Dynamic Slicing in the Presence of Unconstrained Pointers*

Technical Report SERC-TR-93-P

Hiralal Agrawal
Richard A. DeMillo
Eugene H. Spafford

Software Engineering Research Center
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907–2004
debug@cs.purdue.edu

Abstract

Program slices are useful in debugging. Most work on program slicing to date has concentrated on finding slices of programs involving only scalar variables. Pointers and composite variables do not lend themselves well to static analysis, especially when the language involved is not strongly-typed. When debugging a program, however, we are interested in analyzing the program behavior for testcases that reveal a fault. In this paper, we present a uniform approach to handling pointers and composite variables such as arrays, records, and unions for the purpose of obtaining *dynamic* program slices. The dynamic approach proposed works well even when the language involved allows unconstrained pointers and performs no runtime checks, as in C.

1 Introduction

The notion of program slicing has been discussed extensively in the literature [20, 17, 12, 13, 5]. This discussion, however, has mostly dealt with finding slices for programs involving scalar variables (see Section 6, Related Work). Slicing would be even more useful in debugging programs that use complex data-structures involving pointers—when interstatement dependencies are hard to visualize by manual examination of the source code. Scalar variables are relatively easy to handle because the memory location that corresponds to a scalar variable is fixed and known at compile time; it does not vary during the course of program execution. Hence, if one statement modifies a scalar variable and another statement references a scalar variable, it is easy to determine, at compile time, if the latter statement references the memory location modified by the former.

The chief difficulty in dealing with an indirect reference through a pointer or an array element reference[1] is that the memory location referenced by such an expression cannot, in general, be determined at compiled time. Further, when such a reference occurs inside a loop, the memory

*This paper appeared as [3].
This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD–8913133), and by National Science Foundation Grant CCR–8910306.
[1]Not regarding an array as a single unit.

location referenced may vary from one loop iteration to another. The difficulty is compounded if the language used is not strongly-typed and permits integer arithmetic over pointer variables. Techniques proposed in [8, 10, 14] may be used to obtain conservative approximations of what a pointer might point to, but in the presence of unconstrained pointers, as in C, such analysis has only limited usefulness. In this case we are forced to make the most conservative assumption: an indirect assignment through a pointer can potentially define *any* variable. The outcome of this assumption is that static slices of programs involving pointers tend to be very large; in many instances they include the whole programs themselves.

Fortunately, while debugging a program we normally have a testcase that reveals the fault and we wish to analyze the program behavior for that testcase, not for *any* testcase. Dynamic slices provide precisely this facility. It is possible to perform precise dynamic interstatement dependence analysis even when the language is not strongly-typed.

In this paper, we present an approach to obtain dynamic program slices when the language permits unconstrained pointers. Besides pointers, composite variables such as arrays, records, and unions are also handled uniformly under this approach. It also allows precise interprocedural dynamic slicing to be performed. We first present a general framework to obtain static slices in the presence of pointers and composite variables, and then extend it to the dynamic case. While the static slicing algorithm assumes that an indirect assignment may potentially modify any variable, the dynamic slicing algorithm detects exact dependencies.

We have built a prototype debugging tool, named *Spyder*, that uses the approach described here to find both static and dynamic program slices [2] for programs written in C. The tool supports a powerful debugging paradigm involving dynamic slicing [5] and execution backtracking [4] with the help of which program bugs may be localized quickly.

To see how dynamic slices differ from static slices, consider the program in the main window panel of Figure 1. It reads a date (month, day, year) and finds the corresponding day-of-the-year and day-of-the-week. Consider the case when this program is executed for the date January 1, 1990, i.e., (month=1, day=1, year=1990). Statements in reverse "video" in the figure show the static slice with respect to `date.day_of_the_year` at line 88. Figure 2 shows the corresponding dynamic slice. If day-of-the-year is computed incorrectly for this testcase, the value of `date.day` must be incorrect. Thus the error must be inside the procedure `read_date` invoked on line 50. Clearly, the dynamic slice in Figure 2 will help us localize the fault much more quickly compared to the static slice in Figure 1. Figures 1 and 2 are screen dumps of our prototype tool *Spyder* in operation.

In Section 2, we first present a framework for obtaining static and dynamic program slices when the program uses only scalar variables. Then in Sections 3 and 4 we extend our framework to handle pointers and composite variables such as arrays and records for static and dynamic cases, respectively. Section 5 discusses how our approach may be extended to the interprocedural case. Finally, Section 6 outlines related work.

2 Background

2.1 Notation

In the following sections we use a *let-in* construct (adapted from a similar construct in the programming language ML [16]). Consider the following generic use of *let*:

$$let <declarations> \ in <expression>$$

Here, $<declarations>$ consists of a sequence of name bindings that may be used inside $<expression>$. The scope of these bindings is limited to $<expression>$. The result of evaluating $<expression>$

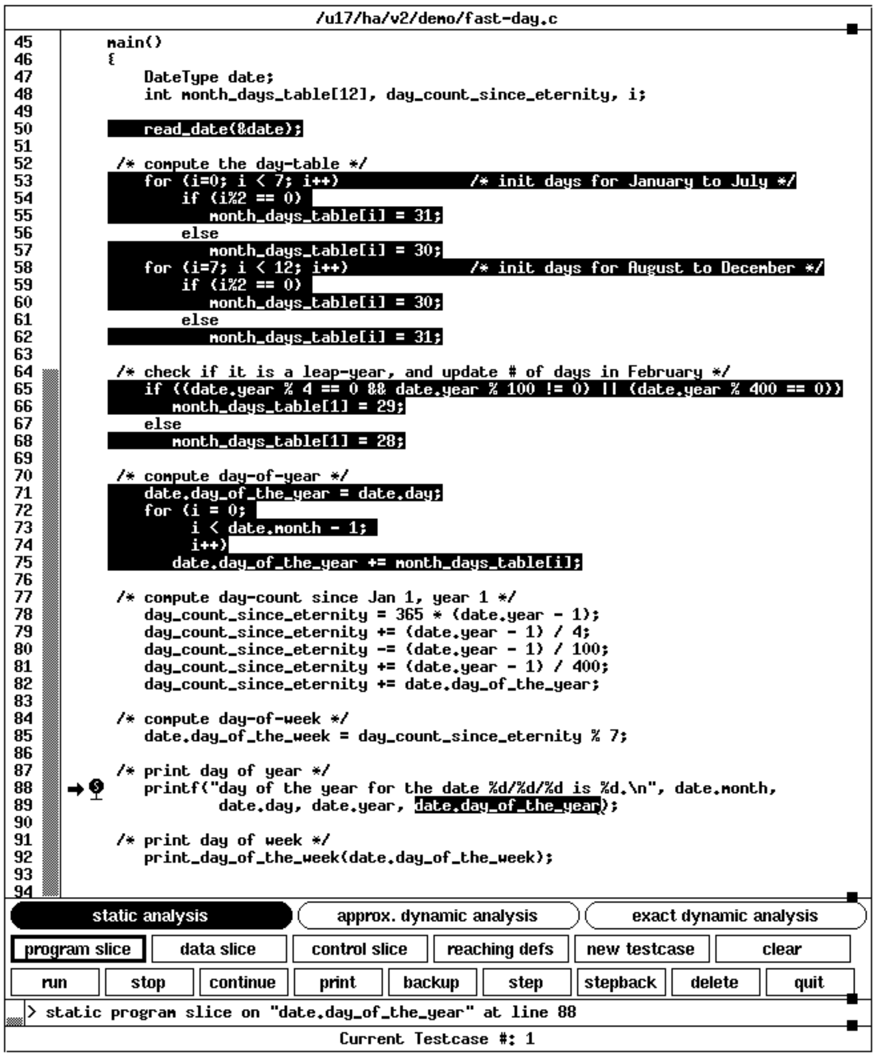

Figure 1: Static slice with respect to date.day_of_the_year at line 88.

is returned as the value of the *let* construct. For example, the following expression evaluates to 5.

$$let\ a = 2, b = 3\ in\ a + b$$

Names may also be bound using "pattern matching" between two sides of the symbol =. For example, if the complex number $X + Yi$ is represented by the tuple (X, Y), then the sum of two complex numbers $complex_1$ and $complex_2$ may be defined as follows:

$$\begin{aligned}
&sum(complex_1, complex_2) = \\
&\quad let\ complex_1 = (real_1, imaginary_1), complex_2 = (real_2, imaginary_2) \\
&\quad in\ (real_1 + real_2, imaginary_1 + imaginary_2)
\end{aligned}$$

In the above expression, $real_1$, $imaginary_1$, $real_2$, and $imaginary_2$ were all defined using pattern matching.

We also use $\bigcup$ notation to denote set unions. For example, if $S = \{x_1, x_2, \ldots, x_n\}$, then we have:

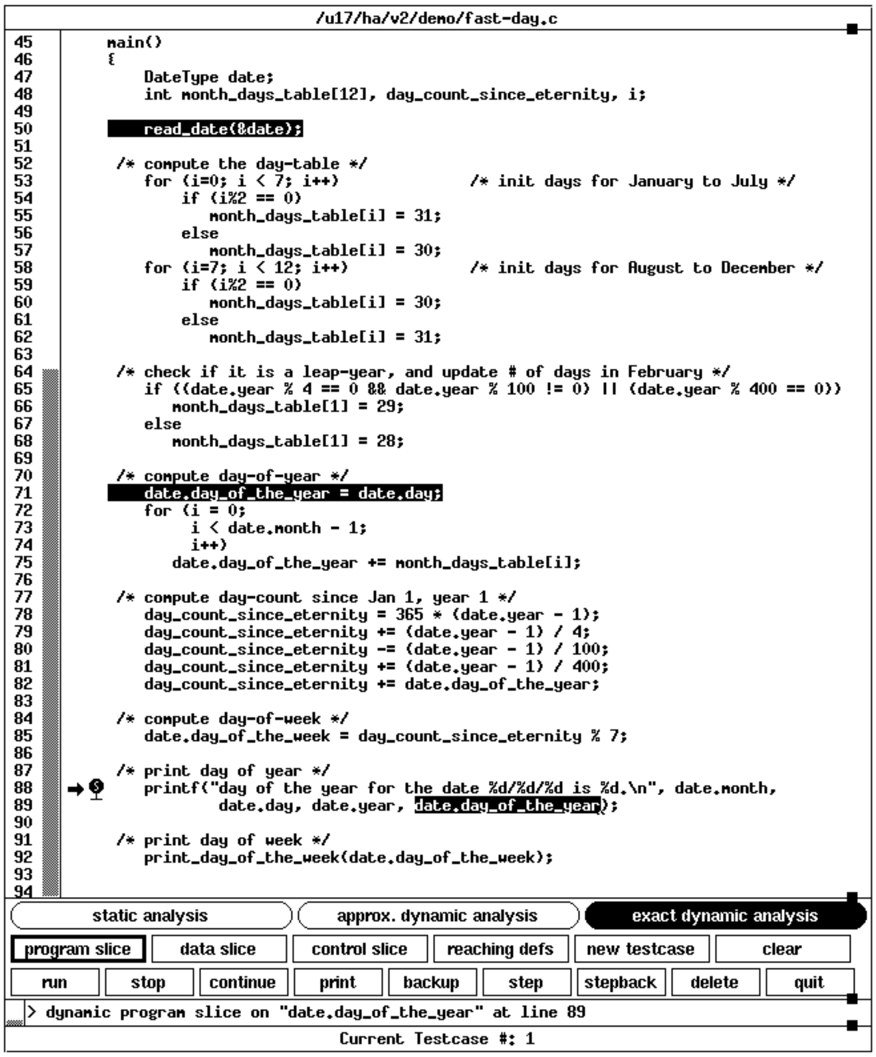

Figure 2: Dynamic slice with respect to date.day_of_the_year at line 89 for the testcase (month=1, day=1, year=1990).

$$\bigcup_{x \in S} f(x) \;\equiv\; f(x_1) \cup f(x_2) \cup \ldots \cup f(x_n)$$

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         I := 1;
S4:         while (I <= N)
            do
S5:            read(X);
S6:            if (X < 0)
               then
S7:                 Y := f1(X);
               else
S8:                 Y := f2(X);
               end_if;
S9:            Z := f3(Z, Y);
S10:           I := I + 1;
            end_while;
S11:        write(Z);
        end.
```

Figure 3: An Example Program

$\bigcup$'s may also be composed together. For example, if $S_1 = \{x_1, x_2\}$ and $S_2 = \{y_1, y_2\}$, then we have:

$$\bigcup_{x \in S_1} \bigcup_{y \in S_2} g(x, y) \equiv \bigcup_{\substack{x \in S_1 \\ y \in S_2}} g(x, y) \equiv g(x_1, y_1) \cup g(x_1, y_2) \cup g(x_2, y_1) \cup g(x_2, y_2)$$

2.2 Simple Static Slicing

The Flow-graph *Flow* of a program P is a tuple (V, A) where V is the set of vertices that correspond to simple statements and predicate expressions in the program (assignments, reads, writes, etc., and condition expressions in if-then-else, while-do, etc.)[2], and A is the set of directed edges between vertices in V. If there is an arc from node v_i to node v_j it means that control can pass from node v_i to node v_j during program execution. Each vertex in the flow-graph has a *use* and a *def* set associated with it. The *use* set of a vertex consists of all variables that are referenced during the computation associated with the vertex, and the *def* set consists of the variable computed at the vertex, if any.

Consider, for example, the program in Figure 3. Symbols f_1, f_2, and f_3, in statements S7, S8, and S9, respectively, are used to denote some unspecified side-effect-free functions with which we are not presently concerned; only the names of variables used in the computation are relevant. Labels S1, S2, etc., are included only for reference; they are not part of the program. Figure 4 shows the flow-graph for this program. Node annotations U and D show *use* and *def* sets, respectively, for all nodes in the flow-graph.

[2]In program optimization applications, vertices of a flow-graph correspond to *basic-blocks* in the program. But for our purposes, it is more convenient to associate vertices with simple-statements and predicates.

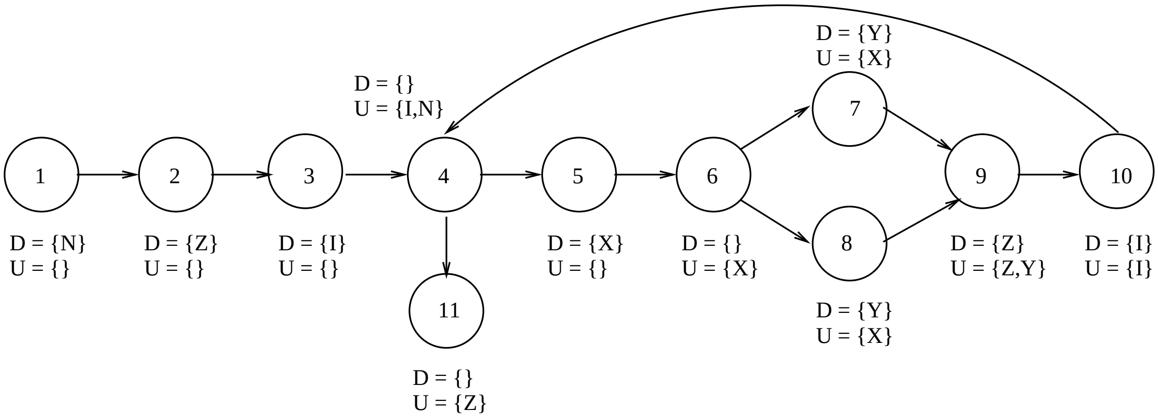

Figure 4: Flow Graph with *use*(U) and *def*(D) sets of the program in Figure 3

Static Reaching Definitions

Given a flow-graph, $\mathcal{F}$, a node, n, in $\mathcal{F}$, and a variable, var, we define $SRD(var,\ n,\ \mathcal{F})$, the set of all reaching definitions of variable var at node n in flow-graph $\mathcal{F}$, to be the set of all those nodes in $\mathcal{F}$ at which var is assigned a value and control can flow from that node to node n without encountering any redefinitions of var along the control-flow path. More precisely:

$$
\begin{aligned}
&SRD(var,\ n,\ \mathcal{F}) = \\
&\qquad let\ \mathcal{F} = (V,\ A) \\
&\qquad in\ \textstyle\bigcup_{(x,n)\in A}\ if\ \ var{\in}def(x)\ then\ \{x\} \\
&\qquad\qquad\qquad\qquad else\ SRD(var,\ x,\ (V,\ A{-}\{(x,n)\}))
\end{aligned}
$$

For example, for the flow-graph $\mathcal{F}$ in Figure 4, $SRD(\mathrm{Z},\ 11,\ \mathcal{F}) = \{2,9\}$.

Program Dependence Graph

The data dependence graph *DataDep* of a program P is a pair $(V,\ D)$, where V is the same set of vertices as in the flow-graph of P, and D is the set of edges that reflect data-dependencies between vertices in V. If there is an edge from vertex v_i to vertex v_j, it means that the computation performed at vertex v_i directly depends on the value computed at vertex v_j.[3] Or, more precisely:

$$
\begin{aligned}
&DataDep(P) = \\
&\qquad let\ Flow(P) = (V,\ A), \\
&\qquad\quad D = \bigcup_{\substack{n\in V\\ var\in use(n)\\ x\in SRD(var,\ n,\ Flow(P))}} \{(n,x)\} \\
&\qquad in\ (V,\ D)
\end{aligned}
$$

For example, the solid edges in Figure 5 denote data-dependencies among vertices of the flow graph in Figure 4. As $SRD(\mathrm{Z},\ 11,\ \mathcal{F}) = \{2,\ 9\}$, there are data dependence edges from node 11 to nodes 2 and 9 in Figure 5.

[3] At other places in the literature, particularly that related to vectorizing compilers, e.g., [9], the direction of edges in dependence graphs is reversed. For the purposes of program slicing, however, our definition is more suitable, as will become apparent later.

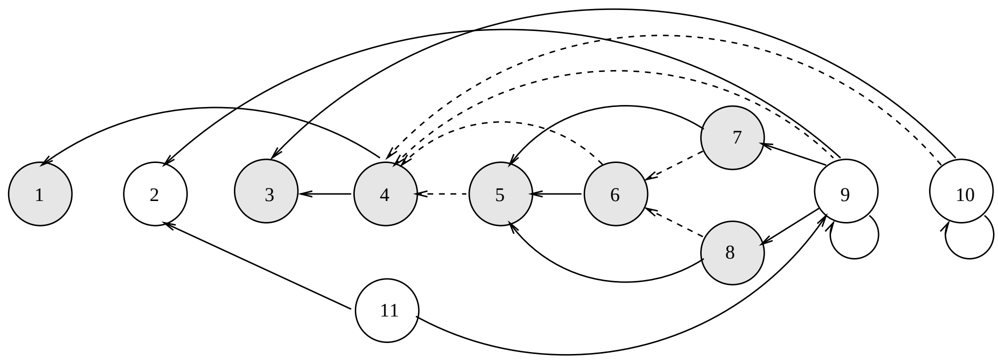

Figure 5: Program dependence graph of the program in Figure 3. Solid edges denote data-dependencies and dashed edges denote control-dependencies between vertices. Shaded nodes give the static slice with respect to variable Y at statement 9.

The control dependence graph *ControlDep* of a program P is a tuple (V, C), where V is the same set of vertices as in the flow-graph of P, and C is the set of edges that reflect control-dependencies between vertices in V. If there is an edge from v_i to v_j in *ControlDep*, it means that node v_i may or may not be executed depending on the boolean outcome of the predicate expression at node v_j.[4]

For example, the dashed edges in Figure 5 denote the control dependencies among its vertices. As statements 7 and 8 are immediately nested under the predicate at statement 6, there are control dependence edges from nodes 7 and 8 to node 6 in Figure 5.

$ControlPred(v)$ denotes the predicate statement upon which node v is control dependent. For example, for the program dependence graph in Figure 5, we have $ControlPred(3) = \phi$ whereas $ControlPred(7) = \{6\}$.

The Program dependence graph *ProgramDep* of a program P is obtained by merging the data and control dependence graphs of P.[5] Or,

$$
\begin{aligned}
&ProgramDep(P) = \\
&\qquad let\ DataDep(P) = (V,\ D),\ ControlDep(P) = (V,\ C) \\
&\qquad in\ (V,\ D \cup C)
\end{aligned}
$$

For example, Figure 5 shows the program dependence graph of the program in Figure 3.

Static Slice

Given a program, P, a node, n, in its flow-graph, and a variable, *var*, the static slice of P with respect to *var* at node n is constructed as follows: We first find all reaching definitions of *var* at node n. Then, from each reaching definition obtained, we find all reachable nodes in the program dependence graph of the program. $StaticSlice(P,\ var,\ n)$ can be precisely defined as follows:

$StaticSlice(P,\ var,\ n) =$

[4] This definition of control dependence is for programs with structured control flow. For such programs, the control dependence subgraph essentially reflects the nesting structure of statements in the program, and can be easily built in a syntax-directed manner. In programs with arbitrary control flow, a control dependence edge from vertex v_i to vertex v_j implies that v_j is the nearest inverse dominator of v_i in the control flow graph of the program [9].

[5] In other applications like vectorizing compilers, a data dependence graph may include other types of dependence edges besides data and control dependence, e.g., anti-dependence, output-dependence etc., but for the purposes of program slicing, the first two suffice.

let $\mathcal{F}$ = *Flow*(P), $\mathcal{D}$ = *ProgramDep*(P)
in $\bigcup_{x \in SRD(var,\, n,\, \mathcal{F})}$ *ReachableNodes*(x, $\mathcal{D}$)

where *ReachableNodes*(v, $\mathcal{G}$) is the set of vertices in $\mathcal{G}$ that can be reached from v by following one or more edges in $\mathcal{G}$.

For example, the shaded nodes in Figure 5 give the static slice with respect to variable Y at statement 9.

2.3 Simple Dynamic Slicing

Let $\mathcal{F}$ be the flow-graph of program P. Let *test* be a testcase consisting of a specific set of input-values read by the program. We denote the execution history of the program P for *test* by a sequence *hist* = $<v_1, v_2, \ldots, v_n>$ of vertices in $\mathcal{F}$ appended in the order in which they are visited during the program execution. The execution history at any instance denotes the partial program execution until that instance.

Consider, for example, the program in Figure 3 and testcase ($N = 2$, $X = -4, 3$). The execution history of the program for this testcase is $<1, 2, 3, 4^1, 5^1, 6^1, 7, 9^1, 10^1, 4^2, 5^2, 6^2, 8, 9^2, 10^2, 4^3, 11>$. Note that we use superscripts 1, 2, etc., to distinguish between multiple occurrences of the same statement in the execution history.

Last(*hist*) denotes the last node in *hist*, and *Prev*(*hist*) denotes the subsequence with all but the last node in *hist*. That is,

Last($<v_1, \ldots, v_{n-1}, v_n>$) = v_n
Prev($<v_1, \ldots, v_{n-1}, v_n>$) = $<v_1, \ldots, v_{n-1}>$

We use the notation <*Prev*(*hist*) | *Last*(*hist*)> to denote the two parts of *hist*. Also, <> denotes the empty sequence.

Also, *LastOccur*(v, *hist*) denotes the last occurrence of the node v in *hist*. For example, *LastOccur*(9, $<1, 2, 3, 4^1, 5^1, 6^1, 7, 9^1, 10^1, 4^2, 5^2, 6^2, 8, 9^2, 10^2, 4^3, 11>$) = $\{9^2\}$.

Dynamic Reaching Definitions

DRD(*var*, *hist*) denotes the last occurrence of the node in *hist* that assigns a value to *var*. Or,

DRD(*var*, <>) = ϕ
DRD(*var*, <*prevhist* | *lastnode*>) =
if *var*$\in$*def*(*lastnode*) *then* {*lastnode*}
else *DRD*(*var*, *prevhist*)

For example, *DRD*(Y, $<1, 2, 3, 4^1, 5^1, 6^1, 7, 9^1, 10^1, 4^2, 5^2, 6^2, 8, 9^2, 10^2, 4^3, 11>$) = {8}.

Note that both *LastOccur* and *DRD* result in either the empty set, implying no occurrence of the desired node, or in a singleton consisting of a unique node.

Dynamic Dependence Graph

The dynamic dependence graph, *DynamicDep*, of an execution history *hist* is a tuple (V, A), where V is the *multi-set* of flow-graph vertices (i.e., multiple entries of the same element are treated as distinct), and A is the set of edges denoting dynamic data dependencies and control dependencies between vertices. We use the symbol $\uplus$ to denote disjunctive union of elements. *DynamicDep* is defined as follows:

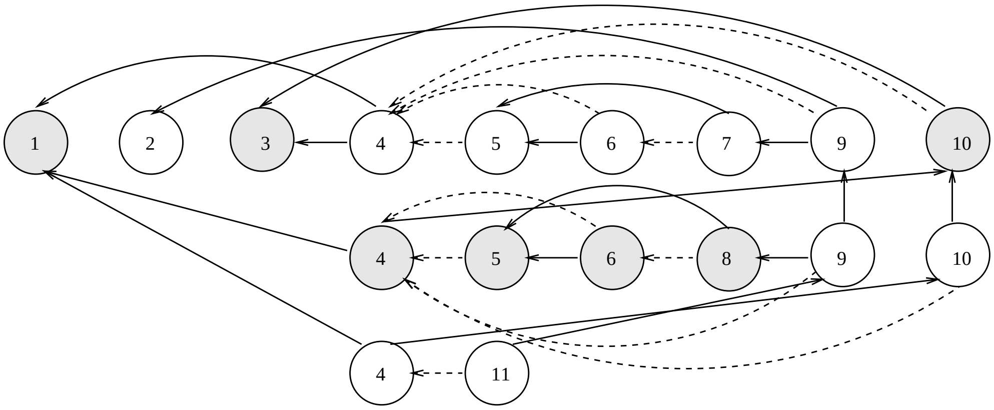

Figure 6: Dynamic dependence graph of the program in Figure 3 for testcase (N = 2, X = −4, 3). Solid edges denote data-dependencies and dashed edges denote control-dependencies between vertices. Shaded nodes give the dynamic slice with respect to variable Y at the end of program execution.

$$
\begin{aligned}
&\mathit{DynamicDep}(<>) = (\phi,\ \phi)\\
&\mathit{DynamicDep}(<\mathit{prevhist} \mid \mathit{next}>) =\\
&\qquad \mathit{let}\ \mathit{DynamicDep}(\mathit{prevhist}) = (V,\ A),\\
&\qquad\quad D = \bigcup_{\substack{var \in use(next)\\ x \in DRD(var,\ prevhist)}} \{(\mathit{next},\ x)\},\\
&\qquad\quad C = \bigcup_{\substack{x \in ControlPred(next)\\ y \in LastOccur(x,\ prevhist)}} \{(\mathit{next},\ y)\}\\
&\qquad \mathit{in}\ (V \uplus \{\mathit{next}\},\ A \cup D \cup C)
\end{aligned}
$$

Consider again the program in Figure 3, and the same testcase (N = 2, X = −4, 3). Figure 6 shows the corresponding dynamic dependence graph. Notice the two occurrences of node 9. The first occurrence depends on nodes 2 and 7 for values of variables Z and Y, respectively, whereas the second occurrence depends on its first occurrence and node 8 for the same values, respectively.

Dynamic Slice

Given an execution history, *hist*, of a program, P, on a test-case, *test*, and a variable, *var*, the dynamic slice of P with respect to *hist* and *var* is the set of all statements in *hist* whose execution had some *effect* on the value of *var* as observed at the end of the execution. Note that unlike static slicing where a slice is defined with respect to a given location in the program, we define dynamic slicing with respect to the end of execution history. If a dynamic slice with respect to some intermediate point in the execution is desired, then we simply need to consider the partial execution history up to that point.

Once we have constructed the dynamic dependence graph for the given execution history, we can easily obtain the dynamic slice for a variable, *var*, by first finding the node that corresponds to the last definition of *var* in the execution history, and then finding all nodes in the graph reachable from that node. *DynamicSlice* can be defined precisely as follows:

$$\mathit{DynamicSlice}(\mathit{hist},\ \mathit{var}) = \bigcup_{x \in DRD(var,\ hist)} \mathit{ReachableNodes}(x,\ \mathit{DynamicDep}(\mathit{hist}))$$

For example, the shaded nodes in Figure 6 give the dynamic slice for variable Y at the end of the execution for the testcase (N = 2, X = −4, 3).

3 Static Slicing with Pointers and Composite Variables

In Section 2.2, we defined reaching definitions for scalar variables. A definition of a variable *var* at a statement S_1 reaches its use at statement S_2, if there is a path from S_1 to S_2 in the flow-graph of the program, and no other node along the path defines *var*. But what if S_1 defines an array element $A[i]$, and S_2 uses an array element $A[j]$; if S_1 defines a field of a record $s.f$, and S_2 uses the whole record s; or if S_1 defines a variable X, and S_2 uses a pointer dereference expression $*p$? To be able to handle such situations, we introduce below the notion of intersection of two l-valued expressions.

Intersection of L-valued Expressions

An expression is said to be an l-valued expression if a memory location can be associated with it. A simple check to find if an expression is an l-valued expression is to check if it can appear on the left hand side of an assignment statement. For example, expressions var, $A[i]$, $s.f$, $B[i].r.x$, $*p$, are all l-valued expressions. On the other hand, none of the expressions 103, $x + y$, and $a > b$, is l-valued. The presence of pointers and composite variables such as arrays and records in a programming language requires that both *use* and *def* sets of statements be defined in terms of l-valued expressions.

A *use* expression e_1 is said to *intersect* with a *def* expression e_2, if the memory location associated with e_1 may overlap with that associated with e_2. We identify three types of intersections between l-valued expressions: *complete intersection, maybe intersection*, and *partial intersection.* We informally describe these below; see [1] for precise definitions.

Complete Intersection

A *use* expression e_1 completely intersects a *def* expression e_2 if the memory location associated with e_1 is totally contained in that associated with e_2. For example, consider the following code fragment:

```
S1:     X := ...
          ⋮
S2:       := ... X ...
```

Here, use of variable X at S2 completely intersects its definition at S1. Also, in the following code fragment,

```
S1:     s := ...
          ⋮
S2:       := ... s.f ...
```

use of field $s.f$ at S2 completely intersects the definition of record s at S1.

Maybe Intersection

Consider the following situation:

```
S1:     A[i] := ...
             ⋮
S2:          := ... A[j] ...
```

Whether or not the use of $A[j]$ at S2 intersects with the definition of $A[i]$ at S1 depends on the actual values of variables i and j at statements S1 and S2, respectively. If their values are the same, the two expressions intersect, otherwise they do not. We refer to such intersections as *maybe* intersections. Use of pointer dereferencing also causes *maybe* intersections. In the following code fragment,

```
S1:        *p := ...
              ⋮
S2:           := ...X ...
```

use of variable X at S2 *maybe*-intersects with the definition at S1 because the pointer variable p may or may not be pointing at variable X.

Partial Intersection

Consider the following scenario:

```
S1:        s.f := ...
              ⋮
S2:           := ...s ...
```

The whole record s is used at S2, but only one of its fields is defined at S1. A similar situation occurs if an array is used at S2, and only one of its elements is defined at S1. We refer to such intersections as *partial* intersections. If a *use* expression e_1 partially intersects with a *def* expression e_2, we define $PreExp(e_1, e_2)$ to be the portion of the memory location associated with e_1 that lies before that associated with e_2. Similarly we define $PostExp(e_1, e_2)$ to be the portion of the memory location associated with e_1 that lies after that associated with e_2.

Static Reaching Definitions Revisited

Let *CompleteIntersect*, *MaybeIntersect*, and *PartialIntersect* be boolean functions that determine if two l-valued expressions have *complete, maybe*, or *partial* intersections, respectively. We can now extend our definition of *SRD*, defined in Section 2.2 for programs involving only scalar variables, to that involving pointers and composite variables.

$$
\begin{array}{l}
SRD(var,\, n,\, \mathcal{F}) = \\
\quad let\ \mathcal{F} = (V,\, A) \\
\quad in\ \bigcup_{(x,n)\in A}\ if\ \ def(x) = \phi \\
\qquad\qquad then\ SRD(var,\, x,\, (V,\, A - \{(x,n)\})) \\
\qquad\qquad else\ let\ def(x) = \{var'\},\, A' = A - \{(x,n)\} \\
\qquad\qquad\quad in\ if\ CompleteIntersect(var,\, var') \\
\qquad\qquad\qquad then\ \{x\} \\
\qquad\qquad\qquad else\ if\ MaybeIntersect(var,\, var') \\
\qquad\qquad\qquad\quad then\ \{x\} \cup SRD(var,\, x,\, (V,\, A')) \\
\qquad\qquad\qquad\quad else\ if\ PartialIntersect(var,\, var') \\
\qquad\qquad\qquad\qquad then\ \{x\} \cup SRD(PreExp(var,\, var'),\, x,\, (V,\, A')) \\
\qquad\qquad\qquad\qquad\qquad \cup\ SRD(PostExp(var,\, var'),\, x,\, (V,\, A')) \\
\qquad\qquad\qquad\qquad else\ SRD(var,\, x,\, (V,\, A'))
\end{array}
$$

Note that *maybe* and *partial* intersections may occur together. For example, consider the following situation:

S1:   A[i].f := ...

$\vdots$

S2:   := ...A[j] ...

Because we check for *maybe* intersection before *partial* intersection, the former takes precedence over the latter whenever they occur together.

The definitions of data dependence, control dependence, and a static slice remain the same as given in Section 2.2. Only we now use the new definition of static reaching definitions described above to find the data dependence edges of the program dependence graph.

4 Dynamic Slicing with Pointers and Composite Variables

Dynamic slicing differs from static slicing in that the former has no *maybe* intersections. This implies that for each use of a scalar variable, there is at most one dynamic reaching definition; and for each use of a composite variable, there is at most one dynamic reaching definition of each of its scalar components. To define dynamic slices in the presence of composite variables and pointers, we generalize the notion of an l-valued expression to that of a memory cell. A *memory cell* is a tuple $(adr,\ len)$ where adr represents its address in memory, and len represents its length in bytes.[6] The memory-cell corresponding to an l-valued expression e_1 is given by the tuple $(AddressOf(e_1),\ SizeOf(e_1))$, where $AddressOf(exp)$ gives the current address associated with the l-valued expression exp at runtime, and $SizeOf(exp)$ gives the number of bytes required to store the value of exp. We now define *use* and *def* sets of all simple statements and predicates in terms of memory cells instead of l-valued expressions. Though the length component of these memory-cells may be determined at compile time, the address components may have to be determined at runtime just before the corresponding simple statement or predicate is executed.

Also, instead of determining intersection of l-valued expressions, we now check if two memory cells intersect. Using this formulation, we redefine DRD function as follows:

$DRD(cell,\ <>) = \phi$
$DRD((adr,\ 0),\ hist) = \phi$
$DRD(cell,\ hist) =$
  $let\ hist\ =\ <prevhist \mid next>$
  $in\ if\ \ def(next) = \phi$
    $then\ DRD(cell,\ prevhist)$
    $else\ let\ def(next) = \{cell'\}$
      $in\ if\ \ CellIntersect(cell,\ cell')$
        $then\ \{next\} \cup DRD(PreCell(cell,\ cell'),\ prevhist)$
            $\cup\ DRD(PostCell(cell,\ cell'),\ prevhist)$
        $else\ DRD(cell,\ prevhist)$

CellIntersect(*useCell*, *defCell*) returns true if there is any overlap between the two cells. *PreCell* and *PostCell* return the non-overlapping portions of the *useCell* that lie before and after the overlapping portion, respectively. It is possible that one or both of these portions may be empty

[6] Or the smallest addressable unit on the computer, e.g. a word. For languages where memory allocation for a variable is not necessarily contiguous, definition of a memory-cell may be changed to include the set of all its contiguous sub-cells.

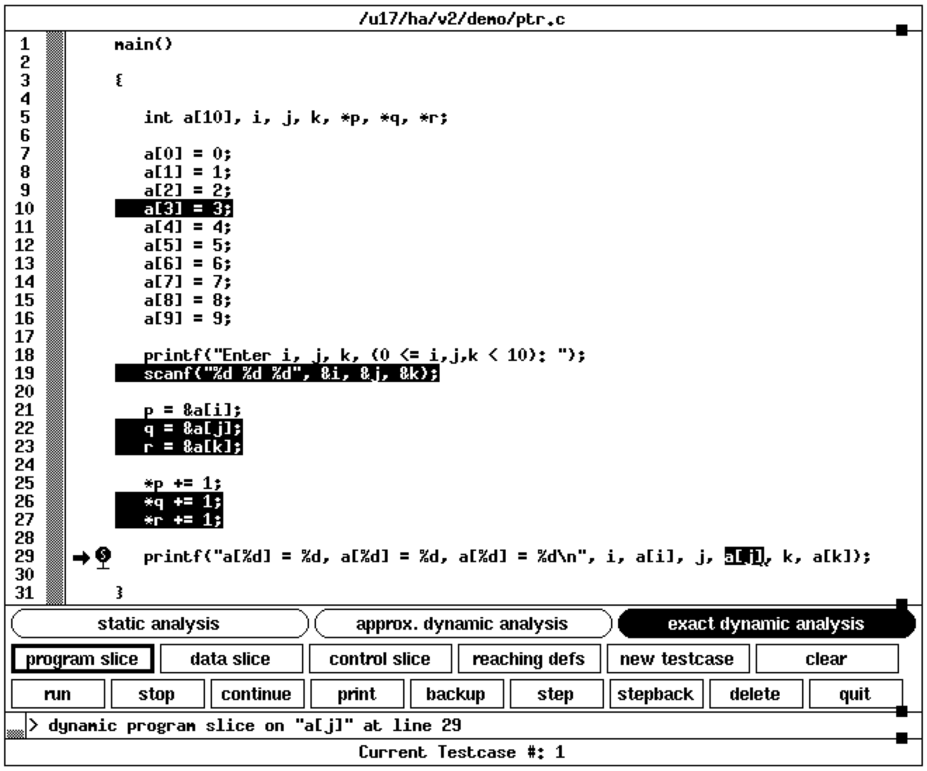

Figure 7: Dynamic slice with respect to a[j] on line 29.

(*len* = 0). The case when both pre- and post-cells are empty is analogous to *complete* intersection in static slicing; the case when one or both are non-empty is analogous to *partial* intersection; and, as we mentioned earlier, there are no *maybe* intersections in the dynamic case.

The advantage of this formulation is that all the usual problems associated with handling pointers in the static case are automatically taken care of in the dynamic case because all *use* and *def* sets are resolved in terms of memory cells; there is no ambiguity in determining if two memory cells overlap. As in static slicing, the definitions of data dependence, control dependence, and dynamic slice remain the same as given in Section 2.3. Only the definition of dynamic reaching definitions has changed.

Consider, for example, the simple program in Figure 7. It initializes the array a, reads three values i, j, and k, and increments the ith, jth and kth elements of the array, accessing these elements indirectly via pointers p, q, and r, respectively. The figure also shows the dynamic slice with respect to a[j] on line 29, for the testcase (i = 1, j = 3, k = 3). The static slice would have included the whole program.

Figure 8 shows a variant of the above program where a loop is used to initialize the array instead of using a separate assignment for each array element. If we execute this program for the same testcase (i = 1, j = 3, k = 3), we get the following output: a[1] = 2, a[3] = 4, a[10] = 0. Instead of printing the value of a[3] it prints that of a[10]. This implies that the value of k somehow got corrupted during the program execution. If we obtain the dynamic slice of k on line 27, we would expect only line 8 to be in the slice as that is the only place in the program where k is modified. Instead, we find that the loop on lines 10–17 is included in the dynamic slice, as shown in Figure 8. This suggests that the variable k was clobbered during the execution of the loop. Further

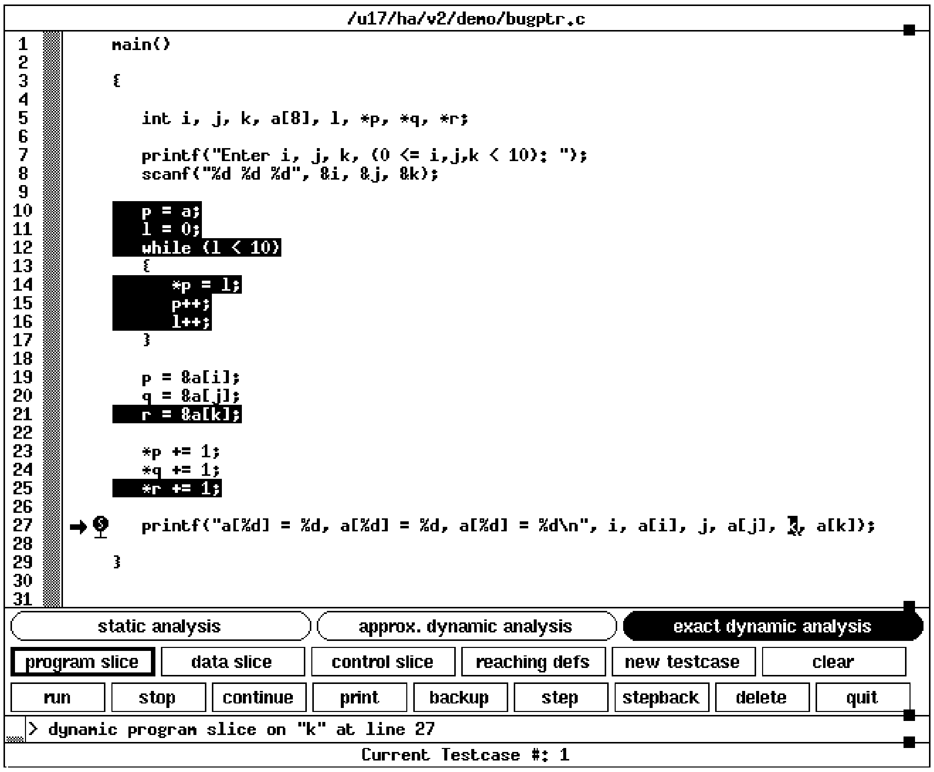

Figure 8: Dynamic slice with respect to k on line 27.

examination reveals that the fault indeed lies with the loop predicate: it iterates ten times when the array is declared to be only eight elements long. It is situations like this where precise dynamic analysis in terms of memory cells is helpful in revealing the fault.

5 Interprocedural Dynamic Slicing

The dynamic slicing approach described above can be easily extended to obtain slices of programs with procedures and functions. We first consider the case when parameters are passed by value, as in C. In this case, we simply need to treat a procedure invocation, $proc(actual_1, actual_2, \ldots, actual_n)$, to be a sequence of assignments $formal_i = actual_i, 1 \leq i \leq n$, where $formal_i$ is the ith formal parameter of *proc*. The *use* set of each of these assignments is computed in terms of memory-cells just before the procedure is invoked, and the *def* set is computed just after the control enters the procedure. Memory cells that correspond to *def* sets belong to the current activation record of *proc* on the stack.

Figure 9 includes a variant of the program in Figure 2 where segments of code in the main program have been moved inside procedures. Figure 9 also shows the interprocedural dynamic slice with respect to date.day_of_the_year on line 91 for the same testcase used for the dynamic slice in Figure 2.

Note that unlike interprocedural static slicing [12], our approach for dynamic slicing does not require that we determine which global variables are referenced inside a procedure, or which variables may be aliases to each other, nor do we need to eliminate name conflicts among variables in different procedures.

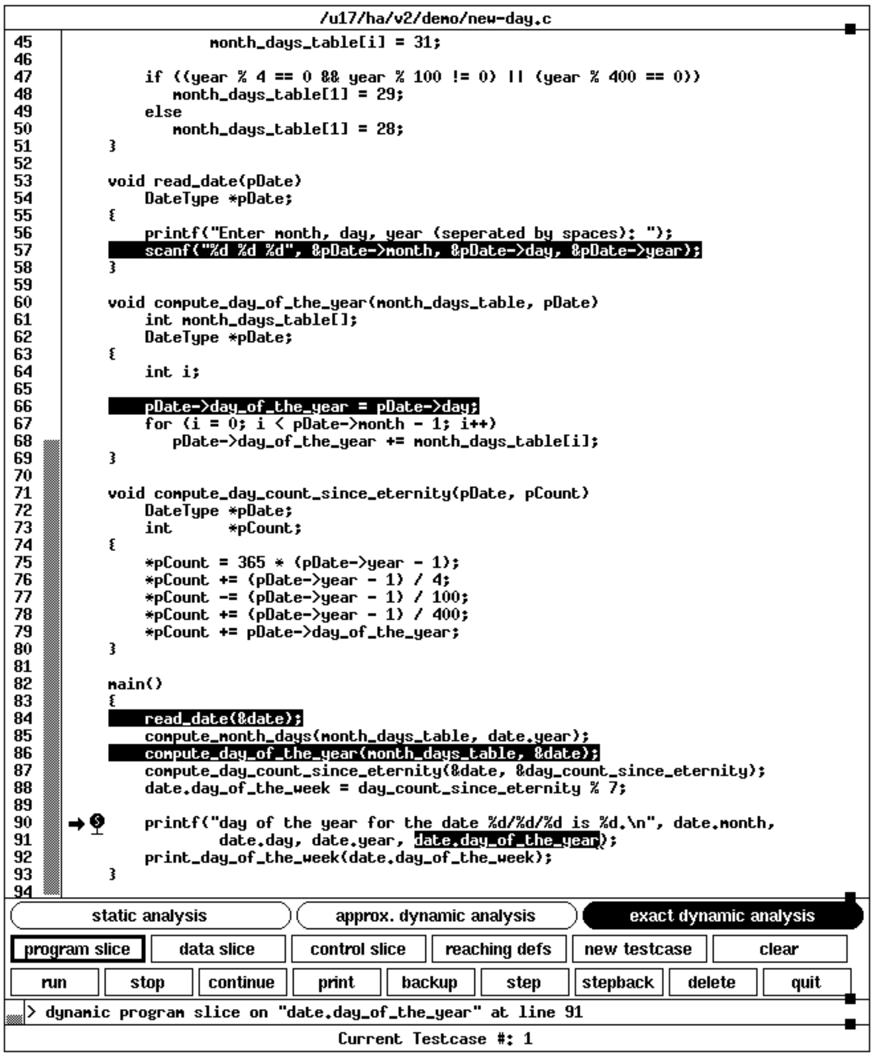

Figure 9: Dynamic slice with respect to date.day_of_the_year at line 91 for the testcase (month=1, day=1, year=1990).

Call-by-reference is even easier to handle: no initial assignments to formal parameters need to be made. The address of a formal parameter variable is automatically resolved to that of the corresponding actual parameter. Call-by-result parameter passing is handled by making assignments, $actual_i = formal_i$, just before control returns to the calling procedure. Call-by-value-result can be handled similarly by making appropriate assignments both at the beginning and the end of the procedure.

6 Related Work

The concept of static program slicing was first proposed by Weiser [19, 20]. Ottenstein and Ottenstein later presented an algorithm in terms of graph reachability in the program dependence graph, but they only considered the intraprocedural case [17]. Horwitz, Reps, and Binkley extended the program dependence graph representation to what they call the "system dependence graph" to find interprocedural static slices under the same graph-reachability framework [12]. Bergeretti and Carré have also defined information-flow relations somewhat similar to data- and control dependence relations, that can be used to obtain static program slices (referred to as "partial statements" by them) [7]. Podgurski and Clark have extended the regular notion of control dependence (which they refer to as "strong control dependence") to "weak control dependence" that includes interstatement dependencies involving program nontermination [18]. Uses of program slicing have also been suggested in many other applications, e.g., program verification, testing, maintenance, automatic parallelization of program execution, and automatic integration of program versions (see, e.g., [20, 7, 11]).

Korel and Laski extended Weiser's static slicing algorithms for the dynamic case [13]. Their definition requires that if any one occurrence of a statement in the execution history is included in the slice then all other occurrences of that statement be automatically included in the slice. For example, if the program in Figure 3 is executed for the testcase (N = 2, X = −4, 3), and we find the dynamic slice for the variable Y at statement S9 during the second iteration, their definition will require that statement S7 be also included in the dynamic slice, even though the current value of Y is totally unaffected by its execution. Miller and Choi also use dynamic dependence graph to perform flow-back analysis [6] in their Parallel Program Debugger PPD [15]. These approaches also treat array elements as separate variables. But as they do not resolve *use* and *def* sets in terms of memory cells, they will fail to detect interstatement dependencies like that illustrated in Figure 8. Also our approach provides a uniform way to handling pointers and all types of composite variables.

7 Conclusions

Static program slices tend to be large and imprecise when the program being debugged involves pointers and composite variables such as arrays, records, and unions. They lose their usefulness altogether if the language involved is not strongly-typed and permits use of unconstrained pointers. While debugging, however, we normally have a concrete testcase that reveals the fault and we wish to analyze the program behavior for that particular testcase. Dynamic program slices help us find interstatement dependencies for a given testcase. In this paper we have shown that we can find accurate dynamic slices even in the presence of unconstrained pointers and composite variables. The approach outlined provides a uniform framework for handling pointers as well as various types of composite variables. It does not require that the language be strongly-typed or that any runtime checks (out-of-bound array element reference, illegal pointer dereference, etc.) be performed.

Acknowledgments

We would like to thank Michal Young for our discussions with him on notations used in this paper and helping us write LaTeX macros for generating them.

References

[1] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1991.

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Efficient debugging with slicing and backtracking. Technical Report SERC-TR-80-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1990.

[3] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Fourth Symposium on Testing, Analysis and Verification (TAV4)*. ACM/IEEE-CS, October 1991. Also issued as SERC Technical Report SERC-TR-93-P.

[4] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, pages 21–26, May 1991.

[5] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. ACM SIGPLAN. SIGPLAN Notices, 25(6):246–256, June 1990.

[6] R. M. Balzer. Exdams: Extendible debugging and monitoring system. In *AFIPS Proceedings, Spring Joint Computer Conference*, volume 34, pages 567–580, Montvale, New Jersey, 1969. AFIPS Press.

[7] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[8] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. ACM SIGPLAN. SIGPLAN Notices, 25(6):296–310, June 1990.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[10] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989. ACM SIGPLAN. SIGPLAN Notices, 24(7):28–40, July 1989.

[11] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[12] Susan Horwitz, Thomas Reps, and David Binkeley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[13] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, October 1988.

[14] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. ACM SIGPLAN, July 1988. SIGPLAN Notices, 23(7):21–34, July 1988.

[15] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988. ACM SIGPLAN. SIGPLAN Notices, 23(7):135–144, July 1988.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[17] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittaburgh, PA, April 1984. ACM SIGSOFT/SIGPLAN. SIGPLAN Notices, 19(5):177–184, May 1984.

[18] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[19] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

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