering, parallelization, portability, and reuse component generation.

2 Definitions and Representation Issues

This section contains a discussion of the graph theoretic and program representation concepts necessary for the material in the following sections. After background syntax and terminology on graph theory, the main ideas presented are:

- Programs have been intuitively represented in various ways using mathematical graphs as the representation vehicle.
- The intuitive program semantics ascribed to these representations have a mathematical underpinning which gives a precise mathematical semantics to the representation.

It is the semantics of the program representation graphs which allows reasoning about slicing which is based on the graph forms.

The following terminology and definitions apply throughout this paper. We present them here to form a common basis for discussion; the literature varies considerably in details of syntax and terminology.

2.1 Terminology and Definitions

In this paper, we will employ the following conventions unless we explicitly note otherwise. A *subprogram* is a unit of code; the term is intended to denote regular procedures and functions, including "main" programs, and where appropriate, more exotic code units such as Ada tasks. If the language under discussion permits, a subprogram may be specification, body, or both. If subprogram A contains subprogram B nested within it, a reference to A will in general *not* include any reference to B unless B is explicitly referenced. A *package* is a collection of subprograms and implies at least the possibility of separate compilation, with allowances made for language systems which do not have the capacity of separate compilation. Package includes the Ada notion of package but is not limited to Ada, as it also may be used to mean a set of units in a standard object library. *Module* is used as a general term to include both subprogram and package as described above, when specifying either would be too restrictive. *Component* specifically refers to a module potentially residing in a reuse repository. A component is thus a code asset of a repository, either before or after it has been reused by incorporation

into a software system. We use the term *element* to mean a named programmatic entity. Types, structures, variables, subprograms, tasks, and exceptions are all included in this term, but statements, even labeled statements, are specifically excluded.

We describe certain characteristics of a program element by using the terms visible, hidden, unprotected, and protected. A program element is *visible* if it is visible and available at least for examination by non-privileged portions of the software system. We use the term *hidden* to refer to a program element which is not visible outside the scope of its module. An element is *unprotected* if there is no language mechanism applied to it which prevents non-privileged access to its internal structure, while an element is *protected* if some form of language-based mechanism, not including simple scope, is used to limit access to its internal structure by non-privileged portions of the system. Thus, visible and hidden refer to access to the element's name, while unprotected and protected refer to an element's internal structure. For example, from the standpoint of a main Pascal program, a local variable within a subprogram is visible and unprotected, as only the scoping conventions of the language make the variable inaccessible to the main program. As another example, the variable *MyVariable* declared on line 3 of the Ada package specification shown in Figure 1 is visible and protected. It is visible because it is available by name to any part of the system which *with*s this package, and protected because its internal structure is not available outside the package. Finally, type *MyType2* on line 6 is hidden and protected, as neither its name nor its structure are visible outside the package. The reason for making a point of using this terminology is twofold. First, these concepts are language-independent, even though they have been implemented to various extents in different languages. However, as the implementations are generally not pure, we do not wish to use terms of a specific language, in order to avoid the implication that we are referring to a specific language's implementation of one of these concepts. Second, in discussing the mechanisms of slicing, there are considerations based upon a program element's visibility and protection status before and after the slicing transformation. Since these considerations are language-independent, it is important that colorations from existing language implementations not creep into the discussion.

```
1 package MyPackage is
2   type MyType1 is private;
3   MyVariable: MyType1;
4   ...
5 private
6   type MyType2 is ...;
7   ...
8 end MyPackage;
```

Figure 1 Example for visibility and protection

2.2 General Theoretic Concepts and Definitions

In this section we present a fair amount of notation and terminology, not out of any claim that we are making a contribution, but solely to form a basis for the discussions in this paper. The literature contains a wide variation and disagreement in the notation used to represent the concepts and structures in this section; we explain our usage here because of this lack of consensus. The usage presented here does not come from any single source but rather is a personal blending of ideas from many sources. Two sources which were of sufficient help here to warrant mention are [12,21].

2.2.1 Sets, Graphs, and Trees

For subset[2] notation, we use $A \subseteq B$ to indicate that set A is a subset of, and possibly the same as, set B, and $A \subset B$ to indicate that A is a proper subset of B so that $A \neq B$. The cardinality of A is denoted by $|A|$. A graph[3] is a pair (N,A) where N is a finite nonempty set of *nodes* or *vertices*, and $A \subseteq N \times N$ is a set of directed *edges* or *arcs* between nodes. The edge denoted (x,y) or $x \rightarrow y$ leaves the tail or source node x and enters the head or target node y, making x a *predecessor* of y, and y a *successor* of x. The number of predecessors of a node is its *in-degree*, and the number of successors is its *out-degree*. Since the edges are members of a set, a graph may have at most one edge from a given node x to another node y. A structure which allows multiple edges from one node to another is a *multigraph*. In other words, a graph consists of a set of nodes and a set of edges, while a multigraph consists of a set of nodes and a bag of edges. A *path* from x_1 to x_k is a sequence of length k of vertices $(x_1, x_2, \ldots, x_k)$ with $x_i \in N$, $1 \leq i < k\text{-}1$ such that each pair $x_i \rightarrow x_{i+1} \in A$. Two graphs G_1 and G_2 are *isomorphic*, denoted $G_1 = G_2$ iff there exists a one-to-one correspondence between their sets of nodes and a one-to-one correspondence between the sets of edges such that the corresponding edges also agree on the corresponding source and target nodes.

A graph is a tree if it satisfies the three properties:

1 There is exactly one node, called the root, with in-degree 0.
2 Every node except the root has in-degree 1.
3 There is a unique path from the root to each node.

For a tree, predecessor and successor are usually called *parent* and *child* respectively. The transitive and reflexive closures of the parent and child relations are the *ancestor* and *descendant* relations, respectively.

2.2.2 Partial Orderings and Lattices

A reflexive, antisymmetric, transitive relation on a set S is a *partial ordering*, denoted by $\sqsubset$. The pair $(S,\sqsubset)$ is a partially ordered set, or *poset*. For a given poset $(S,\sqsubset)$, $\sqsubset$ denotes the reflexive reduction where $\sqsubset = \sqsubset - \{(x,x) | x \in S\}$. Given a poset $(S,\sqsubset)$ with $a,b \in S$, then a *join* or *least upper bound* of a and b is an element c:

$$c \in S \mid a \sqsubset c \wedge b \sqsubset c \wedge \neg\exists x (x \in S \wedge a \sqsubset x \sqsubset c \wedge b \sqsubset x \sqsubset c)$$

Similarly, a *meet* or *greatest lower bound* of a and b is an element c such that:

$$c \in S \mid c \sqsubset a \wedge c \sqsubset b \wedge \neg\exists x (x \in S \wedge c \sqsubset x \sqsubset a \wedge c \sqsubset x \sqsubset b)$$

If a and b have a unique join, it is denoted $a \sqcup b$; a unique meet, $a \sqcap b$. A set of pairwise incomparable elements of a poset is called a *cochain*.

---

2 Unless specified otherwise, all sets herein are finite.

3 Unless specified otherwise, all graphs herein are directed graphs.

A *lattice* L is a poset, every pair of elements of which have a unique join and meet; the lattice is denoted by the triple $L = (S,\sqcup,\sqcap)$. An element a of a lattice L is a *minimal element* if there does not exist an element b of L such that $b \sqsubset a$. A minimal element a is also a *least element* if $a \sqsubset b$ for every b in L. If L has a least element, it is unique. Similarly, a is a *maximal element* if there does not exist a b in L such that $a \sqsubset b$, and a unique maximal element a is a *greatest element* if $b \sqsubset a$ for every b in L. Each element of the poset is said to be contained in a node of the lattice. Sometimes the node and the element it contains are used interchangeably.

The *power set* of a set S, denoted 2^S, is the set of all subsets of S, i.e., $2^S = \{T \mid T \subseteq S\}$. A particular lattice structure of interest in this paper is the following. Given a finite set S and the usual set union and intersection operations denoted by $\cup$ and $\cap$, the poset $(2^S,\subseteq)$ is the basis for the lattice $(2^S,\cup,\cap)$. Each node of this lattice contains a unique subset of the elements of 2^S. We will often refer to this structure as a subset inclusion lattice. This lattice is of interest here because if S is the set of all statements of a program then 2^S is the set of all subsets of the program statements. Since a slice is a subset of program statements, then every slice corresponds to an element of 2^S, and thus to a node in the lattice $(2^S,\cup,\cap)$. This lattice is therefore a convenient structure for discussing slices of a program and their relationships.

This lattice may be depicted using a Hasse diagram as a graph in which the greatest element, the set S, is a node of in-degree 0, and the least element, $\varnothing$, is a node of out-degree 0. An edge $a \rightarrow b$ is drawn in the diagram iff $b \subseteq a$ and there does not exist a node c such that $b \subseteq c \subseteq a$. In this case, a is considered the parent of b, and b the child of a. Notice that this excludes the possibility that two separate nodes in the lattice contain the same element. For example, given the set $S = \{1,2,3\}$, the lattice $L = (2^S,\cup,\cap)$ is shown in Figure 2.

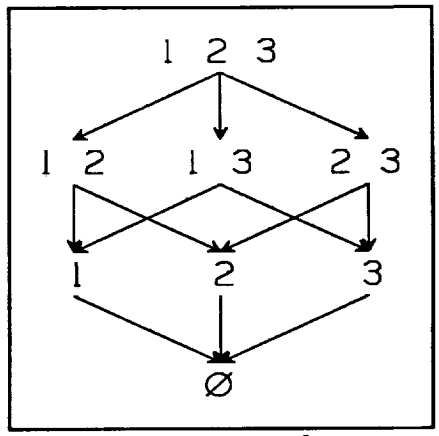


Figure 2 Power set lattice for {1,2,3}

Given a poset $(A,\sqsubset)$, the relation $\sqsubset$ defines a set of ordered pairs of elements of A. Given a set $B \subseteq A$, then some of the ordered pairs of $\sqsubset$ may also be ordered pairs of elements of B. The set of those elements of $\sqsubset$ which consist of ordered pairs of elements of B is called the *restriction* of $\sqsubset$ to B, and is a partial ordering of B. If the poset $(A,\sqsubset)$ is a lattice, the new poset $(B,\sqsubset)$ is not necessarily a lattice. However, if B includes at least the greatest and least elements of A, then B must also be a lattice. For example, let A be 2^S as in the example above, so that the poset $(A,\subseteq)$ is shown in Figure 2, and let $B = \{\{1,2\},\{2,3\}\}$. Then $(B,\subseteq)$ is a poset, but is not a lattice. However, if we consider B' as B augmented with the greatest and least elements of A, so that $B' = \{\varnothing,\{1,2\},\{2,3\},\{1,2,3\}\}$, then $(B',\subseteq)$ is a lattice.

2.2.3 Flow Graphs

A *flow graph* is a quadruple $(N,A,start,stop)$ where (N,A) is a graph, $start \in N$ is a distinguished node of in-degree 0 called the *start node*, $stop \in N$ is a distinguished node of out-degree 0 called the *stop node*, there is a path from *start* to every other node in the graph,

and there is a path from every other node in the graph to *stop*.[4] If x and y are two nodes in a flow graph, then x *dominates* y iff every path from *start* to y includes x; y *post-dominates* x (i.e., x is post-dominated by y) iff every path from x to *stop* includes y. The dominance relation is a partial ordering; every node except *start* has a unique nearest dominator. The graph of the reflexive and transitive reduction of the dominance relation is a tree, called the dominance, or dominator, tree. Finding the post-dominators of a flow graph is equivalent to finding the dominators of the *reverse* flow graph, the graph in which the direction of every edge is reversed and the *start* and *stop* labels are exchanged.

2.2.4 Dependences

According to Podgurski and Clarke [41], *dependences*[5] are relationships among program statements and are of two types, control and data flow (or simply data) dependences. In a program, two types of situations create dependences between two statements, or between a statement and a predicate. In Figure 3, a *control dependence* exists between the predicate A on line 2 and the statement B of line 3; the execution of B is control-dependent on the value of A because the value of A immediately controls the execution of B.

```
1 begin
2   if A then
3     B;
4   end if;
5 end;
```

Figure 3 Control dependence

In Figure 4, the assignment statement on line 3 is *data dependent* on the assignment statement on line 2, because the correctness of C's value in line 3 depends upon the prior execution of the statement on line 2. Thus a data dependence exists between two statements when a variable in one may have an incorrect value if the order of execution of the two statements is reversed. Another way of stating this is that one statement is data dependent upon another if data can potentially flow from the latter to the former in a sequence of assignment statements.

```
1  begin
2    C := f(D);
3    E := C;
4  end;
```

Figure 4 Data dependence

2.3 Introduction to Program Representation

In one strict view, only a set of machine instructions in a computer's memory can be termed a "computer program". But this strict interpretation is usually relaxed so that various program representations are spoken of as being, or being equivalent to, computer programs. Common program representation schemes include high-level source code, pseudocode, and flow charts; the purpose of these various representation forms depends upon the context and may include human readability, annotation for verifiability, and transformation for application to a different platform such as a parallel multiprocessor. In the context of program slicing, program representations are used to facilitate the automation of slicing. For a very simple program, a slice can be prepared by hand. But with

---

4 Some authors use *entry* and *exit* for *start* and *stop*, respectively; some authors drop the requirement for a *stop* node and define a graph as (*N,A,start*).

5 Some authors use *dependency*, singular, and *dependencies*, plural.

increasing size and complexity of the program, there is increasing need to employ the assistance of automation. As discussed in Section 3, current automated slicing techniques require that information gleaned from a source code form of the program to be sliced be transformed into some different program representation during the slicing process. Various program representation schemes have resulted from the search for ever more complete and efficient slicing techniques.

In the discussions of program representations which follow, it is important to remember that there is no single correct way of building, say, a control flow graph, nor is there a single exact set of information which must be available for slicing. Each researcher presents a different technique or algorithm, according to the needs of the problem at hand. Nevertheless, there is general agreement as to the class of information to be contained in each type of program representation for the purposes of slicing. The various representations shown here are illustrative of these general agreements, but are not necessarily faithful to any single researcher's style.

A program which is to be modeled with one of these representations is written in some language. While this discussion concentrates on executable languages, we do not wish to exclude the possibility of including non-executable forms such as pseudocode or formal specifications. Each language has its own peculiarities which affect the way it can be represented, and the form of the representation. In the explanations of the different representation mechanisms below, it is useful to keep in mind the differences in languages. For example, C has a switch statement structure which allows multiple exits, but C has no nested subprograms; Pascal has a regular, partitioning, single-exit case statement, but also has nested subprograms; FORTRAN has an equivalence statement parameter passing mechanism which allows variable aliasing by array overlap; Ada has various synchronization mechanisms for tasking. There probably is no perfect universal program representation scheme because each of these language features may call for a somewhat different representation mechanism. Conversely, a program representation may well serve to bridge the gap between disparate languages. For simplicity and for broad applicability of results, most of the research cited herein has developed slicing based on the intersection of the features of the traditional procedural languages FORTRAN, C, Pascal, and Ada.

It is common to represent programs as graphs and lattices pictorially with closed shapes standing for nodes and directed lines representing edges. For simplicity, in fact, the picture is often spoken of as "being" the graph or lattice, or the graph as "being" the program, but it is important to keep in mind that the picture or graph drawing is only a representation of an abstract mathematical or programmatic entity. The model may be imbued with a set of desired semantics, with the nodes and arcs drawn in various shapes and given various labels, provided that there is an unambiguous and consistent mapping between the semantics and the model such that the mathematical or syntactic integrity of the model is maintained. In this case, results derived from mathematical proofs and manipulations on the model give strong credence to the corresponding semantics.

2.4 Control Flow Graph

For the reasons stated in the sections above, it is desirable to represent or model a program with a flow graph. To do so, we must provide a mapping from the program to the flow graph. An example of such a mapping is the partial graph grammar of Figure 5, adapted from Hecht [21]. This example grammar uses syntax-directed translation to map an abstract program into a program flowchart. In this case, the resulting flow graph is a traditional flowchart, with nodes being represented by closed figures of various shapes to indicate their function in the program.

A flowchart is used to depict the flow of control in a program, hence the name *control flow graph*. The nodes represent program statements, predicates, and branch targets, while the edges represent potential control transfers among the nodes. Notice that a flow graph generated by the grammar of Figure 5 is strictly an intraprocedural control flow graph, as no provision is made to represent control passing to a called subprogram or returning from a subprogram. Also notice that an edge $a \rightarrow b$ does not mean that control *must* transfer from a to b during program execution, but only that it *might*, depending upon the input and the program state. Control flow graphs and the control flow analysis used to generate them are well-studied in the literature; for structured programs a single pass through the source code is sufficient for their construction. Constructing flow graphs for unstructured programs which allow arbitrary gotos can be more difficult (see, for example, Wolfe [55]). For this reason, most researchers (e.g., [8,24,45]) confine themselves to structured languages, or structured subsets of languages which do allow non-structured constructs such as the exception mechanism of Ada.

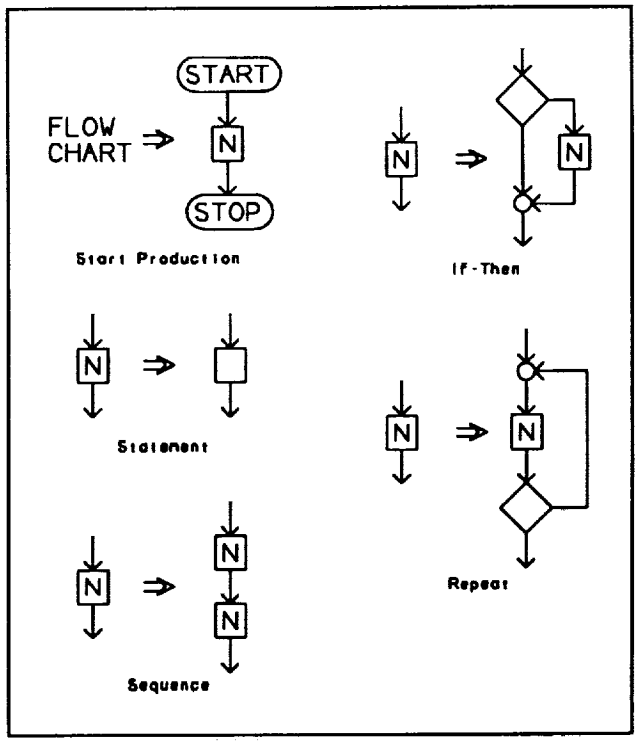


Figure 5 A simple graph grammar

As an example of a control flow graph, consider the program fragment in Figure 6. Figure 7 shows a control flow graph for this program, with nodes labeled to correspond to the line numbers of Figure 6. Line 7 of the program (*end if*) does not contain a statement, but does represent the point at which control from the two branches of the *if* statement rejoin to leave the *if* statement which begins on line 3. Since it is necessary for showing control flow, it is given a node (node 7) in the control flow graph. Similarly, the *begin* on line 1 and the *end* on line 9 represent the absolute start and end of control flow, respectively, so they have nodes in the control flow graph. (They also serve as the start and stop nodes of the flow graph.) In contrast, the *else* on line 5 does not have any control flow (or data flow, see below) significance. Rather, it is simply the syntax mechanism for

```
1 begin
2    input(x);          def(2)={x}
3    if (x=0) then      ref(3)={x}   rd(3)={(x,1)}
4      y := f1(x);      def(4)={y}   ref(4)={x} rd(4)={(x,1)}
5    else
6      y := f2(x);      def(6)={y}   ref(6)={x} rd(6)={(x,1)}
7    end if;
8    output(y);         ref(8)={y}   rd(8)={(x,1),(y,3),(y,4)}
9 end;
```

Figure 6 Program example for flow graphs

separating the two parallel clauses of the *if-then-else* statement. Thus the *else* line has no node in the control flow graph.

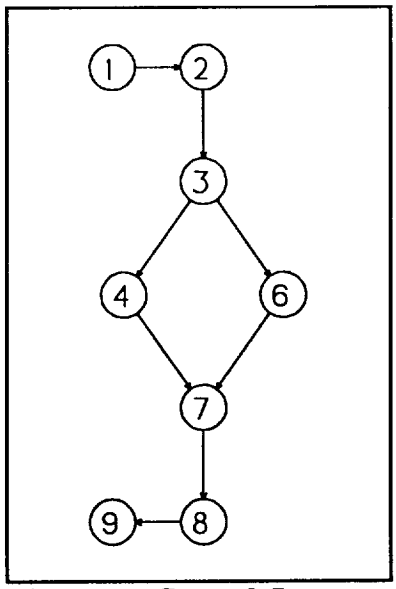


Figure 7 Control flow graph for Figure 6

The node with the largest out-degree is node 3, while that with the largest in-degree is node 7; these correspond to the beginning and end of the *if-then-else* statement. The degree of these nodes is 2, because they model the control flow of a two-way branch. For some applications there would be labels *T* and *F* on the two edges leaving node 3, corresponding to the possible values of the branch predicate. An *n*-way *case* statement in Pascal would be modeled very simply with a pair of nodes of degree *n* above and below the *n* nodes representing the *n* different cases, with the *n* edges leaving the case predicate node optionally labeled with the possible values of the predicate. Because C has a more complex *case* structure in which the use of *break* is optional, the control flow graph modeling a C switch statement is more complex than for Pascal. Specifically, the node corresponding to the predicate of an *n*-way switch with a *default*, counting the *default* as one of the *n* possibilities, must have out-degree *n*, while the predicate node of an *n*-way switch without a default will often have out-degree *n*+1.[6] The in-degree of the node corresponding to the closing brace of the *switch* statement will be in the range from 1 to *n*+1 inclusive, depending upon the arrangement of *breaks* within the *switch* statement.

For example, consider the C program in Figure 8 and a corresponding control flow graph in Figure 9. Node 2, corresponding to the *switch* statement, has out-degree of 5 rather than 4 as would a *case* node in a Pascal program. However, node 9, corresponding to the closing brace on line 9, is of in-degree 4, not 5. This is because node 3, corresponding to case 0 on line 3, has no *break*. Therefore control cannot flow directly from node 3 to node 9, but goes from node 3 to node 4. Since there is no range information for the function

---

6 The node will have out-degree *n*+1 if the *n* choices of the switch fail to cover the range of the selector, and out-degree *n* if the choices of the switch do cover the range of the selector.

a(), there must be an edge from node 2 to node 9, to account for the possibility that *a()* may return a value not included in the discrete range 0..3. An interesting feature of a *break* statement is that it corresponds not to a node in the control flow graph but to an edge. This is because a *break* serves to transfer control directly from the most recent statement or predicate to the closest closing brace. For example, the *break* on line 7 corresponds to the edge (7,9), and indicates that control may pass from the predicate function *e()* of node 7 to the closing brace of node 9. Ada's *case* is different from both C and Pascal as Ada requires that coverage of the range of the selector by the set of choices be determinable at compile time.

```
 1  main () {
 2    switch (a()) {
 3      case 0: b();
 4      case 1: c();
 5              break;
 6      case 2: d();
 7              if (e()) break;
 8      case 3: f();
 9    } /* end switch */
10  }
```

Figure 8 C case statement

Very few slicing researchers treat *case* statements; a few indicate that since a series of *if-then-else* statements can be proven equivalent to any *case* statement, their treatment of *if-then-else* is sufficient to guarantee that their algorithms will slice a *case* statement after suitable code transformation. Nowhere in the literature, however, is there an attempt at a complete treatment of the slicing of *case* statements.

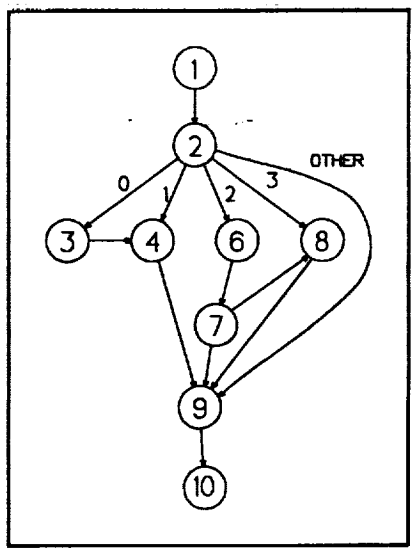


Figure 9 Control flow graph for Figure 8

2.5 Data Flow Graph

While a control flow graph as described above can be used to capture some information about a program, particularly its looping structure, there is much program information which a control flow graph cannot contain. In particular, program slicing as described in Section 3 cannot be performed by using solely the information in a control flow graph. Some of the information needed for slicing is contained in a *data flow graph*, which holds information about a program's variable definitions and references. In current practice, data dependences are usually computed after first constructing various sets of statements which define, reference, or use specific sets of variables.The following definitions are necessary before discussing the data flow graph proper.

A program variable is said to be *defined* (not related to a variable *declaration* in the programming language sense) in a statement if the execution of that statement can alter the value of the variable. Typically, variables are defined by assignment and input statements. Conversely, if a variable is not defined in a statement, it is *preserved* in that statement. A program variable is *referenced* in a statement if the value of that variable is

used during the execution of the statement.[7] Associated with every statement s is a set *def*(s) of variables which are defined in s. Similarly, there is a set *ref*(s) of variables referenced in s. The values of variables in *ref*(s) immediately before s is executed determine the values of *def*(s) immediately after s is executed. If d is a definition of variable v in a statement represented by node s in a control flow graph, that definition *reaches* node t iff there exists a path from s to t which contains no definitions of v other than d in s.[8] Thus there is associated with each node t a set of *reaching definitions*, that is, a set of variable definitions which reach t. The set $rd(t)$ is a set of pairs (v,s), where v is a variable defined at node s, which definition reaches t. Note that (v,s) can be a member of $rd(t)$ only if there is a path in the control flow graph from s to t. Data flow analysis is used to determine the def, ref, and rd sets, and they are represented in a data flow graph.

An example will serve to illustrate these information sets. Consider again the program fragment in Figure 6 on page 10, which features various definitions of and references to variables x and y. The program fragment in the figure is also annotated with variable references, definitions, and reaching definitions, and has line numbers added. For this example, we assume that the functions *f1* and *f2* in lines 4 and 6 are free from side effects with respect to x. That is, x is preserved in lines 4 and 6.[9]

Figure 10, with the same node labeling, shows the reaching definitions of this program. In the program of Figure 6, for example, note that the definition of x in line 2 reaches lines 3, 4, 6, and 8. This is shown in Figure 10 by the edges from node 2 to nodes 3, 4, 6, and 8. This means that $rd(3) = rd(4) = rd(6) = \{(x,2)\}$, and $rd(8)$ includes $(x,2)$. However, nodes 1, 7, and 9 have no reaching definition edges. These nodes were strictly associated with control flow and have nothing to do with data flow.[10] As in the case of the control flow graph above, there is no node in the graph of reaching definitions corresponding to line 5 of the program, as that line has syntactic meaning only. The definitions of y in lines 4 and 6 both reach the reference of y in line 8, so with the inclusion of the reaching definition of $(x,2)$ from above, $rd(8) = \{(x,2),(y,4),(y,6)\}$. Note that in a single execution of this program, only one of lines 4 and 6 can possibly be executed. Nevertheless, in this static data flow analysis, which by definition does not contain information about the input stream, either of lines 4 or 6 could be executed, and so both definitions reach line 8.

---

7 A variable may be both defined and referenced in a single statement, or may be both preserved and referenced, but cannot be both defined and preserved in a single statement.

8 This assumes that there is only a single definition of v in any node x. If there are multiple definitions, then d is assumed to be the last definition of v in x.

9 It is impossible to tell that x is preserved in these lines by the information in Figure 6; it is assumed here for simplicity. Only by examining either the complete source code or the formal specifications for the two functions is it possible to guarantee that they produce no side effects on x. This anticipates the discussion of the difficulties of interprocedural flow analysis.

10 In fact, they need not even appear in a data flow graph, but are included here for comparison with the control flow graph above.

Given the references and reaching definitions of each statement, a *data dependence* is defined as an edge from a node s where a variable v is defined to a node t where v is referenced provided that the definition at s reaches t. That is, $dd(P)$, the set of data dependences for program P, is a set of edges (s,t):

$$dd(P) = \{(s,t)\} \mid \exists v [v \in def(s) \wedge v \in ref(t) \wedge (v,s) \in rd(t)]$$

Figure 11 shows the data dependence graph for the program of Figure 6. It is very similar to the graph of reaching definitions in Figure 10, but does not have the edge from node 2 to node 8. This is because, even though the definition of x at node 2 reaches node 8, x is not referenced at node 8, and thus there is no data dependence based on x between nodes 2 and 8.

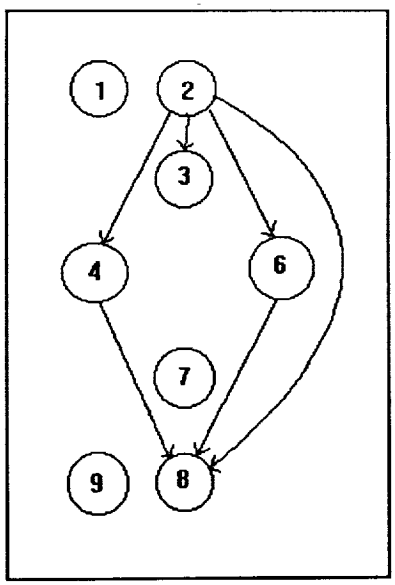


Figure 10 Reaching definitions for Figure 6

In program representations for conventional slicing, as well as for standard compiler-based transformations such as loop-unrolling optimizations and constant propagation, all control and data dependence analysis is performed at the statement or expression level. All the sets of discussed above consist of sets of variables or statements, and similarly refer to variables or statements.

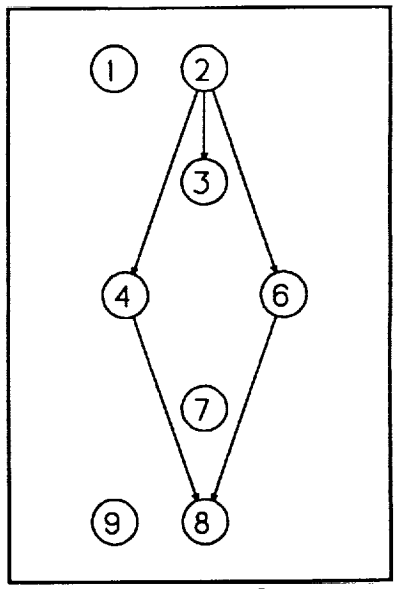


Figure 11 Data dependences for Figure 6

Traditionally, data flow analysis for such purposes as program analysis and compiler optimization has been concerned with how data flows through a program as the execution of the program progresses. That is, it is concerned with how the data values propagate through a program. In slicing for debugging, however, the interest is not in where the data is going, but in where it came from. If a program outputs an erroneous value, indicating a bug, the interest is in working backward to find the source of the erroneous value, not in looking forward to see where the erroneous value will go next. Therefore, some researchers (e.g., Agrawal and Horgan [3]) reverse the direction of the edges in flow graphs, producing a data structure in which data flow can be traced backwards to locate the origin of an erroneous value. Ottenstein and Ottenstein discuss the value of a doubly linked flow graph implementation, to be suitable both for tracing forward flow for optimization and for tracing backward flow for slicing [38] (see also the discussions of the PDG in Sections 2.7 and 3.3.1).

2.6 Annotated Flow Graph

At the time that program slicing was being developed in the 1970s, control flow and data flow graphs were well known. Hecht [21], for example, presented a number of algorithms

for generating flow graphs for various purposes. One problem with these graphs, however, is that they contain no information about the form of the original source code from which they were generated. The transformations between code and flow graphs were one-way, with no mapping from the graph back to the source. A slice, according to the original definition, is a set of source code statements, not a set of nodes and edges in a flow graph. Therefore a slicing procedure which is based on a flow graph requires data structures which contain not only control and data flow information but original source code information as well.

Such a data structure is called an annotated flow graph. In general, an annotated flow graph is a regular flow graph as described above augmented with more information than is necessary for strictly graph-theoretic manipulations of the graph. The augmented information set allows inferences in program semantics to be made based on the mathematical manipulations and analyses of the graph. For slicing, it is common for the nodes to contain the text of the source code statement to which the node corresponds, or a pointer to the statement, perhaps as a (*file, line number*) pair. For example, if the node labels in Figure 11 were known to be strict line number references to the source code text of Figure 6, then the graph in Figure 11 could be considered to be an annotated data dependence graph. For slicing, the annotation needs to be sufficiently rich to allow a slice in the form of syntactically correct source code text to be created from the subset of nodes and edges which the slicing algorithm, working on the annotated flow graph, produces.

2.7 Program Dependence Graph

In 1984, Ferrante, Ottenstein, Ottenstein, and Warren presented their version of a program representation mechanism called the *program dependence graph* [15,38]. Unlike the flow graphs discussed above, a feature of the PDG is that it explicitly represents both control and data dependences in a single program representation. Since efficient, minimal slicing requires information about both kinds of dependences, the PDG has been adopted as an excellent representation for use with slicing algorithms.

The PDG models a program as a graph in which nodes, as above, represent statements and predicates, but the edges represent both data dependences and control dependences. The data dependences are among the variables in the nodes and the control dependences are those on which the execution of the statements in the nodes depend. Strictly speaking, the PDG is a multigraph which can be simplistically viewed as the union[11] of a pair of subgraphs sharing a common set of nodes, one the control flow subgraph, and the other the data dependence subgraph. Since there may be more than one edge between a given pair of nodes, one edge representing a control dependence, one a data dependence, the PDG is a multigraph.

The PDG represents control flow in a more sophisticated way than the previous forms discussed. One of the problems of a standard control flow graph is that it can be very difficult to generate sequential code from an arbitrary control flow graph, but that is a specific requirement of a graph-based slicer. As the final step in the slicing process, the

---

11 In this form of graph union, we have set-union of nodes and bag-union of edges.

slicer must regenerate syntactically correct code from an arbitrary sliced graph. Therefore, rather than using naive control flow as described above for the control subgraph, the PDG incorporates control dependences into its control subgraph. Every edge in a PDG represents a dependence.

2.8 System Dependence Graph

The PDG has a number of desirable characteristics for slicing. However, the PDG as described above does a poor job of handling procedure calls, because the PDG form was developed largely for use in optimization transformations [15]. Optimizations such as branch deletion or loop unrolling never cross procedure boundaries; at most, optimizations transform procedure calls into inline substitution of the procedure body. But since the PDG showed promise for slicing, it was extended by Horwitz, Reps, and Binkley into the *system dependence graph*, which models a main program together with a set of non-nested procedures [25]. This graph is very similar to the PDG; indeed, a PDG of the main program is a subgraph of the SDG (i.e., for a program without procedure calls, the PDG and SDG are identical). The technique for building a SDG consists of first building a PDG for each procedure (functions are modeled as procedures with an extra parameter), including the main procedure, and then adding auxiliary dependence edges which link the subgraphs together. This results in a program representation which includes the information necessary for slicing across procedure boundaries.

The additional dependence edges which link the individual PDGs into an SDG are of the same two types as previously discussed: control dependence edges and data dependence edges. Control dependence edges link every call site of a procedure with the entry point of the subprogram. The authors accomplish data flow analysis by invoking two temporary variables and four graph nodes for each procedure parameter. One node is attached to the calling procedure's graph to represent the value of the calling procedure's actual parameter being copied into one of the temporary variables, while one node is attached to the called procedure's graph to represent the value of the temporary variable begin copied into the called procedure's formal parameter. The other temporary variable and pair of nodes are used at the called's procedure's return to transfer values back to the calling procedure. Then for each parameter, a data dependence edge links each corresponding pair of nodes, completing the linkage of individual PDGs into a single SDG. In this way the SDG models the exact calling context of each procedure call in the system.

3 An Overview of Conventional Slicing

This section presents a high-level description of program slicing and includes a history of the development of slicing, followed by a consideration of attempts to assure the mathematical basis and integrity of the models used. While there is some discussion of the formal technical aspects of slicing, we do not attempt to give a full presentation and analysis of every slicing algorithm. Such treatments may be found in the references cited.

3.1 Weiser Slicing

3.1.1 Introduction

Weiser first presented program slicing in his doctoral dissertation, "Program slicing: formal, psychological, and practical investigations of an automatic program abstraction method" [50]. His original impetus was to develop slicing as an aid to program debugging, inspired by the abstraction mechanisms used by programmers in analyzing existing programs[12] while performing corrective maintenance to debug code.

Before Weiser's work, all abstraction mechanisms to date had decomposed programs into "units" by grouping *sequential* program elements. At the lowest level, a raw dump of an existing executable program consists of a very large number of homogeneous units, in this case bytes, far too many for the mind to grasp. The process of understanding a program consists of organizing this very large number of units into fewer units by some mechanism, allowing the program to be viewed at a higher level of abstraction. For example, a disassembler accomplishes this by producing a sequence of assembly language statements from the raw dump, replacing many sequential bytes of the dump with a single assembly language statement. At a higher level, a compiler allows the human to deal with one source language statement instead of many assembly language statements. Further, a hierarchical organization can be imposed on source statements by grouping many sequential statements into subprograms, and then subprograms into packages or abstract data types, and so on.

After a series of experiments in which he studied the behavior of programmers who were attempting to comprehend and debug FORTRAN programs [50,52], Weiser concluded that while the abstraction and grouping of sequential sets of statements did serve to aid in program understanding, programmers who were specifically attempting to debug a program used a different mental abstraction mechanism for grouping program statements. In particular, they used an abstraction mechanism which grouped generally *non-sequential* sets of statements. Weiser concluded that the statements grouped in this way were those which applied to what he called "units of data components," which range in size from simple variables to the data storage portions of large abstract data types. Rather than looking at sequences of program statements, he found that he could understand the mental abstraction of the debugging process by examining data flow diagrams of the programs. He called his theory of data behavior abstraction *slicing*, and the units of abstraction, *slices*. The slices abstract a program based on the behavior of a program with respect to a specified set of data components — variables — rather than with respect to a sequential statement listing.

Informally, Weiser defined a slice as a projection of a specified subset of behavior from the original behavior of the program. A slice is a complete program which contains a subset of the statements of the original program, and which performs a subset of the computations performed by the original program. The slice is obtained by removing

---

12 All work on slicing to date has considered only procedural languages. All discussions in this paper will assume conventional languages such as FORTRAN, C, Pascal, and Ada unless explicitly noted otherwise.

statements[13] from the original program which do not affect the specified behavior of interest. A slicing algorithm must ensure that 1) the slice contains only a subset of the original program's statements, and 2) the behavior of the slice is also a subset of the original program's behavior. Weiser defined behavior for this purpose as the sequence of values taken on by a variable during the course of a program's execution. In other words, if the value of variable X at some statement a is the behavior of interest, both the slice and the program must compute the same value for X at a, assuming identical input. Weiser used the term *slicing criterion* for the specification of the behavior of interest. His slicing criterion for a program consists of a set of statements of the program and a set of program variables of interest at those statements. A slicing criterion is only meaningful in relation to a specific program. A slice results from applying a specific slicing criterion to a specific program. Weiser's slicing is characterized by performing static analysis on unexecuted source code. The form of slicing which Weiser invented thus became known as *static* slicing.

The reason that slicing is an aid to debugging is exactly because it projects out of a program just those statements which contribute to the computation of the sequence of values taken on by a variable. A bug is evidenced by incorrect program behavior, usually noticed as an incorrect value of a variable.[14] Since every statement which could have influenced the value of the variable is in the slice, then only the statements in the slice must be examined to find the offending statement which caused the erroneous value; the rest of the program need not be considered. Since the slice is no larger than the original program, it is easier to find the offending statement in the slice than in the original program. It also follows that the fewer the number of statements in the slice, the easier it is to find the offending statement.

The requirement that a slice must be compilable, a syntactically correct executable program in its own right, is partly due to the operational requirement in the definition which states that a slice must compute the same sequence of values for a set of variables as did the original program. The satisfaction of this requirement can only be demonstrated if the slice is executed with a specific input, and the results observed. The requirement that a slice be executable is also critical for some important applications of slicing, such as parallelization and reuse component generation. However, for some applications, such as design recovery and module cohesion metrics, the requirement is not necessary to enforce.

The definition of slicing above says nothing about the size or uniqueness of slices. In general, there may be many slices which correspond to a given slicing criterion applied to a given program. The issues of slice size, and the fact that small or minimal slices are generally of the most interest, is discussed more in Section 3.1.2 below. Intuitively, however, Weiser argued that a small slice is most interesting because slicing consists of

---

13 The removal of zero statements is allowed by the definition.

14 In the context of debugging with slicing, a bug only exists if it is observed. If a program never gets the input which causes an erroneous value for a variable, the program is considered bug-free. This is a test-oriented viewpoint rather than a verification-oriented one.

throwing away statements which have no influence on the remaining slice. Slicing "successfully" consists of throwing away as many statements as possible, yielding a small slice, while still meeting the requirements of behavior. We also note here that at least a trivial slice always exists for any given program and slicing criterion. This slice is the original program itself; it is generated by removing no statements from the program.

As an illustrative example of static slicing, consider the Pascal program in Figure 12 (this example was adapted from [36]). This program computes the sum and product of the first n natural numbers. If we are interested in the behavior of this code with respect to variable *sum* at line 14,[15] we use the slicing criterion $C = \langle 14, sum \rangle$. That is, we are interested in the value of *sum* after the completion of the do-loop. A slice which corresponds to this slicing criterion applied to program *Sum_Product* is shown in Figure 13. As this example illustrates, the variable *prod* does not appear in this slice, because *prod* does not contribute, either directly or indirectly, to the value of *sum*. That is, there is no data dependence between *prod* and *sum*. However, *i*, *n*, and the control structure of the do-loop, are retained in the slice, as they do affect the value of *sum* in the statement at the (original) line 14. There are various data and control dependences among these program elements. Also retained in the slice are elements of the program's syntactic structure such as *begin*s, *end*s, and semicolons, as they are necessary for the syntactic correctness of the slice as an executable program in its own right.

```
 1  program Sum_Product;
 2
 3  var
 4    i, n, sum, prod: integer;
 5
 6  begin
 7    read (n);
 8    sum  := 0;
 9    prod := 1;
10    for i := 1 to n do begin
11      sum  := sum  + i;
12      prod := prod * i;
13    end;
14    write (sum, prod);
15  end.
```

Figure 12 Program Sum_Product

While this description of slicing is straightforward and intuitive, an attempt to implement a slicer based on it quickly uncovers a number of problems or issues which complicate the process. The following subsections examine some of these problems and issues.

3.1.2 Problems With Undecidability

The above description of slicing fails to address two problems of undecidability which arise in slicing. The first arises because the description of slicing includes the requirement that the behavior of the slice is a subset of the behavior of the original program. In other words, a slicing algorithm must in effect guarantee the equivalence of (portions of) two arbitrary programs. This is equivalent to the halting problem, and is thus undecidable. This problem can be addressed by requiring that a slicing algorithm need only produce correct slices for the cases in which the program being sliced is known to

15 There is a distinction between "statement 10" and "the statement which appears on line 10". We will always employ the latter usage.

halt. In this view, a slice is considered to be correct if, given the same input, both the program and slice halt, and produce the same sequence of values for the variables named in the slicing criterion before halting (see also Section 3.5).

```
 1  program Sum_Product_1;
 2
 3  var
 4    i, n, sum: integer;
 5
 6  begin
 7    read (n);
 8    sum := 0;
10    for i := 1 to n do begin
11      sum := sum + i;
13    end;
14    write (sum);
15  end.
```

Figure 13 Program Sum_Product sliced on ⟨14,*sum*⟩

The second undecidable problem is the problem of generating a minimal slice, which is also equivalent to the halting problem. Weiser used the term *statement minimal slice* to mean a slice which is a proper subset of every other possible slice for that slicing criterion. Knowing at the outset that finding the statement minimal slice of an arbitrary program for a given slicing criterion is an undecidable problem, Weiser developed an algorithm for approximating what he called *data flow minimal* slices [50]. That is, he developed an algorithm which would produce the smallest slices that can be produced solely by flow analysis of source code. We have already stated that small slices are the most interesting from the point of view of slicing technique. Small slices are also more useful for debugging, because the smaller the slice which contains the bug, the less code the programmer must examine to find the bug.

To show that generating a minimal slice is equivalent to the halting problem, Weiser argued as follows [50]. Consider the Pascal program in Figure 14 and the slicing criterion ⟨13, *x*⟩. If *f* is a constant-zero function then the value of *x* in the slice will be 1 for all inputs. The smallest slice which can produce this behavior is the program:

```
begin; x:= 1; end.
```

However, determining whether an arbitrary total recursive function is a constant-zero function [22] is undecidable. Thus there is no slicing algorithm guaranteed to produce the minimal slice above. Another way of stating this is to say that a slicing program with access to an oracle can always produce slices that are no larger than a slicing program without an oracle. A consequence of this is that there is no algorithm which can be guaranteed to produce all possible slices of a program, because clearly the set of all possible slices includes the minimal slice, which no algorithm can be guaranteed to produce.

As discussed in Sections 3 and 4, some of the work on slicing since its introduction has involved efforts to work around and deal with these undecidability problems.

3.1.3 Loss of Criterion Statement

Another problem with the above slicing description is that both the slicing criterion and the desired behavior of the slice reference one or more program statements in the original program. But the slice is formed by deleting statements from the original program. It may happen that an original program statement which is in the slicing criterion does not appear in the slice at all, having been deleted in the slicing process, and so the slicing

```
 1 program p;
 2 var x: integer;
 3 function f (i: integer): integer;
 4   begin
 5   .
 6   . {an arbitrary total recursive function}
 7   .
 8   end;
 9 begin
10   read (x);
11   if f(x) = 0 then
12     x := 1;
13   write (x);
14 end.
```

Figure 14 A program which blocks a minimal slice

criterion is undefined in the slice. To handle this situation, Weiser proposed using the nearest successor to the statement in the original program which is also in the slice. In more recently developed slicing techniques, this has become a moot point, as slicing using a program dependence graph does not have the problem of criterion statement deletion (see Section 3.3.1).

3.1.4 Statement Alteration vs Statement Deletion

The definition of slicing states that the only action which may be performed on a program is the deletion of statements. But the slice shown in Figure 13 on Page 19 cannot have been produced from Figure 12 simply by statement deletion, as an examination of line 4 in both the original program and in the slice will show. The original line 4 is:

```
i, n, sum, prod: integer;
```

while the corresponding line in the slice is:

```
i, n, sum: integer;
```

Here, no statement was deleted. Rather, since *prod* does not appear in the slice, line 4 has been *altered* to produce the slice so that *prod* is absent. This is because the original line 4 can be viewed as a composite of four variable declarations which the syntax of the language permits to be shortened to a single statement. A similar situation exists between the original line 14 of the program and its corresponding line in the slice. The former is an output statement which references both *sum* and *prod*, while the latter, having been altered, references only *sum*. Because of this, slicing techniques which are defined in terms of source code line numbers usually make simplifying requirements such as that each variable declaration be on a separate line, allowing one variable declaration to be sliced away independently of others, and that every input and output statement reference only a single variable. When slicing is based on alternate representations such as flow graphs, each variable declaration will have its own node in the graph, allowing individual variable declarations to be deleted from the graph. For the same sorts of reasons, most researchers assume simplifying restrictions such as excluding multiple statements on a single line of source code and C-style multiple assignments in a "single" statement (e.g., $a = b = c;$). We emphasize that these restrictions are only for simplicity, not for any substantive theoretical reasons. They merely represent messy implementation

details which are normally omitted from prototypes whose main purpose is to explore theoretical issues.

While the question of statement alteration is not an issue in understanding the theory of slicing, in a practical application of slicing it is one of the details which must be managed. Consider the case of a program being sliced to produce reuse repository components. In this case, the name of each new component generated by slicing the original program would have to be different so that the repository management system could keep track of the new programs — the slices — separately from the original program. Notice that the program names *Sum_Product* and *Sum_Product_1* in the two figures are different, reflecting this.

3.1.5 Syntax Problems

Another problem is that blind deletion of lines in a program can quickly lead to a syntactically incorrect slice. One of the requirements of a slice is that it be a compilable program in its own right. If, while slicing a FORTRAN program, all the statements in the *then* clause of an *if-then-else* statement are deleted, the result is syntactically incorrect because FORTRAN doesn't allow an empty *then* clause. This problem has nothing to do with the concept of slicing but is rather a language issue. Some languages allow null *then* clauses, others do not. There are many such details peculiar to each language. To attempt to eliminate these language details from the consideration of the central ideas of slicing, Weiser stopped using the original source code, and instead used an annotated flow graph as the basis for slicing. Subsequent researchers have followed his lead in this regard, using various types of graphs as program representations for slicing, most using some form of flow graph as the program representation. Weiser found the current representation mechanisms not completely sufficient for his needs, and did considerable work in developing new representation mechanisms adequate for performing slicing. Since all subsequent work in slicing has continued this trend, to some extent the development of slicing and program representation mechanisms have been parallel, and the development of slicing techniques has driven development of program representation mechanisms.

3.1.6 Other Issues

Several other difficulties and anomalies involved in slicing brought up by various researchers deal either with similar messy implementation details or with pathologic cases and are very much side issues to the central theme of slicing. These are usually dispensed with by imposing restrictions such as that a program to be sliced be syntactically correct to start with [54], contain no dead code [18], and no uninitialized variables which might have indeterminate values [27]. As these are often considered appropriate restrictions for "good" programming style anyway, we shall adopt them in the present discussion without loss of generality. Nevertheless, these sorts of issues must be dealt with by a production slicer.

Similarly, slicing is greatly complicated by such constructs as branch test expressions and I/O statement expressions which produce side effects when evaluated [45], and aliasing procedure calls in the form of *f(x,x)* [25]. Again, we will not discuss these sorts of cases further.

3.2 Slicing vs Traditional Modularization

There is an important conflict between slicing and all other traditional programming abstraction techniques. From Parnas [39] to the present, a major focus of software engineering technology has been to modularize programs. This focus strives to separate portions of software systems into independent units, strictly controlling the amount of internal information which each unit presents to the system as a whole, in order to protect the internal operation and consistency of each unit. In general, this increases the robustness of the entire system and makes each unit more reusable. Slicing, by contrast, seeks to follow paths of information flow through a software system; in following an information flow path any distance, a slice will quickly encounter a module boundary. Whether the boundary is of a simple FORTRAN function call or of a separately compiled Ada package, the information hiding mechanisms of the module boundaries tend to frustrate the ability of a slice algorithm to follow the flow of information across those boundaries. Further, the more stringent the information hiding mechanism, the more difficult the slicing process.

This has led to the distinction between *intra*procedural and *inter*procedural slices. An *intra*procedural slice is a slice of a monolithic, single-subprogram code module. An *inter*procedural slice is a slice in the context of multiple subprograms which may include procedure and function calls and returns. No one has yet developed a slicer which can slice a full Ada system which comprises multiple packages, and so no one has yet needed to employ the terms intra-package and inter-package. Weiser considered separately compiled modules of SIMPL-D, but because of the limited interface information available in that language, he could only make worst-case assumptions about the interior of a separately-compiled module. For example, if a variable x is in the interface of a module Y (e.g., the procedure call $Y(x)$) for which the source code is unavailable, it must be assumed that x is modified in Y. This leads to the inclusion of the entirety of Y in every slice which includes x in the slicing criterion and a call to a component of Y. This inclusion is conservative and safe, in that some code which cannot affect x may be included, but no code will fail to be included which can affect x. This leads to a slice which is larger than it should be, in that it contains code which does not affect the value of the variable of interest.

3.3 Developments in Static Slicing

During the middle 1980s, few new results in slicing were reported. Rather, the advancement of slicing in the middle 1980s was largely accomplished by researchers who were not working on slicing as a primary goal at all. Their results had the side effect of improving slicing techniques by advancing the theory and practice of graph-based representation and analysis of programs.

3.3.1 The Program Dependence Graph

In 1984, Ferrante, Ottenstein, Ottenstein, and Warren [15,38], studying a form of internal program representation called the *program dependence graph* (PDG), demonstrated how the PDG could be used as the basis of a new slicing algorithm. While the algorithm was initially restricted to intraprocedural slicing, its importance was that it produced smaller slices than Weiser's algorithm. This new method differed from Weiser's in an important

way: it used a single reachability pass of a PDG (see Section 3) rather than Weiser's incremental flow analysis, and thus produced a slice in linear time. The initial construction of the PDG is a complex, time-consuming operation, varying between $O(n^2)$ and $O(n^3)$ depending upon the exact method used, where n is the number of nodes in the dataflow graph (and also the number of statements in the program), and e the number of edges. Once the PDG is built, however, every slice can be computed in time linear in the size of the sliced program. The process of building the PDG results in the computation and storage of most of the information needed to generate all the slices of the program.[16] In Weiser's approach, every slice must be computed from scratch; no information from the previous slice computation is kept; every slice generated with Weiser's algorithm requires $O(n^2e)$ time. If the purpose of a specific application of slicing is to locate a single program bug, then the effort of building a PDG may not be worthwhile. Indeed, each time a program is modified, the PDG must also be modified to reflect the new program structure. Updating the PDG of a large and complex program after each program change is not a trivial cost if the program is undergoing extensive modification. On the other hand, when the purpose of a specific slicing application is to generate many slices of an unchanging program, such as slice-based total decomposition, then the expense of building a PDG is amortized over the entire set of slices and provides an overall cost savings.

Since a PDG does not consist of statements as does source code, slicing based on a PDG cannot use statements and variables as a slicing criterion in the same way as slicing based directly on source code. Rather, slicing a PDG must use a node (or set of nodes) as the slicing criterion. Since a node represents a single program statement containing just the variables referenced in that statement, a slicing criterion based on a PDG consists of a statement and one or more of the variables in that statement. It cannot include a variable not referenced in that statement — that is, an arbitrary program variable. It must instead use a statement and variables referenced in that statement. In practice, this turns out not to be a restriction, but in fact removes the problem of the slice criterion statement being absent in the slice. In PDG-based slicing, this never occurs. The one serious flaw of PDG-based slicing is that it is intraprocedural. The PDG was not designed to capture control and data flow which crossed procedure boundaries. Weiser's algorithm did produce interprocedural slices, even though the intraprocedural portions of those slices were cruder than corresponding PDG-based slices.

3.3.2 Slicing With Relational Equations

In 1985, Bergeretti and Carré [8] were studying control- and data-flow analysis of programs by using the relational algebra paradigm. They developed a set of information-flow relations, and noted, almost parenthetically, how slices could be pulled directly from their relational tables, once those tables had been built for a given program. This approach to generating slices is similar to Ottenstein and Ottenstein's in that once the structure of the program is analyzed, any given slice can be obtained in linear time. This

16 This is not strictly true. Weiser had proven that computing the minimal slice is equivalent to the halting problem, and the new PDG reachability method of computing slices did not invalidate that proof. The new algorithm could produce all *computable* slices, but not every slice which an oracle could produce. See Section 3.1.2.

approach differs from the others considered here in that it is based on solutions to relational algebra equations rather than on graphs. The algorithm fills relational tables with information about the program in the course of solving the relational equations. Any slice can then be extracted from the tables by performing an appropriate *select* on the tables. The language which Bergeretti and Carré used, however, was a toy language, containing only scalar assignment, if-then-else, and while-do constructs. This, the slices produced by this technique were also strictly intraprocedural.

3.3.3 Interprocedural Slicing

In 1990, Horwitz, Reps, and Binkley [25] made a significant improvement in slice generation technology by producing an interprocedural slicing algorithm based on the system dependence graph which correctly accounted for each procedure's called and calling contexts. Although Weiser's original work did in fact encompass interprocedural slicing, his algorithm did not take the calling context into account, and thus his slices were often much larger than necessary. This is because Weiser's algorithm made conservative and safe worst-case assumptions on variables used as procedural parameters. When using slicing for such purposes as program comprehension or bug location, a slice's usefulness is in general inversely proportional to its size. The Horwitz, Reps and Binkley algorithm made it practical for the first time to generate useful slices which cross procedure boundaries. Interprocedural flow analysis continues to be a major topic in language system research, and new results in this area will continue to improve static slicing technology. For example, Burke and Choi [10] present a method for factoring aliases due to reference parameter passing in procedure calls into data-flow analysis. Their method is more efficient both in terms of time and space overhead of the analysis and in producing less conservative data-flow information than previous techniques. Since Burke and Choi do not address slicing, it is left to future researchers to incorporate the improved data-flow analysis technique into an actual slicer.

3.4 Dynamic Slicing

In 1988, Korel and Laski introduced a new form of slicing [27]. This new form of slicing is dependent on input data and is generated during execution-time analysis, as opposed to Weiser's static-analysis slicing and is therefore called *dynamic* slicing. Similar to the origins of static slicing, dynamic slicing was specifically designed as an aid in debugging, and can be used to help in the search for offending statements which cause program error.

The definition of a dynamic slice is similar to that of a static slice. A dynamic slice is an executable subset of the original program whose behavior is required to be identical to the behavior of the original program with respect to some subset of program variables at some statement. A dynamic slicing criterion of program P executed with input x is a triple $C = \langle x, I^q, V \rangle$, where I is the line number of the statement executed, q, if present, is the repetition count of the Ith statement, and V is a subset of P's variables.[17] Given some

17 This I^q notation is somewhat different from the usage of Korel and Laski. They used q to indicate the absolute count of *all* statements executed. Here q is used only on a statement which is repeated, and indicates the absolute number of times the Ith statement has executed since the beginning of program execution. If q is absent, it indicates the default value of 1.

input x, a program executes statements in some order until the program halts. The ordered list of statements which a program executes is called the *trajectory* of the program. Dynamic slicing is based on a program's specific trajectory given a specific input, rather than upon the space of all possible trajectories as is static slicing.

For example, consider the program in Figure 15. Given the input $x_1 = \langle 3, 4, 5, 6 \rangle$, this program has the trajectory $T_1 = \langle 5, 6, 7, 8, 6, 7, 8, 9, 6, 7, 8, 11, 12, 11, 12, 11, 12 \rangle$; for $x_2 = \langle 2, 4, 6 \rangle$, $T_2 = \langle 5, 6, 7, 8, 7, 8, 11, 12, 11, 12 \rangle$. In general, for a given input, some set of the statements in the program do not execute, other statements execute once, and still others execute more than once; the complete history of which statements execute, and in what order, is captured in the trajectory. Only the statements in the trajectory must be considered in order to find a dynamic slice. For the program in Figure 15, a dynamic slicing criterion might be $C = \langle x_1, 12^2, a[2] \rangle$, producing a dynamic slice which is identical to the original program. Any static slice on the array would also produce this same program, with no statements deleted. However, for the criterion $C = \langle x_1, 12^1, a[1] \rangle$, the corresponding dynamic slice does not contain statement 9, as statement 9 does not affect the value of the first array element given the input of x_1.

```
 1  var
 2    a: array [1..10] of integer;
 3    i, n: 1..10;
 4  begin
 5    read (n);
 6    for i := 1 to n do begin
 7      read (a[i]);
 8      if (a[i] mod 2 = 1) then
 9        a[i] := a[i] * 2;
10    end;
11    for i := 1 to n do
12      write (a[i]);
13  end.
```

Figure 15 Program example for dynamic slicing

We wish to point out that the original definition of static slicing does in fact require the consideration of input data. In the definition, a slice was required to exhibit specific behavior when it was run with specific input. The difference is that while the definition of a static slice refers to input data, the algorithm for generating the static slice has no need to refer to input. This is in contrast to dynamic slices, which do in fact need input data for their generation.

Dynamic slicing was developed to overcome several specific deficiencies in the usefulness of static slicing for locating program bugs. These deficiencies are due to the potentially large size of a static slice compared to a dynamic slice, and stem from the fact that an observed bug occurs at runtime, during actual program execution. A static slice is defined on the condition that the program to be sliced terminates on all possible inputs, even those which are unlikely to be encountered in normal use. A bug, however, is noticed on a specific input. There are several situations in which the structure of a program leads to differences between static and dynamic slices of the program.

The first shortcoming of static slicing which dynamic slicing overcomes is due to the handling of individual array elements. In all static slicing to date, an array is considered to be a single variable. Any modifying reference to any array element is considered to alter the value of the entire array. This means that if a particular array element is part

of the static slicing criterion, every reference to any of the array's elements becomes part of the slice, regardless of whether the specific element of interest is affected. In dynamic slicing, each array element is tracked as a separate data location, as shown in the example above. The main reason that arrays are handled as they are in static slicing is convenience. While some array element references can be determined statically (e.g., *a[3]* always refers explicitly to a specific element), other references cannot (e.g., *a[i]*). Rather than having two ways of handling array references, static slicing algorithms usually take the easier approach of a single mechanism. (See also Section 4.4.7, Page 38).

A second shortcoming of static slicing is that pointer variables severely complicate slicing. Consider the C program in Figure 16. A static slice corresponding to the criterion ⟨15, *i*⟩ must include every statement in the original program. This is because at line 13, the pointer variable *p* is being assigned, and it is not possible for a static analysis to determine to which variable location *p* might point during some run of the program. In a more complex program, an integer pointer could be pointing to any integer variable which is visible in the current scope, and thus an assignment to this pointer would require the inclusion of *every* statement in the current scope which includes an integer variable. Indeed, while this situation exists for a strongly typed language, for a language such as C the situation is worse. Since C allows void pointers and limitless pointer typecasting, a single pointer variable can potentially refer to every data location in the program. As Ferrante, Ottenstein, and Warren put it, "Pointers in a language such as C can preclude PDG construction altogether since they can point to anything. In the worst case, one would have to conservatively assume that all objects are aliased" [15]. Not all pointers are so badly behaved, of course. The difficulties with C's pointers are due to that language's total lack of mathematical basis for any disciplined use of pointers. At the opposite end of the spectrum, the use of pointers in a language such as RESOLVE [40], which is based very strictly on mathematical concepts, is as deterministic and verifiable as the use of any other reference mechanism [47]. Therefore, static slicing a RESOLVE module with pointers should be no more difficult than static slicing a RESOLVE module without pointers.

```
 1  #include <stdio.h>
 2
 3  void main (void)
 4  { int  i  = 0;
 5    int  j  = 0;
 6    int  *p = &i;
 7    char c;
 8
 9    c = getchar();
10    if (c == 'y')
11      p = &j;
12
13    *p = 1;
14
15    printf ("i: %i\n", i);
16    printf ("j: %i\n", j);
17  }
```

Figure 16 Pointer variable program

Even for C, though, pointers present no problem for dynamic slicing. Consider the program in Figure 16, as shown in Figure 17, with the same variable and statement of interest as before, and the same pointer variable *p*, in the context of dynamic slicing. Because a dynamic slice depends on the input to a specific execution, the value of *c* is known during generation of the slice. This allows the inclusion of line 13 in the slice to be based on whether the assignment at line 11 takes place. Because a dynamic slicing criterion is

a triple, the criterion for this situation could be ⟨*y*,15,*i*⟩ (that is, the character *y* is the actual input, the statement of interest appears on line 15, and the variable of interest is *i*). This criterion generates the slice of Figure 17, which static slicing cannot generate. The distinction between static and dynamic slices can be summed up by saying that a static slice is all statements that *could* influence the value of a variable for *any* inputs, while a dynamic slice is all statements that *did* influence the value of a variable for a *specific* input [3].

```
#include ⟨stdio.h⟩

void main (void)
{ int  i  = 0;

  printf ("i: %i\n", i);
}
```

Figure 17 Pointer variable program sliced on ⟨*y*,15,*i*⟩

Korel and Laski's approach to computing dynamic slices was based on Weiser's original incremental data flow analysis technique. But this incremental technique is 1) more expensive and 2) produces larger slices than the dependence-graph techniques of Ottenstein and Ottenstein [38] or the information-flow relations of Bergeretti and Carré [8]. However, just as static slices can be constructed using these newer reachability techniques, so also can dynamic slices. This was expounded by Agrawal and Horgan [3], who develop the techniques of dynamic slicing in steps using five increasingly sophisticated algorithms. The second of those algorithms computes exactly the same slices as Korel and Laski's technique, although less expensively.[18] The Agrawal and Horgan dynamic slice technology, as well as Horwitz, Reps, and Binkley static slices, were incorporated into a prototype debugging tool by Agrawal, DeMillo, and Spafford [1]. In 1991, they advanced dynamic slicing by developing methods to handle pointers and composite data structures such as arrays, records, and unions, even in the case of unconstrained pointers without runtime checks, such as are found in C [2].

3.5 Semantics and Mathematical Models

In overview, conventional program slicing consists of the intuitively simple process of deleting selected statements from a syntactically correct program to produce a new, smaller, syntactically correct program slice, with certain requirements and constraints placed on the relationship between the original program and the slice.

18 Agrawal and Horgan use a slightly different definition of a dynamic slice than Korel and Laski. This allows them to produce smaller dynamic slices in their final algorithm. In addition, their approach is less expensive, even for Korel and Laski slices.

In general, the overall process of slicing as described in the current literature may be presented as:[19]

$$\underbrace{\textit{Original Code} \rightarrow \textit{RepresentationA}}_{\textit{Transformation A}} \rightarrow \underbrace{\textit{RepresentationB}}_{\textit{Slicing Algorithm}} \rightarrow \underbrace{\textit{New Code}}_{\textit{Transformation B}}$$

What assurance is there that the *New Code* which results from this process bears the relation to the *Original Code* which the definition of slicing requires? There is no absolute guarantee because all questions of program equivalence eventually reach the borders of undecidability, but various researchers have directed their efforts towards defining the process as precisely as possible and making the process as correct as possible. A full treatment of general program equivalence is far beyond the scope of this paper, but a few comments are in order.

To completely prove that the process above is correct would require that the three steps above, the two *Transformations* and the *Slicing Algorithm*, each be separately proved correct, and then that their composition also be proved correct. Furthermore, since each researcher presents a slightly different variation of each of the steps, each researcher's results in fact require a unique proof. In general, proofs of slicing algorithms in this sense are not provided in the literature. Early reports of slicing in the literature made no attempt to prove semantic results of slicing algorithms. In the late 1980s, researchers began to consider the question of program semantics in intermediate representation and slicing work. In 1988, for example, Hausler provided an alternate definition of slicing based not on flow analysis at all but rather on the functional semantics of a specific programming language. Hausler associated a denotational slicing definition with each statement type in the program's grammar, thus allowing a systematic mathematical basis for constructing slices [20]. Unfortunately, his results were based on a very simple language, and it is unclear how they would be expanded to include any form of procedure call or non-structured construct.

In the same year, Horwitz, Prins, and Reps reported that although the PDG representation of a program had been first proposed as early as 1972, no proof existed that two unequivalent programs in source code form would map to unequivalent PDGs, and thus there was no a priori reason to accept a PDG as an appropriate program representation mechanism [24]. They proceeded to prove that in a certain defined sense, the PDG is indeed an "adequate" representation mechanism, based on the concept of *strong equivalence* between programs. Two programs are strongly equivalent iff given some initial state and identical input, either both programs diverge[20] or both halt with the same final state of all variables. They proved that if the PDGs of two programs are isomorphic

---

19 *Original Code* and *New Code* can themselves be considered to be representations of abstract programs, and so the upper line of the process could equivalently be written:
Representation1→Representation2→Representation3→Representation4.

20 Here divergence includes infinite loops and an abnormal halt due to an unrecoverable error such as divide by zero or overflow.

then the programs are strongly equivalent. However, their result applies only to a simple, structured, scalar, intraprocedural language, and they did not consider slicing transformations in their consideration. Reps and Yang extended this result to slicing by proving two results. First, they showed that if both the original program and the slice halt on the same input, then their slicing algorithm guarantees that the slice and the program will produce the same sequence of values for each variable in the slice. Second, they showed that if their slicing algorithm is used to decompose a program into two or more slices, then the program will halt on any state for which all the slices halt [45]. Again, these results apply only to a structured language. For their slicing algorithm, Reps and Yang modify the Weiser's original definition, allowing a broader interpretation of what is an accurate slice of a program by removing the previous restriction that the order of statements in the slice must be identical to their order in the original program. Determining that a slice is correct is based on isomorphism of the PDGs of program and slice instead. Thus Weiser's original definition of a slice being produced strictly by removing statements from a program is not accurate for Reps-Yang slices.

Again studying a simple intraprocedural *while*-language, Lakhotia [29] provides a graph-theoretic framework for helping to factor the issues of program semantics away from the mechanics of slicing itself. The original definition of slicing applied to statements and variables, thus explicitly grouping program semantics with slicing, while later slicing techniques were defined in terms of graph operations which included program semantic definitions of the individual operations. Lakhotia 1) explicitly translated Weiser slicing into graph operations and 2) isolated the program semantics from the graph operations of all the slicing techniques. This allows a more direct comparison of the different versions of slicing, and also makes more explicit which portions of slicing are concerned with program semantics, and which with mechanical graph operations. Lakhotia, in fact, was trying to eliminate all concerns with program semantics from the mechanics of slicing, to reduce slicing itself to strictly graph operations. In this view, the issues of program semantics are isolated in the steps of creating the graph representation of the program before slicing, and regenerating a program from the sliced graph representation after slicing.

Podgurski and Clarke [41] find the traditional definitions of control and data dependence insufficient to deal with the semantics of a program being represented by a flow graph, and introduce the notions of weak and strong syntactic dependence and semantic dependence. These new types of dependence are refinements of previous definitions of types of dependences in an effort to more precisely state the dependence relationships within a program. As the authors reiterate, the general question of when an arbitrary change in a program's syntax will alter the program's semantics is undecidable. Efforts such as these, however, continue to expand the envelope of situations in which the program semantics due to program changes can be understood, and thus the change mechanisms safely automated.

In summary, intraprocedural control and data flow dependences for simple, structured, scalar languages have been well enough studied, and their semantics well enough formulated, that provable assertions about the semantics of slices of terminating programs can be stated fairly confidently for several of the main slicing algorithm variants. In contrast, interprocedural flow analysis is still an active area of research, not

well enough understood for substantial proofs about the semantics of interprocedural slices. Similarly, non-structured languages and languages with late binding or real-time constraints have not even been considered for slicing, much less for formal semantic proofs based on their slices.

4 Applications of Conventional Slicing

This section discusses the uses to which slicing has been applied or for which it has been proposed. This list is intended to be comprehensive to date, but it is a certainty that new applications will be found for slicing in the future. At the end of the section, we discuss several possibilities for future applications.

Weiser built simple prototype FORTRAN and SIMPL-D slicers, but he and others envisioned a robust, possibly integrated [38], language-independent slicing tool, which accepted an intermediate program representation from one of a suite of language-specific front ends. In his thesis, Weiser described four areas to which static slicing could be applied. They were program maintenance, debugging, requirements verification, and operating system kernel tailoring. Later, he added the ideas of program testing [54] and of parallelization by assigning slices to separate processors for execution [51].

4.1 Maintenance

Most of the applications of slicing fall into the maintenance portion of software engineering. This is partly because slicing is defined only on existing code, and partly because of the nature of the interaction between slicing and programs. The suitability of slicing for maintenance applications is due to the ability of slicing to abstract or respond to certain patterns of knowledge or design information which is built into programs. These patterns have been variously described as *plans, clichés,* and *concepts* (see Section 4.3 below). To accomplish the activities of software maintenance, it is necessary for the maintainer to be able to recognize and sometimes isolate these patterns in programs. If the maintainer is intimately familiar with the program, then this knowledge may already be in hand. Sometimes, good documentation will supply the information. But often, this information is not readily available, and must be extracted from the code itself before maintenance modifications can safely commence. Unfortunately for the recovery of this kind of knowledge, programmers are trained in the techniques of traditional hierarchical, sequential modularization, and these techniques are heavily used in writing programs.[21] Since modularization frustrates the easy extraction of non-sequential programming patterns, slicing tools have been employed, either to aid the maintainer in the recovery effort, or by automation to somewhat obviate the need for the maintainer to directly handle the information. With the understanding that this common theme of extracting and managing non-sequential programming patterns or concepts runs through all maintenance applications, we will now turn to a discussion of specific slicing applications.

---

21 We are not arguing that modularization is wrong or bad. Rather, our claim is that it is only one view of design knowledge, and that as use of modularization tends to obscure other views, its exclusive use is equivalent to working in 2-D when 3-D is available.

While we discuss each application separately, it is clear that the greatest benefit from each application will appear when each of these applications is incorporated into an integrated software maintenance environment [31], or better yet, is incorporated into an integrated software development environment.

4.1.1 Modification and Reverse Engineering

Gallagher and Lyle [18,33] developed an application for slicing by extending the definition of slicing. They used current slicing techniques to create a *decomposition* slice. In contrast to a Weiser slice which uses a slicing criterion which names a statement or set of statements at which the behavior of a variable of interest will be examined, a decomposition slice is a slice on a variable which does not depend on a set of statements, but includes all statements relevant to the variable of interest in the entire program. They also introduced the notion of a slice *complement* which can be thought of as the portion of the original program left when the decomposition slice is removed, plus the statements required for syntactic correctness. The complement is also a slice.

Using this concept of a decomposition slice, Gallagher and Lyle decompose a program by generating a decomposition slice on every program variable, and then arranging these decomposition slices into a poset based on the partial ordering of set membership of program statements. Their technique produces a total program decomposition, and thus every statement is in at least one slice. The relationships among program statements imposed by the poset structure allow them to assert four rules for modifying the program which guarantee that 1) the modifications will only have an affect within the slice and not in the complement (which is the rest of the program), and that 2) retesting the program after modification need only be concerned with the statements within the slice and not those of the complement. Thus only the slice need be retested, not the entire program.

Gallagher and Lyle argue that using the decomposition poset with their rules of allowed modifications in effect presents the maintainer with a semantically constrained context. In traditional modification techniques, the maintainer is free to make any changes but then must exhaustively test and analyze the results to ensure that no undesirable effects have been inadvertently created. In contrast, Gallagher and Lyle's method prevents the maintainer from making inadvertent errors in the first place with the aid of an automated tool, thus greatly relieving the burden of post-testing on the maintainer. Presumably, the added effort of formulating a desired modification within the constraints of the poset-induced context is much less than the amount of post-testing which is obviated, resulting in a considerable net savings of maintainer effort.

Carrying this idea a step farther, from the realm of a single standalone program modification into the realm of integrated software maintenance, the poset structure also facilitates the redocumentation which is critical to the integrity of a long-lived system or component. As incremental modifications are performed on a component, the poset of slices will differ at particular vertices from the original poset. An examination of the differences specifically identifies those portions of the current component which are in need of redocumentation [7].

4.1.2 Debugging

Debugging an existing program was the original impetus for the development of static slicing, and it is still the application which automatically springs to mind when program slicing is mentioned. In his thesis, Weiser provided experimental evidence that programmers unconsciously use a mental form of slicing when engaged in program debugging, as well as empirical evidence that slicing did indeed aid programmers in bug location. To date, the only application for dynamic slicing which has been advanced is bug location of a working program. By definition, dynamic slicing depends upon input data, and so a dynamic slice is to some extent ephemeral, changing each time the program is run with different data. Optimizations, transformations, any form of generic characterization, are only valid if they can be stated for all possible input values; dynamic slicing cannot be used for these.

4.1.3 Integration of Program Variants

Horwitz, Prins, and Reps [23] developed the concept of using slicing to automatically integrate two versions of a base program which had been developed separately, in parallel. Consider the situation in which a stable, correct, base version of a program is given to two teams of developers, each of which is charged with enhancing or perfecting a distinct portion of the program. Starting with the original program *Base*, team *a* develops program variant *A* and team *b* develops program variant *B*, each by modifying their respective portion of the program. At the end of the development, the two variants must be merged into a single new program while ensuring that 1) the new functionalities of the modifications are incorporated into the result, and 2) the modifications do not conflict, or "interfere". Horwitz, Prins, and Reps formally defined this notion of interference, and developed an algorithm which accepts as input three programs *Base*, *A*, and *B*, where *A* and *B* are variants of *Base*, and produces as output either a new program *C* which incorporates all the behaviors of *A* and *B* or the determination that the two variants interfere. Slicing is central to the algorithm, in that it is used to determine the new behaviors of *A* and *B*, and to determine whether their behaviors interfere. The specific algorithm they developed, however, works only for a small structured language which includes assignment, conditional, and while statements with no subprograms, pointers, or non-scalar variables.

4.1.4 Parallelization

An application of slicing which was discussed at length in the early 1980s (e.g., [53]) is its use to decompose a conventional program into substantially independent slices for assignment to separate processors as a way to parallelize the program. Interest in this application for slicing has recently revived, and it is a current research area of considerable activity [badger & weiser]. A program for which slices with small overlap can be found would be an appropriate subject for this type of parallelization. Assuming that a combination slicer-compiler could produce a sliced executable suitable for a parallel machine, an issue of some complexity is the problem of reconstructing the original behavior by "splicing" the results of the separate outputs of the slices. A study of this is given in [53].

4.1.5 Software Portability

A very recent application which has been proposed for slicing is slicing to aid software portability [34]. Consider a software system which was written with a specific platform as

its target. Scattered throughout the system might be operating system calls or data structures which assume a specific numerical format. Porting the system to a new platform requires finding and modifying all portions of the system which depend on the original platform. This familiar task is usually frustrating due to the ease with which bugs can be introduced due to hidden linkages and dependences in the system.

For example, consider the case of a program which performs file I/O by declaring a file block data structure variable and repeatedly passing the address of this variable to various operating system I/O procedures. Locating and isolating every instance of file I/O in this program consists merely of using the file block data structure variable as the slicing criterion. Then to port this software to a different platform which uses a different I/O mechanism, the modification techniques of Gallagher and Lyle mentioned above will guarantee that the modifications are made safely without introducing bugs into the balance of the program.

4.2 Non-Maintenance Software Engineering

4.2.1 Requirements Verification

Consider a compiler designed with two main functionalities, producing object code and producing a cross-reference listing; execution-time switches control whether each functionality is enabled. Also assume that the compiler was developed to meet a specific set of stated requirements. In this situation, the complete compiler can be considered to be an amalgam of two separate quasi-independent sub-programs, a code generator and a cross-reference generator. Clearly the program will be written with the two main functionalities of code generation and cross-reference generation intermixed in its source code. Some functionalities in the program, such as lexical analysis, will be shared by both generators, while others will be specific to the code generator or to the cross-referencer. Therefore, the portions of the program which meet the requirements of code generation are not all grouped together, but are spread throughout the program, and similarly for the requirements of the cross-reference generator. Identifying those portions of the program which satisfy the requirements for the code generator would be a difficult task. In a large system, there would be hundreds or thousands of requirements, and evaluating the conformance to each is a major task in software development. A slicer, however, being able to project out of the large system a single behavior of interest, can produce a slice which contains only, say, the code generator, allowing an easier evaluation of how well the code generator conforms to its requirements. Weiser described this sort of requirements analysis in his paper [50], but to our knowledge it has never been pursued.

4.2.2 Module Cohesion

In 1989, Ott and Thuss [36] used slice profiles as a means of helping to quantify the somewhat subjective determination of module cohesion in a program. High module cohesion has been proposed [11] as a desirable property of software design, but its standard definition is subjective, making it difficult to use in practice. Ott and Thuss showed that the degree of cohesiveness of a module can be determined by examining slices generated using the output variables of the module in slicing criteria. Their work strictly uses intraprocedural Weiser slices.

4.2.3 Dead Code Elimination

In the interest of making the list of slicing applications complete, we mention a slicing-based technique for dead code elimination which was described by Gallagher [17]. As described in Section 4.1.1, a program can be totally decomposed into slices by creating a decomposition slice for every variable in the program. If the original program contained no dead code, then the union of all the decomposition slices is the original program. But if the original program P contained dead code, then P - (union of slices) = dead code. While this is a method for dead code recognition, even Gallagher admits that it is probably not an efficient one, and in fact its effectiveness is dependent on the effectiveness of the slice algorithm underlying the decomposition.

4.3 Related Work

The notion of organizing a program in ways other than the traditional hierarchy of units of increasing abstraction is not unique to slicing. Soloway and Ehrlich [49] developed the theory that programming knowledge consists in part of *programming plans*. A programming plan is an abstract structure which a programmer uses as a template or link between a goal and a program fragment instance. A programmer might use, for example, a *data guard plan* to help accomplish the goal of preventing division by zero. In the program, the data guard plan is manifested in the test predicate and control structure necessary to prevent division by zero, while allowing division by appropriate values. While a plan may be a single abstract entity, it is manifested in a program by statements which are, in general, non-sequential. Furthermore, in many cases, an appropriate choice of slicing criterion applied to a program segment is sufficient to recover an intact plan as a slice.

Rich and Wills [46] have developed a prototype module of the Programmer's Apprentice called the *Recognizer* which automatically recognizes *clichés*, which essentially are the manifestation of plans in programs. Similarly to slicers, the Recognizer stores program information in the form of a flow graph; the clichés are found by graph analysis.

In a very similar vein, Kozaczynski, et al., use a combination of syntax patterns and data flow analysis to identify *program concepts* in COBOL code [28]. Their emphasis is on abstract concepts which represent language-independent ideas of computation and problem solving methods. Their purpose is to be able to apply specific types of maintenance-related transformations to programs automatically by recognizing and transforming a concept pattern, with the burden of code comprehension shifted from the maintainer to the concept recognizer.

4.4 Future Work in Conventional Slicing

4.4.1 Slicing a Real Language

To date, the vast majority of slicing algorithms, whether manual or implemented, apply only to very restricted, or toy, languages. Often these languages require additional non-standard extensions, such as variable lists in the end statement, to enable the slicing algorithms to work. we have already mentioned that much slicing has been performed on languages without procedure calls. Weiser did most of his work using FORTRAN IV, which lacks recursion, pointers, nested procedures, case statements, and complete

(nestable) if-then-else constructs. In 1988, Horwitz, Prins, and Reps [23] were still reporting major results based on a tiny language having only scalar variables and constants, assignment and conditional statements, and while loops without breaks. In late 1991, Livadas, Croll, and Roy [30,31], reported on the *p*- and *c*-Ghinsu toolsets which respectively slice subsets of standard Pascal and ANSI C, not including pointer variable declaration, aliasing, recursion, or I/O. To date, however, no one has written a slicer for the full grammar of a mainstream language. In addition, no one has dealt in the literature with the ramifications of multiple exits in either procedures or loops; slicing tasks was mentioned in the Future Research section of Lyle's thesis [32], but there was no indication of how this could be accomplished; slicing in the presence of late binding or real-time constraints has not even been mentioned in the literature.

Extending current static slicing to handle multiple-exit procedures or loops may need only persistence, perhaps messy but straightforward. The two areas of tasking and late binding, however, are likely to involve a great deal of effort and to produce considerable results in program analysis. For example, how would dynamically created tasks be represented in a system dependence graph? What would be involved in slicing generic tasks? Do some tasking synchronization mechanisms make slicing particularly difficult, or fruitful; do some allow task slicing to be reduced to the simplicity of subroutine slicing? How does late binding affect the process of static slicing?

4.4.2 Slicing Large Programs

Weiser [50] reported slicing a FORTRAN program of 380 lines. Basili and Reiter [4] reported slicing FORTRAN programs, the largest of which was 900 executable statements, and gave some results on the complexity and time requirements of the slicer used. Much more recently, Ottenstein and Ellcey [37] reported generating PDGs for FORTRAN programs (not actually slicing them, but just building the PDGs), the largest of which was 1144 statements, and the largest module of which was 230 statements. What happens when large programs, e.g., programs consisting of 10^6 SLOC, are sliced? Do the slices change in character, or are there simply more of the same type? These questions cannot be answered in general until working slicers are built for real languages.

4.4.3 Slicing a Modern Language

One of the languages which Weiser studied [51] was SIMPL-D. This language allows separate compilation of modules, as do many modern languages. Weiser noted the difficulties that this caused for slicing, and handled the problem by falling back to a conservative and safe worst-case assumption in which a called procedure in a separately compiled module will change the value of every global variable, and every local variable bound to the formal parameters in the call. This worst-case assumption was necessary, as not enough information is available in a static analysis of SIMPL-D to improve upon it.

Ada, however, provides several mechanisms (such as the in/out mode of parameters and the library management features of an Ada environment) which allow much better analysis than the worst-case assumption. In addition, Ada has many features which languages such as FORTRAN and Pascal, the basis of most slicing research, do not have. Tasking, generics, exceptions, nestable declaration blocks, overloading, and separate compilation are just some of the features of Ada which present novel aspects for would-be slicers. No one has studied these mechanisms for the purpose of slicing, or examined how

they enable or hobble slicing. Conversely, no one has studied how slicing may affect any of these features. Ada, of course, is not unique in having these features; we are merely using it as an accessible example of modern language features which slicers have yet to address.

Other, more recent languages have features not found in Ada. Even without moving beyond the realm of strictly traditional procedural 3GL languages, there are features such as incremental compilation [16] which surely would be influenced by slicing. Considering newer languages which contain assertions and invariants, might not a compiler which includes a slicer be able to use these constraints upon which to base a code-reduction slice, much like a dynamic slicer uses the input data constraints to limit the slice size? Again, such ideas have yet to appear in the literature. Another untouched language topic is slicing a non-procedural language.

4.4.4 A Bestiary of Slices

Weiser [54] provided several metrics for static slices, including *coverage, overlap, clustering, parallelism,* and *tightness.* However, these metrics were given separately, and it is not clear whether they sufficiently orthogonal to establish a taxonomy. If not, the question is immediately raised as to what other form of metric is needed to cover a taxonomy. Going beyond static slices, there is a question of extending the taxonomy sufficiently to include dynamic slices as well. Dynamic slices are of the same sort as static slices, in that they are sets of program statements. Clearly dynamic slices can exist in a poset of static decomposition slices, but this is merely an arrangement of instances, not a taxonomy of mechanisms themselves.

Extending the logic of the previous paragraph, do slices, in fact, provide a way of characterizing programs? In 1988, Horwitz, Prins, and Reps [24] discussed equivalence and isomorphism in PDGs and their associated programs. Since a slice projects isolated behaviors from a program, is it possible to abstract a program's behaviors into its slices and thus characterize the program? Do the slices thus provide a "behavior signature" of the program? If behavior is too large a concept for slices to adequately encompass, do slices still provide a program signature of some sort which would be of either practical or theoretical interest in characterizing, classifying, describing, or retrieving the sliced artifacts? It is possible that most programs, or at least most programs in a narrowly defined domain, each contain a subset of a set of "basic" or "standard" slices, up to isomorphism and renaming, with the program's unique features isolated in a few unique slices. If this is the case, then slicing can provide an automatic characterization of each program, with the standard slices indicating the program's inclusion of standard domain-specific features, and the unique slices indicating the program's unique features.

4.4.5 Generating Repository Components

At the end of their 1991 paper [18], in the Future Directions section, Gallagher and Lyle mention evaluating slices as candidates for inclusion in a software repository. Of course, actually building useful repository components by slicing will have to await the development of a tool which can slice a real language. But assuming that a real slicer is available, the question, "What is involved in creating and reusing a reusable component by slicing?" is decidedly non-trivial. For example, it seems that a reasonable requirement for decomposition by slicing would be reversibility. Given a program with three

functionalities (such as the unix *wc* utility described by Gallagher and Lyle), and assuming one has sliced the three functionalities into three separate reusable components, how does one go about reconstructing the functionality of the original program from the three components? What kind of interface does each component have? If they interact in the original program, is their interaction preserved after decomposition and reassembly? Does the preservation depend upon the nature of the original interaction? All of the slicing criteria described to date [29] consist of either a statement set alone, with no variable set specified, or upon both a statement set and a variable set. However, for generating repository modules, completely decomposing a module into the largest number of slices, without regard to statement number at all, may be the most logical generation technique. But "largest number" varies according to slicing methodology. Should the slices be independent, or can overlapping, dependent slices be useful in a repository? In the literature, no one has yet considered a methodology for generating reusable components by slicing.

4.4.6 Static Slices via Dynamic Slicing

There is a gulf between the camps of dynamic and static slicing researchers. In much of the literature on dynamic slicing, and in some on static slicing (e.g. [31]), proponents of one camp refer to the other merely by pointing out the advantages of their version of slicing for a particular application, but no one has fully investigated the relationship between them. As discussed in Section 3, Gallagher and Lyle [18] discuss arranging static slices in a poset. Since dynamic slices are simply subsets of program statements just like static slices, dynamic slices must have a place in the poset of static slices. More to the point, it is obvious that a dynamic slice must be a subset of some static slice, although to our knowledge this has not been proven or even stated before. In general, the relationship between static and dynamic slices in the poset has not been considered in the literature.

Because dynamic slicing does not involve the complex and expensive static program analysis which characterizes static slicing, it is easier to perform than static slicing. Several of the most difficult situations for static slicing, e.g., interprocedural slicing, pointers, or large arrays, present no particular problems for dynamic slicing. Therefore it seems that a potential exists for a new application for dynamic slicing, the possibility of approximating a static slice by taking a "limit," or some form of sequence or aggregation, of dynamic slices. This technique might be useful in those cases in which a static slice is particularly difficult to generate, possibly because of extensive use of pointers, a very complex data flow analysis, or a highly unstructured program.

Similarly, finding the statement-minimal static slice of a given slicing criterion, while desirable, is undecidable. If a technique for approximating static slices by dynamic slices were developed, then perhaps a minimal static slice could be approximated, thus overcoming some of the problems with undecidability, or at least expanding the envelope of those static slices which are computable. Even if no absolute technique for generating static from dynamic slices is possible, the investigation might provide heuristics for better approximating algorithms for static slices.

4.4.7 Handling Individual Array Elements

Another way of pushing the envelope of static slices is to better handle array references. Recall that this was one of the reasons cited for the development of dynamic slicing. One possible approach for refining array element reference in static slicing has been developed by Mullin [35], studying the shapes of array operators in general and the indexing operations and partitioning of arrays in particular. She has developed a mathematical formalism for partitioning arrays based on the ψ function, a generalized array indexing function. This function describes a class of array operations which Mullin used to build a group of theorems describing common situations which involve arrays in algorithms and programs. One such situation which Mullin models this way is concurrency of sub-array operations. While Mullin does not explicitly discuss slicing, the similarity of partitioned sub-array operations for slice independence and concurrent sub-array operations for parallelism (which she does discuss) gives a strong indication that her techniques would be applicable to finer static analysis of array references for slicing.

A feature of Ada which points to another way of refining array element references is the *array slice* mechanism. This mechanism is somewhat of a combination of subtyping and subranging operations performed on-the-fly, and could be used as a mechanism by the programmer to make the program more sliceable. A language which implemented a more strict and formal mechanism of subtyping of array element references would facilitate precise slicing of arrays without conscious effort on the part of the programmer.

Bibliography

[1] H. Agrawal, R. DeMillo, and E. Spafford. "Efficient debugging with slicing and backtracking," Technical Report SERC-TR-80-P, Software Engineering Research Center, Purdue University, West Layfayette, Indiana, October 1990.

[2] H. Agrawal, R. DeMillo, and E. Spafford. "Dynamic slicing in the presence of unconstrained pointers," Technical Report SERC-TR-93-P, Software Engineering Research Center, Purdue University, West Layfayette, Indiana, July 1991.

[3] H. Agrawal and J. Horgan. "Dynamic program slicing," Technical Report SERC-TR-56-P, Software Engineering Research Center, Purdue University, West Layfayette, Indiana, December 1989.

[4] V. Basili and R. Reiter, "An investigation of human factors in software development," *Computing*, vol 12, pp 21-38, December 1979.

[5] C. Beck. "Undergraduate nuclear engineering curriculum at North Carolina State College," *Nucleonics*, vol 8, no 1, pp 54-59, January 1951.

[6] C. Beck. "Training for careers in physics in scientific research," an address to the American Association for the Advancement of Science, Atlanta, 29 December 1955.

[7] J. Beck and D. Eichmann. "Program and interface slicing for reverse engineering," *ICSE-15: Proceedings of the 15th International Conference on Software Engineering*, (Baltimore, 17-21 May, 1993).

[8] J. Bergeretti and B. Carré. "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol 7, no 1, pp 37-61, January 1985.

[9] B. Boehm. *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[10] M. Burke and J. Choi. "Precise and efficient integration of interprocedural alias information into data-flow analysis," *ACM Letters on Programming Languages and Systems*, vol 1, no 1, pp 14-21, March 1992.

[11] L. Constantine and E. Yourdon. *Structured design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

[12] N. Deo. *Graph Theory With Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[13] Department of Defense. "DoD Software Reuse Vision and Strategy," *CrossTalk*, no 37, pp 2-8, October 1992.

[14] R. D'Ippolito, et al. "Putting the *engineering* into software engineering," in C. Sledge (ed.), *Software Engineering Education* (SEI Conference, 5-7 October, San Diego), pp 287-289, 1992.

[15] J. Ferrante, K. Ottenstein, and J. Warren. "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol 9, no 3, pp 319-349, July 1987.

[16] P. Fritzson. "A systematic approach to advanced debugging through incremental compilation," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging,* (Pacific Grove, CA, 20-23 March), 1983.

[17] K. Gallagher. "Using program slicing in software maintenance," Ph.D. Dissertation, University of Maryland Baltimore County, 1990.

[18] K. Gallagher and J. Lyle. "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol 17, no 8, pp 751-761, August 1991.

[19] A. Hall. "Is software engineering?" in C. Sledge (ed.), *Software Engineering Education* (SEI Conference, 5-7 October, San Diego), pp 5-7, 1992.

[20] P. Hausler. "Denotational program slicing," in *Proceedings of the 22nd Hawaii International Conference on System Sciences*, vol 2, pp 486-494, January 1989.

[21] M. Hecht. *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, 1977.

[22] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[23] S. Horwitz, J. Prins, and T. Reps. "Integrating non-interfering versions of programs," in *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, J. Ferrante and P. Mager, eds, pp 133-145, (San Diego, 13-15 January 1988), ACM Press, New York, 1988.

[24] S. Horwitz, J. Prins, and T. Reps. "On the adequacy of program dependence graphs for representing programs," in *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, J. Ferrante and P. Mager, eds, pp 146-157, (San Diego, 13-15 January 1988), ACM Press, New York, 1988.

[25] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol 12, no 1, pp 26-60, January 1990.

[26] K. Kang, S. Cohen, R. Holibaugh, J. Perry, and A. Peterson. *A reuse-based software development methodology*, Technical Report CMU/SEI-29-SR-4, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, January, 1992.

[27] B. Korel and J. Laski. "Dynamic program slicing," *Information Processing Letters*, vol 29, no 3, pp 155-163, October 1988.

[28] W. Kozaczynski, J. Ning, and A. Engberts. "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol 18, no 12, December 1992.

[29] A. Lakhotia. "Graph theoretic foundations of program slicing and integration," *Technical Report CACS-TR-91-5-5*, Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA, December 2, 1991.

[30] P. Livadas. "The c-ghinsu tool," Technical Report SERC-TR-49-F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, March 1991.

[31] P. Livadas, S. Croll, and P. Roy. "Towards an integrated software maintenance environment," in *Proceedings of the First Software Engineering Research Forum*, (Tampa, Florida, 7-9 November), 1991.

[32] J. Lyle. "Evaluating variations on program slicing for debugging," Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, 1984.

[33] J. Lyle and K. Gallagher. "A program decomposition scheme with applications to software modification and testing," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, vol 2, pp 479-485, January 1989.

[34] J. Mooney, M. Sitaraman and D. Eichmann. Personal communication, 25 September 1992.

[35] L. Mullin. "A mathematics of arrays," Ph.D. Dissertation, Syracuse University, Syracuse, New York, 1988.

[36] L. Ott and J. Thuss. "The relationship between slices and module cohesion," *Proceedings of the 11th International Conference on Software Engineering*, pp 198-204, May 1989.

[37] K. Ottenstein and S. Ellcey. "Experience compiling FORTRAN to program dependence graphs," *Software - Practice and Experience*, vol 22, no 1, pp 41-62, January 1992.

[38] K. Ottenstein and L. Ottenstein. "The program dependence graph in a software development environment," *ACM SIGPLAN Notices*, vol 19, no 5, pp 177-184, May 1984.

[39] D. Parnas. "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol 15, no 12, pp 1053-1058, December 1972.

[40] T. Pittel. "Pointers in RESOLVE: specification and implementation," M.S. Thesis, Department of Computer and Information Sciences, Ohio State University, Columbus, OH, 1990.

[41] A. Podgurski and L. Clarke. "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol 16, no 9, pp 965-979, September, 1990.

[42] R. Pressman. *Software Engineering: A Practitioner's Approach*, 2nd ed., McGraw-Hill, New York, 1987.

[43] J. Purtilo and J. Atlee. "Module reuse by interface adaptation," *Software - Practice and Experience*, vol 21, no 6, pp 539-556, June 1991.

[44] R. Prieto-Diaz. "Making software reuse work: an implementation model," *Proceedings of the First International Workshop on Software Reusability*, (5-6 July), 1991.

[45] T. Reps and W. Yang. "The semantics of program slicing," Technical Report #777, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin, June 1988.

[46] C. Rich and L. Wills. "Recognizing a program's design: a graph-parsing approach," *IEEE Software*, vol 7, no 1, pp 82-89, January, 1990.

[47] M. Sitaraman. Personal communication, 1993.

[48] C. Sledge (ed.). *Software Engineering Education* (SEI Conference, 5-7 October, San Diego), 1992.

[49] E. Soloway and K. Ehrlich. "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol SE-10, no 5, pp 595-609, September, 1984.

[50] M. Weiser. "Program slicing: formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. Dissertation, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, Michigan, 1979.

[51] M. Weiser. "Program slicing," *Proceedings of the Fifth International Conference on Software Engineering*, pp 439-449, May 1981.

[52] M. Weiser. "Programmers use slices when debugging," *Communications of the ACM*, vol 25, no 7, pp 446-452, July 1982.

[53] M. Weiser. "Reconstructing sequential behavior from parallel behavior projections," *Information Processing Letters*, vol 17, no 10, pp 129-135, October 1983.

[54] M. Weiser. "Program slicing," *IEEE Transactions on Software Engineering*, vol SE-10, no 4, pp 352-357, July 1984.

[55] M. Wolfe. *Flow Graph Anomalies: What's in a Loop?* Technical Report CS/E 92-012, Oregon Graduate Institute, Department of Computer Science and Engineering, Beaverton, OR, February 1991.

[56] E. Yourdon. *Decline and Fall of the American Programmer*, Prentice-Hall, Englewood Cliffs, NJ, 1992.