

# Application of Dynamic Slicing in Program Debugging\*

Bogdan Korel, Jurgen Rilling  
Department of Computer Science  
Illinois Institute of Technology  
Chicago, IL 60616, USA  
korel@charlie.iit.edu

## Abstract

A dynamic program slice is an executable part of a program whose behavior is identical, for the same program input, to that of the original program with respect to a variable(s) of interest at some execution position. In the existing dynamic slicing tools dynamic slices are represented in a textual form, i.e., a dynamic slice is displayed to programmers in the form of highlighted statements or in the form of a subprogram. Although dynamic slicing does narrow the size of the program, it is still up to the programmer to analyze the text of a dynamic slice and identify a faulty part in the program. The textual representation of a dynamic slice does not provide much guidance in program debugging and understanding of program behavior, which frequently is a major factor in efficient debugging. During dynamic slice computation different types of information are computed and then discarded after computation of the dynamic slice. In this paper we propose new dynamic slicing related features that exploit this information to improve the process of program debugging. These features were implemented in our dynamic slicing tool that can be used for program debugging.

## 1. Introduction

A static program slice consists of all statements in program  $P$  that may affect the value of variable  $v$  at some point  $p$  [19, 20]. Slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance (e.g., [5, 7, 8, 17, 19]). As originally introduced, static slicing involves all possible program executions. In debugging practice, however, we typically deal with a particular incorrect execution and, consequently, are interested in locating the cause of incorrectness of that execution. Therefore, we are interested in a slice that preserves the program behavior for a specific program input, rather than that for the set of all inputs. This type of slicing is referred to as *dynamic* slicing [11]. Several different techniques for computation of dynamic slices have been proposed, e.g., [1, 6, 9, 13, 16]. By taking a particular program execution into account, dynamic slicing may significantly reduce the size of the slice as compared to static slicing. The notion of dynamic slicing has been also extended for distributed programs [3, 4, 14]. Dynamic program slicing is not only used in software debugging but also in software maintenance and software testing [2, 7, 10, 21, 22].

---

\* This research has been partially supported by the NSF Research Initiation Award grant CCR-9308895

In the existing dynamic slicing tools, dynamic slices are represented in a textual form, i.e., a dynamic slice is displayed to programmers in the form of highlighted statements or in the form of a subprogram. Although dynamic slicing does narrow the size of a program, it is still up to the programmer to analyze the text of a dynamic slice and identify the faulty part in the dynamic slice. In addition, the textual format of a dynamic slice does not provide much guidance in program debugging and understanding of program behavior. Understanding of program behavior is frequently a major factor in efficient debugging. During dynamic slice computation different types of information are computed and then discarded after computation of the dynamic slice. In this paper we have proposed new dynamic slicing related features that exploit this information for the purpose of program debugging, e.g., executable dynamic slices, partial dynamic slicing, influencing variables, and contributing nodes. These features were implemented in the dynamic slicing tool that is used to improve the process of program debugging. These dynamic slicing related features have evolved during experimentation with the dynamic slicing tool. Our preliminary experience has shown that in many debugging situations these features may be helpful for programmers during fault localization.

In section 2, basic concepts used in the paper are presented. Section 3 overviews the existing research in dynamic program slicing. In Section 4 the dynamic slicing related concepts are introduced. Section 5 presents the dynamic slicing tool that supports the presented concepts. Finally, in Conclusions future research is highlighted.

## 2. Background

A program structure is represented by a flow graph(s) that consists of a set of nodes and a set of arcs. A node corresponds to an assignment statement, an input or output statement, a procedure call statement, a procedure entry statement, a goto statement, a break statement, a continue statement, a label statement, or the predicate of a conditional or a loop statement, in which case it is called a *test* node. An arc corresponds to a potential transfer of control from one node to another node. An *execution trace* is a sequence of nodes that has actually been executed for some input during program execution. For example,  $T_x = \langle 7, 8, 9, 10, 11, 12, 10, 11, 12, 10, 11, 12, 10, 13, 14, 15, 16, 17, 18, 19, 1, 2, 3, 20, 4, 5, 21, 22, 17, 18, 22, 17, 23 \rangle$  is the execution trace when the program in Figure 1 is executed on the input data  $n=3, a=(1,2,-3), k=1$ ; this execution trace is presented in Figure 2 in a more detail. An execution trace is an abstract list (sequence) whose elements are accessed by position in it. Node  $X$  at position  $p$  in  $T_x$  will be written down as  $X^p$  and referred to as an *action*. For instance,  $16^{17}$  is an action, i.e., node 16 at position 17.  $X^p$  is a test action if  $X$  is a test node. By  $v^q$  we denote *variable  $v$  at position  $q$* , i.e., variable  $v$  before execution of node  $T_x(q)$ . The notion of execution position is introduced in here only for presentation purposes, i.e., programmers do not identify the execution position of a node in these terms (they usually express the execution position differently, e.g.,  $17^{18}$  may be expressed as execution of node 17 at the entry to the while loop, and  $17^{29}$  as execution of node 17 after one iteration of this loop). An *use* of variable  $v$  is an action in which this variable is referenced. A *definition* of variable  $v$  is an action which assigns a value to that variable. Used and defined variables are determined during program execution by capturing the memory address for each used or defined variable.

```

program sample;
var
n,max,min,s,i,k: integer;
a: array [1..100] of integer;
1 procedure find_max(s1:integer; var s2:integer);
  begin
2,3   if s1>s2 then s:=s1; /* correctly s2:=s1;
      end;
4 procedure Find_min(x: integer; var y: integer);
  begin
5,6   if x<y then y:=x;
      end;

begin
7   readln (n) ;
8   readln (k) ;
9   i:=1 ;
10  while i <= n do begin
11   readln (a[i]);
12   i := i + 1;
      end ;
13  max := a[1] ;
14  min := a[1] ;
15  s := a[1] ;
16  i := k+1;
17  while i <= n do
  begin
18   if a[i]>0 then
  begin
19     Find_max(a[i],max);
20     Find_min(a[i],min);
21     s := s + a[i] ;
      end;
22   i := i + k ;
  end ;
23  write(max,' ',min,' ',s) ;
end.

```

**Figure 1.** A sample program.

```

71   readln (n)
82   readln (k)
93   i := 1
104  i <= n
115  readln (a[i])
126  i := i + 1
107  i <= n
118  readln (a[i])
129  i := i + 1
1010 i <= n
1111 readln (a[i])
1212 i := i + 1
1013 i <= n
1314 max := a[1]
1415 min := a[1]
1516 s := a[1]
1617 i := k+1
1718 i <= n
1819 a[i]>0           { a[2] < 0 }
1920 Find_max(a[i],max);   { a[2] }
2121 procedure Find_max(s1:integer;var s2:integer);
2222 s1>s2
2323 s:=s1;
2024 Find_min(a[i],min);   { a[2] }
425 procedure Find_min(x:integer; var y:integer);
526 x<y
2127 s := s + a[i]           { s:=s + a[2] }
2228 i := i + k
1729 i <= n
1830 a[i]>0           { a[3] < 0 }
2231 i := i + k
1732 i <= n
2333 writeln(max,' ',min,' ',s)

```

**Figure 2.** An execution trace of the program of Figure 1 on input: n=3, a=(1,2,-3), k=1.

### 3. Dynamic program slicing

A dynamic slice is an executable part of the program whose behavior is identical, for the same program input, to that of the original program with respect to a variable of interest at some execution position  $q$ . A slicing criterion of program  $P$  executed on program input  $x$  is a tuple  $C=(x,y^q)$  where  $y^q$  is a variable at execution position  $q$ . A *dynamic slice* of program  $P$  on slicing criterion  $C$  is any syntactically correct and executable program  $P'$  that is obtained from  $P$  by deleting zero or more statements, and when executed on program input  $x$  produces an execution trace  $T'_x$  for which there exists the corresponding execution position  $q'$  such that the value of  $y^q$  in  $T_x$  equals to the value of  $y^{q'}$  in  $T'_x$ . A dynamic slice  $P'$  preserves the value of  $y$  for a given program input  $x$ . It is assumed that an environment in which  $P$  and  $P'$  are executed assigns the same initial values to variables in  $P$  and  $P'$ . There may be many different dynamic slices for the same slicing criterion, and the goal is to find the slice with the minimal number of statements. This goal may not be achievable in general. However, it is possible to determine a safe approximation of the dynamic slice that will preserve the computation of the values of variables of interest.

A dynamic slice has originally been defined [11] as an executable program (slice). Different versions of the original dynamic slice have been proposed in the literature, e.g., [1, 9]; in most cases, they are non-executable "pieces" of code. Several algorithms have been proposed for dynamic slice computation, e.g., [1, 6, 11, 15]. Most of the existing methods of dynamic slice computation are based on "backward" analysis, i.e., after the execution trace of the program is recorded, the dynamic slicing algorithm traces backwards in the execution trace to compute the dynamic slice. For long program executions a forward approach of dynamic slice computation was proposed in [15]. It has been shown [13, 18] that taking into account a particular program execution dynamic slicing may significantly reduce the size of the slice as compared to static slicing. Since dynamic program slicing is based on the actual program execution, different types of "run-time" information about the program can be collected during program execution, e.g., the values of array indexes and pointers are known at each step of program execution. Dynamic slicing derivation methods exploit this information in order to compute more precise dynamic slices.

In this paper we concentrate on the computation of executable dynamic slices originally defined in [11]. The other forms of dynamic slices are subsets of the executable dynamic slices and are much easier to compute. Several algorithms have been proposed for dynamic slice computation, e.g., [1, 6, 11]. Most of those algorithms use the notion of *data* and *control dependencies* to compute dynamic program slices. After the execution trace of the program is recorded, the dynamic slice is derived from the recorded execution trace using the data and control dependencies. The data dependence captures the situation where one action (node) assigns a value to an item of data and the other action (node) uses that value. For example, in the execution trace of Figure 2, 3<sup>23</sup> assigns a value to variable  $s$  and 21<sup>27</sup> uses that value. The dynamic control dependence between actions captures the dependence between test actions and actions that have been chosen to be executed by these test actions. For instance, in the execution trace of Figure 2, the execution of 19<sup>20</sup> is depended on the outcome of test action 18<sup>19</sup>, but 22<sup>28</sup> is not dependent

on the outcome of 18<sup>19</sup>. The dynamic dependencies are used to compute a dynamic slice by tracing backwards the dependencies between actions in  $T_x$  starting from action  $v^q$ . After setting all actions as unmarked and not visited, the last definition of  $y^q$  is identified and marked, where by a *last definition* of variable  $v^k$  in execution trace  $T_x$  we mean action  $Y^p$  that assigns a value to variable  $v$  and  $v$  is not modified between positions  $p$  and  $k$ . For example, the last definition of variable  $s^{33}$  in execution trace of Figure 2 is action 21<sup>27</sup>. In the next step of the algorithm a marked and not visited action  $X^k$  is selected and set as visited. All last definitions of all variables used in  $X^k$  are marked (this step corresponds to finding data dependencies between actions), and then all actions for which there exists a control dependence between them and  $X^k$  are marked. This process of selecting of marked and not visited actions continues until all marked actions are visited. A dynamic slice is constructed from  $P$  by removing nodes (statements) whose actions were not marked in  $T_x$ . All actions that are marked by the dynamic slicing algorithm are referred to as *contributing* actions, whereas all actions that are not marked are referred to as *non-contributing* actions. A more detailed explanation of this algorithm can be found in [13]. When this dynamic slicing algorithm is applied for variable  $s$  at execution position 33 in the execution trace of Figure 2, the algorithm marks all actions that contribute to the computation of variable  $s$  at position 33 (this marking is presented in Figure 4 in which all contributing actions are shown in bold). The dynamic slice is constructed by removal of all statements (from the original program) whose actions are not marked during dynamic slice computation, and this dynamic slice is presented in Figure 3a. Similarly, a dynamic slice for variable  $max$  at position 33 is computed, and it is shown in Figure 3b.

We have developed a dynamic slicing tool that supports dynamic slicing for Pascal programs. The dynamic slicing tool has been developed in object-oriented Pascal and under a Windows environment. This dynamic slicing tool is an extension of the debugging environment PELAS [12]. In this tool, dynamic slicing is used to guide programmers in the process of program debugging. Several dynamic slicing tools have also been reported, e.g., [2, 9], in the literature. In order to compute dynamic slices, dynamic slicing tools provide certain functionality of conventional debuggers, e.g., breakpoints, step-wise execution, etc. Our tool also supports these features provided by conventional debuggers. The following sections provide a brief overview of the major functions of the dynamic slicing tool.

### Breakpoints

Setting a breakpoint in the tool can be done just by clicking the left mouse button on any line in the source code. The line will be automatically highlighted in red color. When a breakpoint is reached during program execution, the program is suspended. The programmer can then examine various components of the program state and verify its correctness, e.g., the values of program variables. Figure 6 shows a breakpoint at the beginning of the while loop and the current execution position at the call to the *Find\_max* procedure. Removing breakpoints is done in a similar way. Setting and removing breakpoints can be also be done from the *Execute* menu shown in Figure 6.

```

program sample;
var
n,max,min,s,i,k: integer;
a: array [1..100] of integer;
1  procedure Find_max(s1:integer; var s2: integer);
    begin
2,3  if s1>s2 then s:=s1;
    end;
begin
7   readln (n) ;
8   readln (k) ;
9   i:=1 ;
10  while i <= n do begin
11   readln (a[i]);
12   i := i + 1;
    end ;
13  max := a[1] ;
16  i := k+1;
17  while i <= n do
    begin
18   if a[i]>0 then
        begin
19   Find_max(a[i],max);
21   s := s + a[i] ;
        end;
22   i := i + k ;
    end ;
23  write(max,' ' ,min,' ' ,s) ;
end.

```

```

program sample;
var
n,max,min,s,i,k: integer;
a: array [1..100] of integer;
begin
7   readln (n) ;
8   readln (k) ;
9   i:=1 ;
10  while i <= n do begin
11   readln (a[i]);
12   i := i + 1;
    end ;
13  max := a[1] ;
23  write(max,' ' ,min,' ' ,s) ;
end.

```

**a.** A dynamic slice for *s* at node 23.

**b.** A dynamic slice for *max* at node 23.

**Figure 3.** Dynamic program slices variables *s* and *max* at node 23.

### Program Execution

The tool supports two modes of program execution: continuous execution and step-wise execution. A programmer may execute a program in the step-wise mode by pressing a special key (in this case, only one statement is executed). During program execution the current execution position is highlighted in yellow color in the program text. Executing program can also be done from the *Execute* menu shown in Figure 6.

### Finding Dynamic Slices

The first step to compute a dynamic slice is to execute the program. When a program is executed and its execution is suspended at some point, e.g., at a breakpoint, the programmer can specify a variable for which a dynamic slice should be derived. At any execution position a programmer may request the computation of a dynamic slice for a selected variable(s) by selecting *Compute Slice* option in the *Slice* menu. The tool prompts the user to enter a variable name and then

computes the dynamic slice. The programmer can view the dynamic slice by selecting the *Slice/Show Slice* on the menu bar. Figure 11 shows the dynamic slice for variable *s* at the `writeln` statement in the sample program of Figure 1 that was executed on the input shown in Figure 2. The derived dynamic slice is displayed to a programmer in the form of highlighted statements in the program text (see Figure 11a), or the dynamic slice is shown by removing from the original program all statements that do not affect the value of the selected variable (see Figure 11b).

#### 4. Dynamic slicing related concepts in program debugging

Dynamic slicing has originally been proposed to guide programmers in the process of program debugging by narrowing the size of the suspected part of incorrect code, but it can also be used in the process of program understanding of correct programs during software maintenance. In debugging programmers are interested in the localization of a fault that caused an incorrect output(s) during program execution on a particular program input. One aid to program debugging is to reduce the amount of detail a programmer sees. Program slicing transforms a large program into a smaller one that contains only statements relevant to the computation of an incorrect output. Understanding of program behavior is frequently a major factor in efficient debugging. However, the current slicing techniques do not provide any means to help the understanding of program behavior related to the incorrect output.

Typically, a program performs several functions and contains several outputs. We assume that each program output can be represented by a variable or a set of variables at a certain program point. For example, the program of Figure 1 has three outputs: Variable *max* at statement 23 represents the function of finding the maximal value, variable *min* represents the function of finding the minimal value, and variable *s* represents the function of finding the sum of array elements. When this program is executed on the input  $n=3$ ,  $a=(1,2,-3)$ ,  $k=1$ , it produces the following output:  $max=1$ ,  $min=1$ ,  $s=4$  (the execution trace is shown in Figure 2). However, the expected output is  $max=2$ ,  $min=1$ , and  $s=3$ . A programmer may be now interested in the localization of a fault(s) related to the incorrect computation of the sum of array elements and the maximal array element that are represented by variables *s* and *max*, respectively.

Static slicing [19] may be used to identify these parts of the program that potentially contribute to the computation of the incorrect output for all possible programs inputs. Static slicing is helpful to gain a general understanding of these parts of the program that contribute to the incorrect output. Although static slicing has many advantages in the process of program debugging, static slices are frequently still large subprograms because of the imprecise computation of these slices.

Dynamic slicing may more precisely identify these parts of the program that contribute to the computation of the incorrect output for a given program execution. A slice (static or dynamic) of a program is usually represented in the textual form, i.e., a dynamic slice is displayed to programmers in the form of highlighted statements in the original program or in the form of a subprogram by removing all statements from the original program that do not affect the incorrect output. Although dynamic slicing does reduce the size of a slice, it is still up to the programmer to identify the suspected part of the program in the slice and, eventually, to find new places of incorrectness that can lead to the localization of a fault. For some programs there may be a relatively small decrease in the size of a slice, or because of the significant size of the software

system, the size of a slice may still be very large and hard to comprehend. Using traditional program slicing methods programmers may still have difficulties to understand the program and its behavior and to identify the cause of incorrect program behavior. For example, dynamic slices of Figure 3 computed for incorrect variables  $s$  and  $max$  may be of limited use. The dynamic slice for variable  $s$  is almost as big as the original program and may be not very useful because of its size. On the other hand, the slice for  $max$  is too small to localize the fault in the program (notice that this slice does not contain the faulty statement because of the lack of influence of the faulty statement on the variable  $max$ ).

In this situation program slicing is of limited use, and understanding of program behavior becomes a major element in efficient debugging. Since dynamic slices are only represented in the textual form, they may provide limited guidance in the process of debugging and understanding of program behavior. Therefore, it is important to devise methods that concentrate the programmer's attention only on the most essential parts of the program and its execution that relate to the incorrect output.

Traditionally, in order to understand a program's behavior, a programmer uses conventional debuggers that support breakpoint facilities and step-wise program execution. Breakpoints allow a programmer to specify places in a program where the execution should be suspended. When a breakpoint is reached and the execution is suspended, the programmer can then examine various components of the program state and check the correctness of values of variables. Programmers may also execute a program in a step-wise manner in order to observe the program execution. Conventional debuggers, however, do not provide any means for identification of contributing program parts of the program being debugged. Using debuggers is an inefficient and time consuming approach of understanding of program behavior, especially when a programmer is interested in observing only these parts of the program behavior that relate to the incorrect output. The programmer may observe a large amount of unrelated computation and it is frequently almost impossible for him/her to distinguish related computations from unrelated computations. In order to make the process of program debugging more efficient it is important to focus the programmer's attention on the "essential" components (statements, variables, etc.) of the program and its execution. Dynamic slicing techniques provide means to prune away unrelated computation. During dynamic slice computation different types of information are computed related to the program execution, for example, contributing actions and non-contributing actions. After computation of a dynamic slice, all this information is discarded. We propose to take advantage of the already computed information and use it in the process of program debugging.

In what follows we describe the concepts of *executable slices*, *influencing variables*, *contributing nodes*, and *partial dynamic slicing* that are "extensions" of executable dynamic slicing. These concepts are supported by our dynamic slicing tool. In the following discussion it is assumed that a dynamic slice for slicing criterion  $C=(x,y^q)$  has been computed (where  $y$  corresponds to the incorrect output). In particular, we assume that a dynamic slice for the incorrect variable  $s$  at the writeln-statement of the program of Figure 1 has already been computed. Figure 11 shows this dynamic slice. Notice that the dynamic slice for variable  $max$  is not used because it is too small.

### **Executable slice**

An obvious extension of dynamic slicing is the provision for executable slices, i.e., programmers should be able to execute dynamic slices in order to observe the slice execution to find new places of incorrect program behavior that relates to the computation of the incorrect output. Clearly, programmers should be able to execute a dynamic slice, suspend its execution at breakpoints, and examine values of selected variables to find new places of incorrectness. One approach is to create a dynamic slice (a subprogram) compile it, and then execute it. However, during execution of a dynamic slice, values of some variables may be meaningless (i.e., they are not equal to the values of the same variables in the original program at the corresponding position) because some statements have been removed from the original program. Notice that in a dynamic slice computed for slicing criterion  $C=(x,y^q)$  only the value of variable  $y$  at position  $q$  is preserved. The remaining variables may not necessarily have the same values. Since, it is almost impossible for programmers to recognize the "meaningless" values of variables during execution of the dynamic slice, it would therefore be important that a slicing tool identify at each point of the slice execution variables that have the same values during slice execution and the execution of the original program. This approach may be inconvenient to use because of the need of recompilation of the dynamic slice. An alternative approach is to execute an original program, display a dynamic slice to the programmer, and show only these parts of the program execution that relate to the dynamic slice, i.e., parts of the execution that correspond to contributing actions (parts of the execution that correspond to non-contributing actions are not shown to the programmer). The latter approach has been implemented in our dynamic slicing tool. The tool uses information about contributing and non-contributing actions to display or skip (not display) the dynamic slicing related computation. The major advantage of this approach is that the values of variables are the same in the slice and in the original program.

### **Influencing Variables**

While observing a program or slice execution, one important question arises as to what variables should be observed in order to increase the chances of finding new places of incorrect program behavior. This becomes very important for programs with a large number of variables. Dynamic slicing algorithms allow the "essential" variables to be identified, i.e., those variables at each point of program execution that have influence on the value of  $y^q$ . We refer to those variables as "influencing" variables. Informally, variable  $z^p$  ( $p < q$ ) is an "influencing" variable with respect to  $y^q$  if its value is used to compute the value of variable  $y^q$  by one of the contributing actions between  $p$  and  $q$ , i.e., an influencing variable contributes to the computation of the value of  $y^q$ . At each step of program execution, a list of "influencing" variables that influence the value of  $y^q$  may be displayed to the programmer. For example, after a dynamic slice for variable  $s$  at statement 23 was computed (a dynamic slice is shown in Figure 3a), it is possible to identify at each execution position variables that are used to compute variable  $s$  at node 23. Figure 4 contains an execution trace of Figure 2 together with influencing variables at every execution position. By focusing a programmer's attention on the influencing variables, it is easier for him/her to understand the influence of variables in the execution trace on the incorrect output. By reducing the number of variables to be observed, the process of debugging may be more efficient. Moreover, by observing a smaller number of variables during program execution, it may be easier for programmers to identify the "existence" or "lack" of influence of some

variables on the incorrect program output. It is very likely that these variables carry incorrect values. For example, variable  $s$  does not have influence on the final value of  $s$  at the entry to the while loop in the execution trace of Figure 2. This observation may lead, for example, to the early localization of the fault in the program of Figure 1. In our tool, the user can select the *Analysis/Influencing Variables* option from the menu to display influencing variables at any execution position. The influencing variables are listed in a separate window. In Figure 7 influencing variables are listed for the current execution position at " $s:=s+a[i]$ " statement.

### Contributing nodes

Certain parts of the execution do not contribute to the computation of the incorrect output for which a dynamic slice was computed. In many situations a programmer may be interested in stepping only through the program execution (or dynamic slice execution) that contributes to the computation of the incorrect output. It is possible that an action of node  $X$  contributes to the computation of the incorrect output but a different action of the same node  $X$  in the same execution trace does not contribute to the computation of this output. It is almost impossible for a programmer to distinguish contributing and non-contributing actions in the execution trace. Therefore, we have developed a simple technique which may help programmers to better understand the contributing computations. Since during dynamic slice computation contributing and non-contributing actions are identified, they are used to indicate to the programmer the executions that contribute to the incorrect output and to skip the executions that do not contribute to the incorrect output. The tool offers two different options to indicate the contributing nodes. In the first option a contributing node is highlighted in green, whereas non-contributing node is displayed in yellow (a regular displaying of the current execution position). The programmer may step through the program execution using a step-wise execution mode, and each execution position is displayed in green or yellow color to indicate the contributing/non-contributing parts of program execution. In the second option, the tool opens a dialog box with a message indicating whether the node is contributing or not to the computation of a variable of interest. These options can be selected from the *Analysis* menu. Figure 8 shows that the statement " $i:=k+1$ " contributes to the computation of variable  $s$  at the `writeln`-statement.

### Partial Dynamic Slicing

Executable dynamic slices allow for the support of the notion of "partial dynamic slicing," i.e., dynamic slices determined on the execution subtrace rather than the whole execution trace. A partial dynamic slice is this part of a dynamic slice (computed for the slicing criterion  $C=(x,y^q)$ ) that affects the computation of the value of variable  $y$  at position  $q$  on the selected subtrace. This may be very useful when analyzing the execution of loops and procedures. Programmers may be interested in these loop iterations that affected the computation of  $y$  at  $q$  or this part of a procedure that affected the value of  $y$  at  $q$  during the current procedure call. For instance, after a dynamic slice for variable  $s$  at node 23 was computed, two partial dynamic slices may be derived that are shown in Figure 5. Figure 5a shows statements (a partial dynamic slice) that contributed to the computation of variable  $s$  at node 23 during the first iteration of the while-loop, whereas Figure 5b shows the statements that contributed to the computation of variable  $s$  during the second iteration of the while loop.

Influencing variables before execution of each action

7 <sup>1</sup>	<b>readln (n)</b>	
8 <sup>2</sup>	<b>readln (k)</b>	n
9 <sup>3</sup>	<b>i := 1</b>	k, n
10 <sup>4</sup>	<b>i &lt;= n</b>	k, n, i
11 <sup>5</sup>	<b>readln (a[i])</b>	k, n, i
12 <sup>6</sup>	<b>i := i + 1</b>	a[1], k, n, i
10 <sup>7</sup>	<b>i &lt;= n</b>	a[1], k, n, i
11 <sup>8</sup>	<b>readln (a[i])</b>	a[1], k, n, i
12 <sup>9</sup>	<b>i := i + 1</b>	a[1], a[2], k, n, i
10 <sup>10</sup>	<b>i &lt;= n</b>	a[1], a[2], k, n, i
11 <sup>11</sup>	<b>readln (a[i])</b>	a[1], a[2], k, n, i
12 <sup>12</sup>	<b>i := i + 1</b>	a[1], a[2], a[3], k, n, i
10 <sup>13</sup>	<b>i &lt;= n</b>	a[1], a[2], a[3], k, n, i
13 <sup>14</sup>	<b>max := a[1]</b>	a[1], a[2], a[3], k, n
14 <sup>15</sup>	<b>min := a[1]</b>	a[2], a[3], k, n, max
15 <sup>16</sup>	<b>s := a[1]</b>	a[2], a[3], k, n, max
16 <sup>17</sup>	<b>i := k+1</b>	a[2], a[3], k, n, max
17 <sup>18</sup>	<b>i ≤ n</b>	a[2], a[3], k, n, i, max
18 <sup>19</sup>	<b>a[i]&gt;0</b>	a[2], a[3], k, n, i, max
19 <sup>20</sup>	<b>Find_max(a[i],max);</b>	a[2], a[3], k, n, i, max
1 <sup>21</sup>	<b>procedure Find_max (s1: integer; var s2: integer);</b>	a[2], a[3], s2, k, n, i, max
2 <sup>22</sup>	<b>s1&gt;s2</b>	a[2], a[3], s2, s1, k, n, i, max
3 <sup>23</sup>	<b>s:=s1;</b>	a[2], a[3], s1, k, n, i
20 <sup>24</sup>	<b>Find_min(a[i],min);</b>	a[2], a[3], k, n, i, s
4 <sup>25</sup>	<b>procedure Find_min(x: integer; var y: integer);</b>	a[2], a[3], k, n, i, s
5 <sup>26</sup>	<b>x&lt;y</b>	a[2], a[3], k, n, i, s
21 <sup>27</sup>	<b>s := s + a[i]</b>	a[2], a[3], k, n, i, s
22 <sup>28</sup>	<b>i := i + k</b>	a[3], k, n, i
17 <sup>29</sup>	<b>i ≤ n</b>	a[3], k, n, i
18 <sup>30</sup>	<b>a[i]&gt;0</b>	a[3], k, n, i
22 <sup>31</sup>	<b>i := i + k</b>	k, n, i
17 <sup>32</sup>	<b>i ≤ n</b>	n, i
23 <sup>33</sup>	<b>write(max,' ',min,' ',s)</b>	

Contributing actions are shown in bold.

**Figure 4.** An execution trace of Figure 2 with contributing actions for the dynamic slice of Figure 3a and the influencing variables for  $s^{33}$  at node 23.

Partial dynamic slicing allows to observe the "dynamics" of program execution in a textual form that may significantly aid in debugging and understanding of program behavior. For example, after analyzing the partial slice of Figure 5a, a programmer may realize that procedure *Find\_max* influences the sum of array elements (represented by *s*) on the first iteration of the loop; however, this procedure should not have any influence on the final value of *s*. Based on this observation, the programmer may then analyze this procedure on the first iteration of the loop and localize the fault in this procedure.

In order to compute a partial dynamic slice the user has to indicate the beginning and the end of the subtrace of interest. Typically these subtraces relate to one or more loop iterations or a subtrace of a procedure call. In order to indicate the beginning of the subtrace, the user has to move the execution to the desired position (either by setting a breakpoint or by step-wise execution). When the program reaches the desired position, the user selects the *Partial Slice/Begin of Partial Slice* option from the menu. This indicates the beginning of the subtrace. The user then continues the execution to the desired execution position and selects the *Partial Slice/End of Partial Slice* option to indicate the end of the subtrace. The tool then automatically displays the partial slice by highlighting the corresponding statements. Figure 9 shows a partial slice for the first while-loop iteration and Figure 10 shows a partial slice for the second while-loop iteration for variable *s*.

```

1 procedure Find_max(s1:integer; var s2:integer);
  begin
2,3   if s1>s2 then s:=s1;
      end;

```

```

17  while i <= n do begin
18    if a[i]>0 then begin
19      Find_max(a[i],max);
21      s := s + a[i] ;
      end;
22    i := i + k ;
      end ;

```

**a.** Partial dynamic slice for the first iteration of the while-loop.

```

17  while i <= n do begin
18    if a[i]>0 then begin
      end;
22    i := i + k ;
      end ;

```

**b.** Partial dynamic slice for the second iteration of the while-loop.

**Figure 5.** Partial dynamic program slices with respect to variables *s* at node 23.

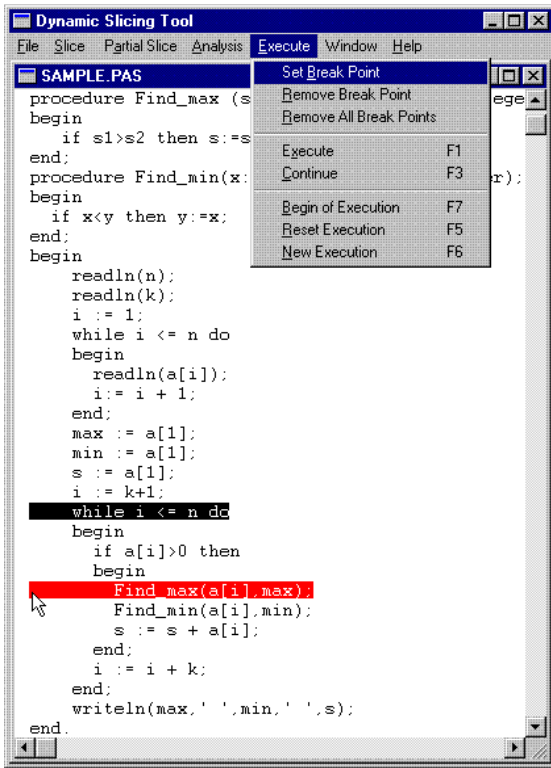


Figure 6. Setting a breakpoint.

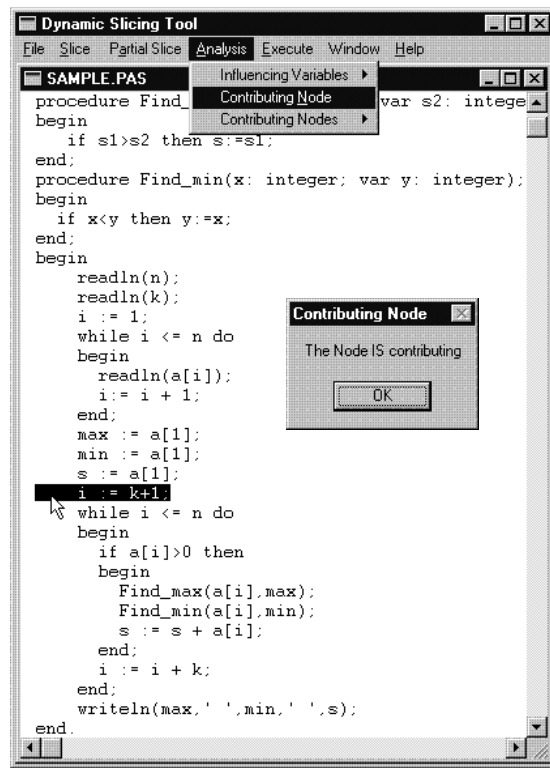


Figure 8. Displaying a contributing node.

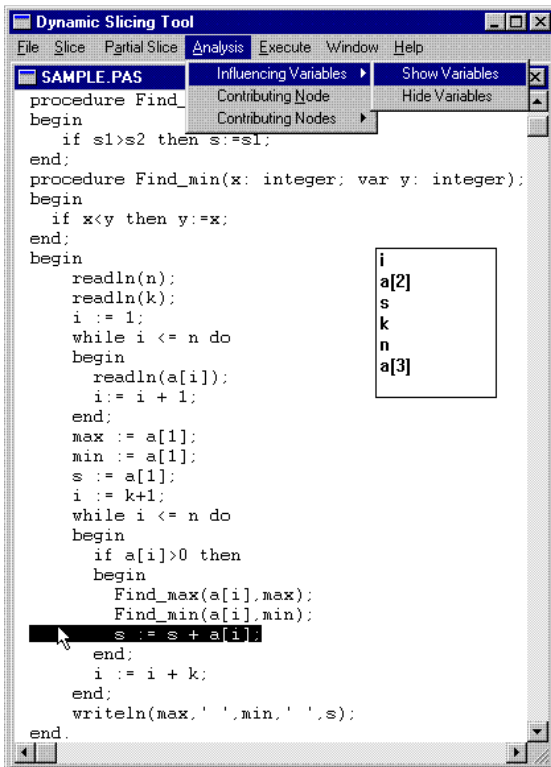


Figure 7. Displaying influencing variables.

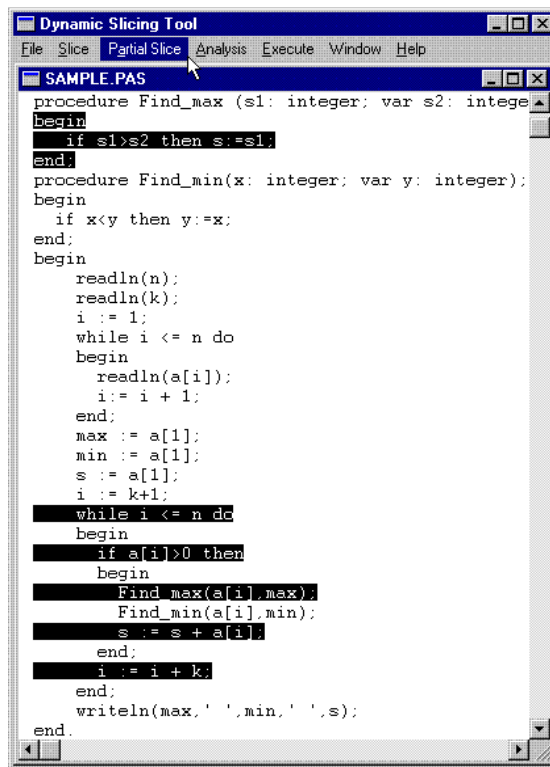


Figure 9. Partial slice for the first loop iteration.

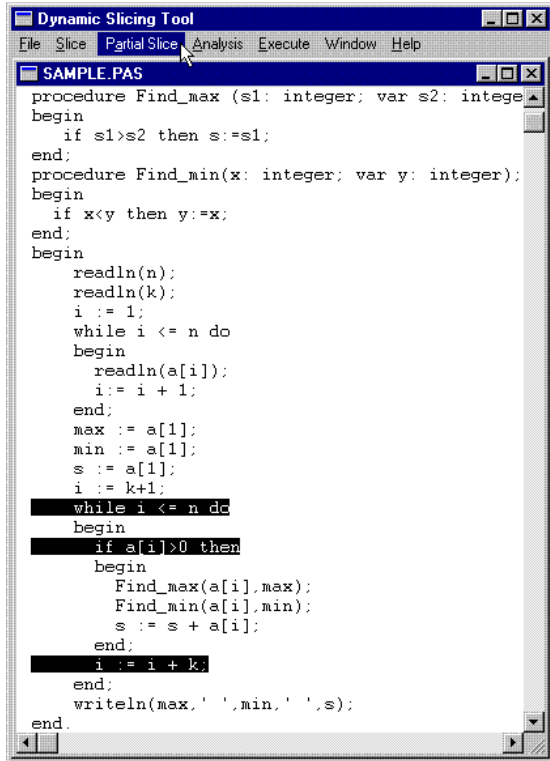
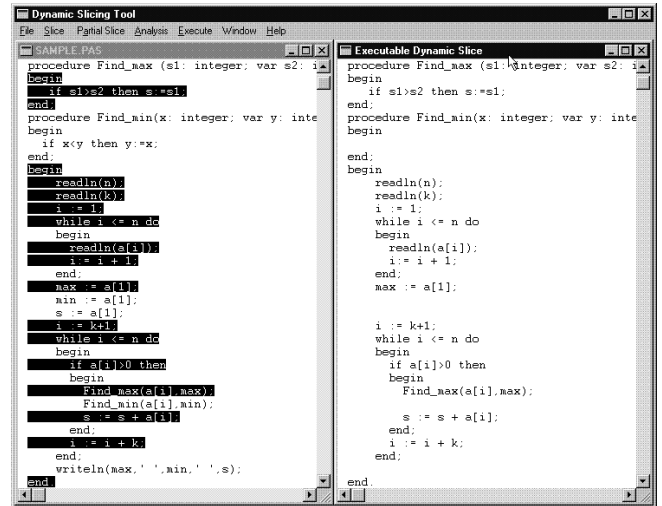


Figure 10. Partial slice for the second iteration.



a. Highlighting a slice.      b. A slice with removed statements.

Figure 11. Displaying dynamic slices

## 5. Conclusions

In this paper we have presented novel dynamic slicing features that are used to guide programmers in the process of debugging. The following features have been proposed: executable dynamic slices, partial dynamic slicing, influencing variables, and contributing nodes. These dynamic slicing related features have evolved during experimentation with our dynamic slicing tool. Our preliminary experience has shown that these features may be of a great help for programmers during the process of program debugging. However, more research and experimentation is needed in order to better understand the advantages and limitations of these features. So far we have performed experiments with programs up to 2000 lines in size, but we are planning to perform experiments on larger programs to determine the usability and scalability of the presented features and their trade-offs. The dynamic slicing features presented in this paper are not suitable for non-executable dynamic slices. Other methods of dynamic slice computation are not appropriate for the purpose of program understanding of program behavior, e.g., [1, 9], because some of these algorithms produce non-executable dynamic slices or the dynamic information is not derived during dynamic slice computation.

For programs with very long executions the forward approach of dynamic slice computation has been proposed in [15]. In the forward approach, a dynamic slice is computed during program execution and an execution trace is not recorded. We plan to develop methods that will not require major storing of the execution trace and will support the proposed dynamic slicing features. We are also planning to develop additional dynamic slicing features that may be useful in program debugging based on the experience with our dynamic slicing tool.

## References

- [1] H. Agrawal, J. Horgan, "Dynamic program slicing," SIGPLAN Notices, No. 6, 1990, pp. 246-256.
- [2] H. Agrawal, R. DeMillo, E. Spafford, "Debugging with dynamic slicing and backtracking," *Software Practice & Exp.*, vol. 23, No. 6, 1993, pp. 589-616.
- [3] J. Cheng, "Slicing concurrent programs," *Proc. of the 1-st Intern. Workshop on Automated and Alg. Debugging*, 1993, pp. 244-261.
- [4] E. Duesterwald, R. Gupta, M. L. Soffa, "Distributed slicing and partial re-execution for distributed programs," *Proc. 5th Workshop on Lang. and Compilers for Parallel Comp.*, 1992, pp. 329-337.
- [5] K. Gallagher, J. Lyle, "Using program slicing in software maintenance," *IEEE Tran. on Software Engineering*, vol. 17, No. 8, 1991, pp. 751-761.
- [6] R. Gopal, "Dynamic program slicing based on dependence relations," *Proc. of the Conf. on Software Maintenance*, 1991, pp. 191-200.
- [7] R. Gupta, M. Harrold, M. Soffa, "An approach to regression testing using slicing," *Conference on Software Maintenance*, 1992, pp. 299-308.
- [8] S. Horwitz, T. Reps, D. Binkley, "Interprocedural slicing using dependence graphs," *Trans. on Progr. Lang. and Systems*, vol. 12, No. 1, pp. 26-60, 1990.
- [9] M. Kamkar, *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*, Ph. D. Thesis, Linkoping University, 1993.
- [10] M. Kamkar, P. Fritzon, N. Shahmehri, "Interprocedural dynamic slicing applied to interprocedural data flow testing," *Conference on Software Maintenance*, 1993, pp. 386-395.
- [11] B. Korel, J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, No. 3, 1988, pp. 155-163.
- [12] B. Korel, "PELAS - Program Error Locating Assistant System," *IEEE Trans. on Software Eng.*, vol. SE-14, No. 9, 1988, pp. 1253-1260.
- [13] B. Korel, J. Laski, "Dynamic slicing in computer programs," *The Journal of Systems and Software*, vol. 13, No. 3, 1990, pp. 187-195.
- [14] B. Korel, R. Ferguson, "Dynamic slicing of distributed programs," *Applied Mathematics & Comp. Science J.*, vol. 2, No. 2, 1992, pp. 199-215.
- [15] B. Korel, S. Yalamanchili, "Forward Derivation of Dynamic Slices," *Proc. of the Intern. Symposium on Software Testing and Analysis*, 1994, pp. 66-79.
- [16] B. Korel, "Computation of dynamic slices for unstructured programs," *IEEE Transactions on Software Engineering*, vol. 23, No. 1, pp. 17-34, 1997.
- [17] J. Lyle, M. Weiser, "'Experiments on slicing-based debugging tools," *Proc. of the 1st Conf. on Empirical Studies of Programming*, 1986, pp. 187-197.
- [18] G. Venkatesh, "Experimental results from dynamic slicing of C programs," *Trans. on Progr. Lang. and Systems*, vol. 17, No. 2, pp. 197-216, 1995.
- [19] M. Weiser, "Programmers use slices when debugging," *CACM*, vol. 25, No. 7, 1982, pp. 446-452.
- [20] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.*, SE-10, No. 4, 1982, pp. 352-357.
- [21] L. White, H. Leung, "Regression testability," *IEEE Micro*, April 1992, pp. 81-85.
- [22] D. Binkley, K. Gallagher, "Program slicing," *Advances in Computers (to appear)*.