ELSEVIER

Information and Software Technology 40 (1998) 647–659

INFORMATION AND SOFTWARE TECHNOLOGY

Dynamic program slicing methods

Bogdan Korel*, Jurgen Rilling

Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA

Abstract

A dynamic program slice is that part of a program that ‘‘affects’’ the computation of a variable of interest during program execution on a specific program input. Dynamic program slicing refers to a collection of program slicing methods that are based on program execution and may significantly reduce the size of a program slice because run-time information, collected during program execution, is used to compute program slices. Dynamic program slicing was originally proposed only for program debugging, but its application has been extended to program comprehension, software testing, and software maintenance. Different types of dynamic program slices, together with algorithms to compute them, have been proposed in the literature. In this paper we present a classification of existing dynamic slicing methods and discuss the algorithms to compute dynamic slices. In the second part of the paper, we compare the existing methods of dynamic slice computation. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Classification; Comparison; Dynamic program slicing; Dynamic slicing algorithms

1. Introduction

A program slice consists of all statements in program *P* that may affect the value of variable *v* at some point *p* [34,35]. Slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance (e.g., [11,13,16,28,34]). As originally introduced, slicing (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’s behavior for specific program input, rather than that for the set of all inputs. This type of slicing is referred to as *dynamic* slicing [20]. Several different techniques for computation of dynamic slices have been proposed (e.g., [2,12,14,18,23]). By taking a particular program execution into account, dynamic slicing may significantly reduce the size of the slice as compared to static slicing. The concept of dynamic slicing has also been extended for distributed programs [7,9,22]. Dynamic program slicing is not only used in software debugging but also in software maintenance, program comprehension, and software testing [1,3,6,13,19,24,26,27,30–32,36]

Different methods of dynamic program slicing have been proposed in the literature (e.g., [2,12,18,20,23,25]). The reason for this diversity of slicing methods is the fact that different applications of slicing require different properties of slices. In this paper we present a classification of the existing dynamic slicing methods and discuss the existing algorithms to compute dynamic program slices. In addition, we compare the existing dynamic program slicing algorithms with respect to several aspects. In this paper we concentrate only on the dynamic slicing methods used for sequential programs. Two surveys of different slicing methods, including dynamic slicing, were presented in [6,32]. The major goal of this paper is to extend the discussion on the existing methods of dynamic slicing that were not addressed in these surveys. In addition, several new dynamic slicing methods have recently been developed that were not discussed in these surveys.

The remainder of this paper is organized as follows. Section 2 provides an overview of basic program slicing related terminology. Section 3 provides a general overview of dynamic program slicing. In Section 4 we describe the major existing dynamic slicing algorithms, and in Section 5 we present a comparison of the existing dynamic slicing methods and algorithms.

2. Basic terminology

Program slicing uses methods and terminology from the program dependence theory. A common cornerstone for most slicing algorithms is that programs are represented by a directed graph, which captures the notion of data dependence and control dependence. The program struc-

*Corresponding author. Tel.: +1 312 567 5145; fax: +1 312 567 5067; e-mail: korel@charlie.cns.iit.edu ture is represented by a flow graph $G = (N,A,s,e)$ where (1) N is a set of nodes, (2) A, a set of arcs, is a binary relation on N and (3) s and e are, respectively, unique entry and exit nodes. A node corresponds to an assignment statement, an input or output statement or the predicate of a conditional or a loop statement, in which case it is called a *test* node. A *path* from the entry node s to some node k, $k \in N$, is a sequence $\langle n_1,n_2,\ldots,n_q\rangle$ of nodes such that $n_1 = s$, $n_q = k$ and $(n_i,n_{i+1}) \in A$, for all n_i, $l \le i < q$.

A path that has actually been executed for some input will be referred to as an *execution trace*. For example, $T_x = \langle 1,2,3,4,5,6,7,8,9,10,13,14,6,15\rangle$ is the execution trace when the program in Fig. 1a is executed on the input $n = 3$, $a = (1,2,3)$: this execution trace is presented in Fig. 1b.

A path is feasible if there exists input data which causes the path to be traversed during program execution. A program trajectory has been defined as a feasible path that has actually been executed for some specific input. Notationally, an *execution trace* is an abstract list (sequence) whose elements are accessed by their position in it, e.g. for trace T_x in Fig. 1b, $T_x(6) = 6$, $T_x(13) = 6$.

Node Y at position p in T_x (e.g., $T_x(p) = Y$) will be written as Y^p and referred to as an action. Y^p is a *test* action if Y is a test node. By v^q we denote *variable v* at *position q*, i.e. variable (object) v before execution of node $T_x(q)$. The notion of an execution position is introduced in this paper for presentation purposes.

A *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. Let $U(Y^p)$ be a set of variables whose values are used in action Y^p and $D(Y^p)$ be a set of variables whose values are defined in Y^p. Sets $U(Y^p)$ and $D(Y^p)$ are determined during program execution, especially for array and pointer variables because it is possible to identify the specific array elements that are used or modified by the action during program execution. By *last definition* $LD(v^k)$ of variable v^k in execution trace T_x we mean action Y^p such that (1) $v \in D(Y^p)$ and (2) for all i, $p < i < k$, and all Z such that $T_x(i) = Z$, $v \in D(Z^i)$; in other words, action Y^p assigns a value to variable v and v is not modified between positions p and k. For example, 8^8 is a last definition of max^{14} in the execution trace of Fig. 1b.

3. Dynamic program slicing

A static program slice consists of all statements in program P that may affect the value of variable v at some point p [34]. As originally introduced, static slicing involves all possible program executions, i.e. the static slices preserve the computation of variable v at point p for all program inputs. However, in many situations one is interested in a slice that preserves the program's behavior for a specific program input, rather than that for all inputs. This type of slicing is referred to as *dynamic* slicing. The

```
1     input (n,a);
2     max := a[1];
3     min  := a[1];
4     i := 2;
5     s:= 0;
6     while i ≤ n do
       begin
7          if max < a[i] then
           begin
8             max := a[i];
9             s := max;
           end;
10         if min > a[i] then
           begin
11            min := a[i];
12            s := min;
           end;
13         output (s);
14         i := i +2;
      end;
15    output (max, min)
```

1^{1}	input (n,a);
2^{2}	max := a[1];
3^{3}	min := a[1];
4^{4}	i := 2;
5^{5}	s := 0;
6^{6}	i ≤ n
7^{7}	max < a[i];
8^{8}	max := a[i];
9^{9}	s := max;
10^{10}	min > a[i];
13^{11}	output(s);
14^{12}	i := i +2;
6^{13}	i ≤ n
15^{14}	output (max, min)

Fig. 1. (a) Sample program. (b) An execution trace of the sample program on input $n = 3$, $a = (1,2,3)$.

concept of dynamic program slicing was presented for the first time in [20]. By taking a particular program execution into account, dynamic slicing may significantly reduce the size of the slice as compared to static slicing because during program execution it is possible to determine different types of information, for example array elements or elements of dynamic data structures that are referenced during program execution, which leads to computation of smaller slices. Most of the existing dynamic slicing techniques have been proposed for sequential programs [2,12,18,20,23,25]. The concept of dynamic slicing has also been extended to distributed programs [7,9,22]. Two major types of dynamic slicing have been proposed in the literature: executable dynamic slicing [2,20,23,25] and non-executable dynamic slicing [2,12,18]. Informally, an executable dynamic slice is a slice that can be executed and it preserves a value of a variable of interest, whereas a non-executable slice contains statements that ‘‘influence’’ the variable of interest and in most cases cannot be executed.

3.1. Executable dynamic program slicing

A dynamic slice, as originally presented in [20], 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. 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. An *executable 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 the value of $y^{q'}$ in T_x'. A dynamic slice P' preserves the value of y for a given program input x. The goal is to find the slice with the minimal number of statements, but 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 value of a variable of interest. For example, Fig. 2 shows an executable dynamic slice of the program of Fig. 1a. Notice that when the original program of Fig. 1a and its dynamic slice of Fig. 2 is executed on input: $n = 3$, $a = (1,2,3)$, the value of variable *max* at statement 15 is the same.

3.2. Non-executable dynamic program slicing

For a given slicing criterion $C = (x,y^q)$, a non-executable dynamic slice contains statements that ‘‘influence’’ the variable of interest y^q during program execution on input x. There is no assumption about the execution of the dynamic slice. In most cases non-executable dynamic slices cannot be executed. Most of the existing methods of computation of dynamic slice use the notion of data and control dependencies to compute non-executable dynamic program slices. For example, Fig. 3 shows a sample non-executable dynamic slice of the program of Fig. 1a executed on input $n = 3$, $a = (1,2,3)$. Notice that when this dynamic slice is executed on input: $n = 3$, $a = (1,2,3)$, this dynamic slice does not terminate; as a result, it is a non-executable dynamic slice.

4. An overview of dynamic slicing algorithms

In this section we present an overview of the existing dynamic slicing algorithms [2,12,18,20,23,25]. In the first part we present algorithms for the computation of executable dynamic slices, and in the second part we present algorithms for the computation of non-executable dynamic slices.

```
1       input (n,a);
2       max := a[1];
4       i := 2;
6       while i ≤ n do
        begin
7           if max < a[i] then
            begin
8               max := a[i];
            end;
14          i := i +2;
        end;
15      output (max, min)
```

Fig. 2. An executable dynamic slice of the program of Fig. 1a executed on input $n = 3$, $a = (1,2,3)$ for variable *max* at statement 15.

```
1       input (n,a);
2       max := a[1]
4       i := 2;
6       while i ≤ n do
        begin
7           if max < a[i] then
            begin
8               max := a[i];
            end;
        end;
15      output (max, min)
```

Fig. 3. A non-executable dynamic slice of the program of Fig. 1a executed on input $n = 3$, $a = (1,2,3)$ with respect to variable *max* at statement 15.

4.1. Executable dynamic slicing algorithms

The existing algorithms for the computation of dynamic slices can be classified as trace based (execution trace) algorithms [2,12,20,25] and non-trace based algorithms [2,23]. In the trace based algorithms an execution trace of the program is first recorded and then the computation of a dynamic slice is performed on the recorded execution trace. Most of the existing methods of dynamic slice computation are based on the execution trace of a program. During recording of the execution trace different types of information are recorded depending on the algorithm. Typically, an execution trace contains information about the statement executed and variables defined or used at this statement (especially variables used or defined that cannot be determined by static analysis, e.g. array elements used or defined). In the non-trace based algorithms an execution trace is not recorded and a dynamic slice is computed during program execution.

4.1.1. Execution trace based algorithms

Most of the existing methods of dynamic slice computation are based on the recording of an execution trace. There are two major types of algorithms that compute dynamic slices from the execution trace: program dependence based algorithms and removable block based algorithms. Both types of algorithms use data dependencies to compute dynamic slices. However, program dependence based algorithms additionally use control dependencies to compute dynamic slices, whereas removable block based algorithms use removable blocks to compute dynamic slices.

4.1.1.1. Program dependence based algorithm

. The algorithm for dynamic slice computation presented in [20,21] uses the notion of data and control dependencies to compute dynamic program slices. Dynamic dependencies (e.g., [8,20]) between actions are captured by two types of dependencies: *data dependence* and *control dependence*. 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. Control dependence captures the dependence between test nodes and nodes that have been chosen to be executed by these test nodes. The control dependence is defined as [10]: let Y and Z be two nodes and (Y,X) be a branch of Y. Node Z *postdominates* node Y iff Z is on every path from Y to the exit node e. Node Z postdominates branch (Y,X) iff Z is on every path from Y to the program exit node e through branch (Y,X). Z is *control dependent* on Y iff Z postdominates one of the branches of Y and Z does not postdominate Y. For example, in the program of Fig. 1a, node 14 is control dependent on node 6 because node 14 postdominates branch (6,7) and node 14 does not postdominate node 6.

In the dynamic slice computation after the execution trace of the program is first recorded, the dynamic slice algorithm traverses the execution trace to determine data and control dependencies between actions that occur during program execution. These dependencies are then used to compute a dynamic slice. In this algorithm, after setting all actions as unmarked and not visited, the last definition of y^q is marked. In the first step, a marked and not visited action is selected and set as visited. All last definitions of all variables used in the selected action X^k are identified and marked (this step corresponds to finding data dependencies between actions). In the second step, all actions for which there exists a control dependence and the selected action are marked. In the third step, all actions of node X in the trace are marked. These steps are repeated until all marked actions are visited. A more detailed explanation of this algorithm can be found in [20].

4.1.1.2. Removable block based algorithm

. One of the major limitations of the program dependence based algorithms is that they may compute dynamic slices for unstructured programs incorrectly. For example, for the program of Fig. 4a these algorithms incorrectly compute a dynamic slice for the execution trace of Fig. 4b and variable v at 11 because of the limitations of control dependencies. Dynamic slicing algorithms based on the notion of removable blocks were presented in [24,25]. These algorithms can correctly compute dynamic slices for unstructured programs. Before presenting this approach informally, we introduce the notion of a *removable block*. A removable block (or a block) is the smallest part of program text that can be removed during slice computation without violating the syntactical correctness of the program. For structured programs a block can be an assignment statement, an if-then statement, a while-statement, etc.

For unstructured programs there are additional blocks related to jump statements, e.g. a goto-statement or a break-statement. Notice that test nodes (predicates of conditional statements) are not removable and, therefore, they are not considered as blocks. In Fig. 5 all blocks for the program of Fig. 4a are shown in rectangles. Each block B has a regular entry to B and a regular exit from B. In unstructured programs, because of jump statements, execution may directly enter a block without going through its regular entry. Similarly, execution can exit a block without going through its regular exit. For example, in block B_4 of Fig. 6 the execution can exit this block without going through its regular exit. Notice that in a goto, break, or continue block (statement) the execution never reaches the regular exit in these blocks.

Let $C=(x,y^q)$ be a slicing criterion. In the program dependence based algorithms of dynamic slice computation the major goal is to identify those actions in the execution trace that contribute to the computation of the value of variable y^q by identifying data and control dependencies in the execution trace. However, in dynamic

```
var
v,x,y,z: integer;

1       input(v,x,y);
2       L:;
3       z := x;
4,5     if v>x then goto L1;
6       v := v+x;
7       z := v+y;
8,9     if v>y then goto L;
10      L1:;
11      output(v,z);
```

```
1^1     input(v,x,y);
2^2     L:
3^3     z := x
4^4     v>x
6^5     v := v+x
7^6     z := v+y
8^7     v>y
9^8     goto L
2^9     L:
3^10    z := x
4^11    v>x
5^12    goto L1
10^13   L1:
11^14   output(v,z)
```

Fig. 4. (a) A sample unstructured program. (b) An execution trace of the program of (a); input: $v = x = y = 2$.

slice computation it is not only important to identify such contributing actions but also, at the same time, to identify actions that do not contribute to the computation of y^q. The more such "non-contributing" actions can be identified, the smaller dynamic slices may be computed. The algorithm employs the notion of removable blocks in finding dynamic program slices. In this algorithm only data dependencies are used to identify contributing computations (control dependencies are not used to identify contributing computations), whereas removable blocks and block traces are used to identify non-contributing computations. In this algorithm, after setting all actions as unmarked and not visited, the last definition of y^q is marked. In the first step, a marked and not visited action is selected and set as visited. All last definitions of all variables used in the selected action X^k are identified and marked (this corresponds to finding data dependencies between actions and marking them as contributing) and at the same time the corresponding blocks, to which the actions belong, are marked, i.e. they are identified as part of a dynamic slice.

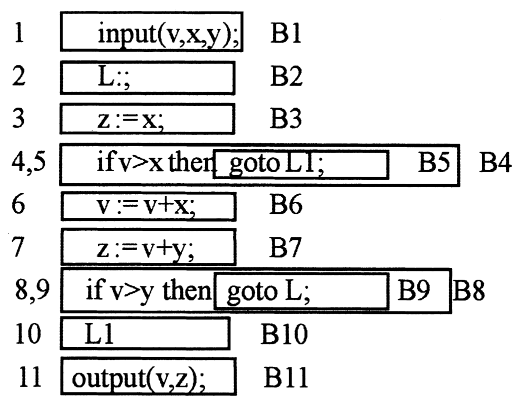

```
1     input(v,x,y);              B1
2     L:;                        B2
3     z:=x;                      B3
4,5   if v>x then  goto L1;  B5  B4
6     v:=v+x;                    B6
7     z:=v+y;                    B7
8,9   if v>y then  goto L;   B9  B8
10    L1                         B10
11    output(v,z);               B11
```

Fig. 5. A program of Fig. 4a with removable blocks.

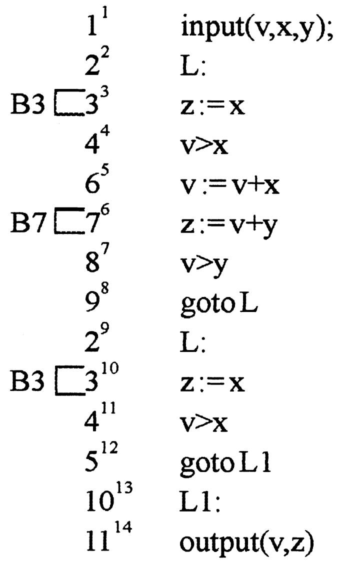

```
     1^1    input(v,x,y);
     2^2    L:
B3   3^3    z:=x
     4^4    v>x
     6^5    v:=v+x
B7   7^6    z:=v+y
     8^7    v>y
     9^8    goto L
     2^9    L:
B3   3^10   z:=x
     4^11   v>x
     5^12   goto L1
     10^13  L1:
     11^14  output(v,z)
```

Fig. 6. Execution trace with "removable blocks".

This step is repeated until all marked (contributing) actions are visited. In the second step, the algorithm identifies non-contributing actions that are part of "regular-entry/regular exit" executions of all blocks that do not belong to the slice.

In the third step, all actions that are not identified as contributing and non-contributing are marked as contributing (marked) and the algorithm continues with the first step. These three steps are repeated until all actions are identified as contributing and non-contributing. A dynamic slice consists of all marked blocks. A more detailed explanation of this algorithm can be found in [25]. When the dynamic slicing algorithm is applied for the program of Fig. 4a and its execution trace of Fig. 4b for variable *v* at statement 11, the following dynamic slice is computed:

```
1      input(v,x,y);
2      L:;
4,5    if v>x then goto L1;
6      v := v+x;
8,9    if v>y then goto L;
10     L1:;
11     output(v,z);
```

Removable blocks of this program are shown in Fig. 5.

4.1.1.3. An extended approach for computation of dynamic slices

. In [25] an extended version of the removable block based algorithm has been proposed. Using blocks as the only components that may be removed during dynamic slice computation may not always lead to the computation of the smallest dynamic slices. The extended approach is based on the observation that a sequence of consecutive blocks may be grouped together to form a larger "logical" component that can be removed from a dynamic slice. Such a "logical" component may be treated as a more general notion of a block. This may lead to computation of significantly smaller dynamic slices. For example, when the extended dynamic slicing algorithm is applied for the program of Fig. 4a and its execution trace of Fig. 4b for variable *v* at statement 11, the following dynamic slice is computed (Fig. 7):

```
1 input(v,x,y);
6 v := v+x;
11 output(v);
```

4.1.2. Non-execution trace based algorithms

In non-trace based algorithms an execution trace is not recorded and a dynamic slice is computed during program execution. There are two major algorithms of this type: the forward algorithm [23,17] and the static program dependence based algorithm [2]. The main motivation for the development of non-execution trace based algorithms is to overcome one of the limitations of the execution trace based algorithms, that is the necessity to record the execution trace during program execution.

4.1.2.1. The forward algorithm

. Forward computation of dynamic slices [23,17] creates dynamic slices during program execution without recording of the execution trace. The idea of finding dynamic program slices is based on the notion of removable blocks

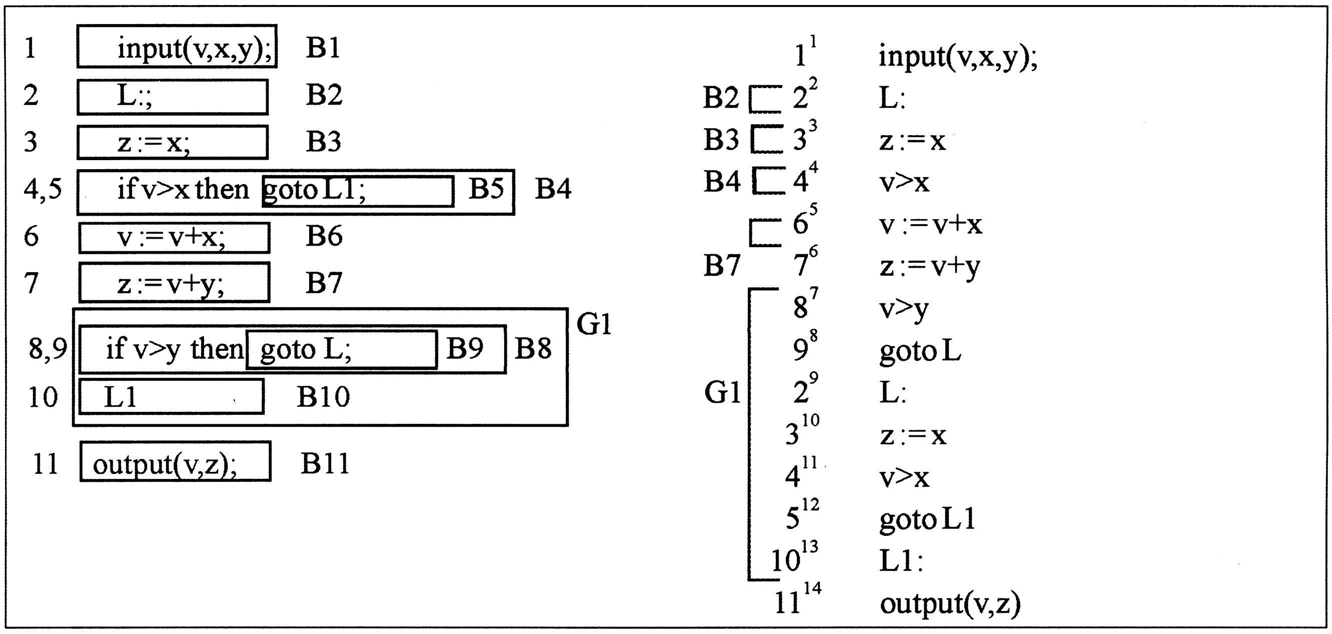


Fig. 7. (a) A program of Fig. 4a with a grouped removable block G1. (b) Execution trace with a "grouped" block G1.

presented in the previous section (Fig. 8). The forward algorithm computes a dynamic slice for every program variable during program execution on input *x*. The forward algorithm starts from the first statement in the program and proceeds "forward" with program execution and at the same time performs the computation of the dynamic slices for program variables along with the program execution. The underlying idea of the forward approach of dynamic slice computation is that during program execution on each regular exit from a block the algorithm determines whether the executed block should be included in a dynamic slice or not.

The following are two sample conditions used by the forward algorithm to compute dynamic slices during program execution.

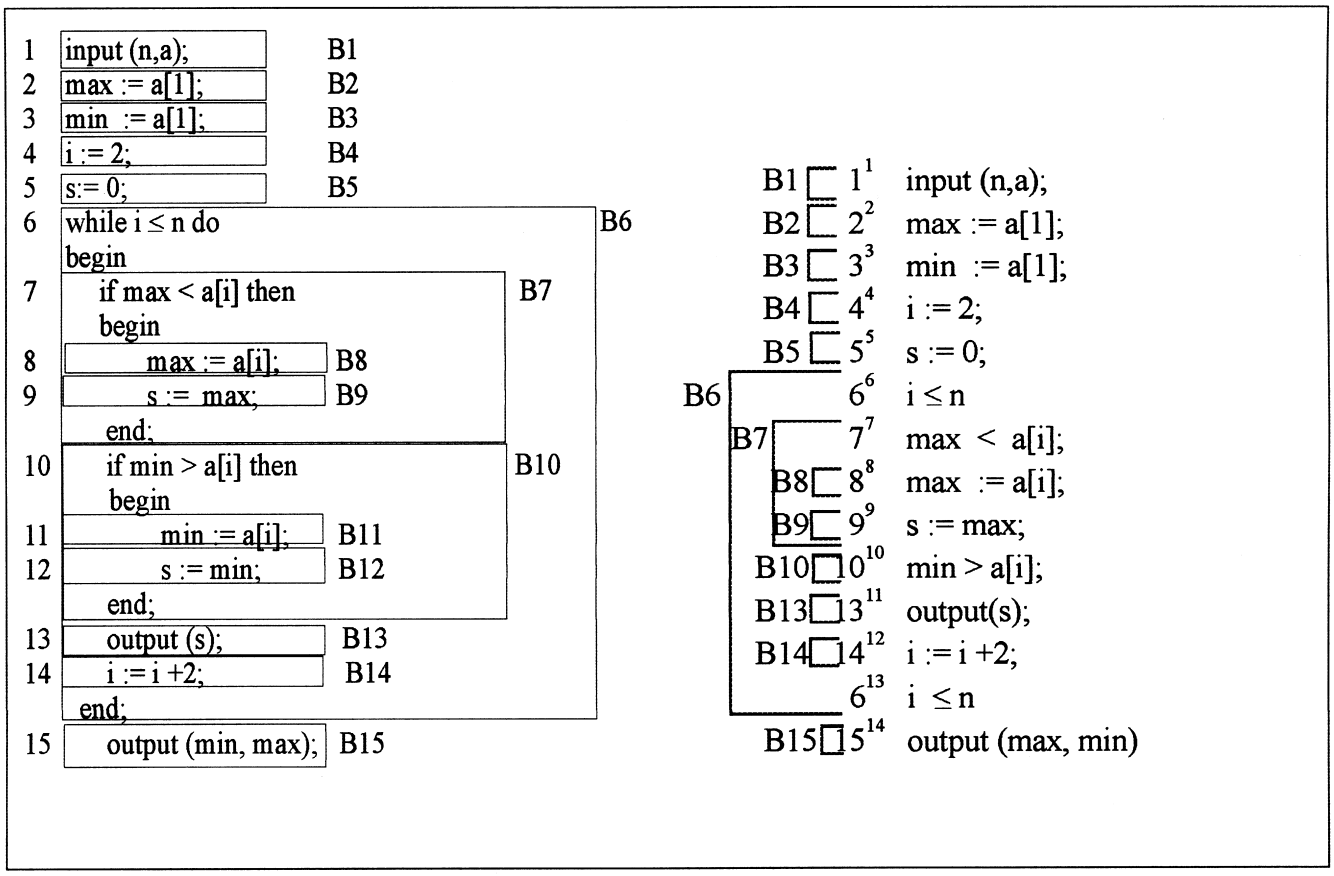
1. If the value of variable *v* has not been modified during execution of block *B*, and none of the nodes executed inside of block *B* belong to the dynamic slice of variable *v*, the block is not included in the dynamic slice for variable *v* at the exit from *B*.
2. A dynamic slice for a variable modified by an assignment is a union of dynamic slices of all variables used at the assignment statement.

Fig. 9 shows in detail how the forward algorithm computes slices for all variables defined in the program of Fig. 8a and its execution trace of Fig. 1b.

Huynh and Song [17] recently extended the original forward approach for programs with structured jump statements, e.g. *breaks* and *continues*.

4.1.2.2. The static program dependence based algorithm

. Agrawal and Horgan present four different approaches to dynamic slice computation [2]. We consider here the first two algorithms presented, which are based on a derived program dependence graph [29,14,15]. The program dependency graph captures the existing data and control dependencies of the program. During program execution the edges of the program dependence graph that occurred during a program execution are marked. After the program execution, the algorithm traverses the program dependence graph only along the marked edges to find the dynamic slice. The first algorithm allows an edge to be marked without actually influencing the computation of the function of interest. The second algorithm marks the executed edges. This produces slices which do not include statements that were executed but have no influence on the variables in the slice. Fig. 10 shows a program dependence graph of the program of Fig. 1a with highlighted edges


```
1   input (n,a);              B1
2   max := a[1];              B2
3   min  := a[1];             B3
4   i := 2;                   B4
5   s:= 0;                    B5
6   while i ≤ n do                          B6
    begin
7     if max < a[i] then               B7
      begin
8        max := a[i];         B8
9        s := max;            B9
      end;
10    if min > a[i] then               B10
      begin
11       min := a[i];         B11
12       s := min;            B12
      end;
13    output (s);             B13
14    i := i +2;              B14
    end;
15  output (min, max);        B15
```

```
      B1   1^1    input (n,a);
      B2   2^2    max := a[1];
      B3   3^3    min  := a[1];
      B4   4^4    i := 2;
      B5   5^5    s := 0;
B6         6^6    i ≤ n
      B7   7^7    max  <  a[i];
      B8   8^8    max  := a[i];
      B9   9^9    s := max;
      B10  10^10  min > a[i];
      B13  13^11  output(s);
      B14  14^12  i := i +2;
           6^13   i  ≤ n
      B15  15^14  output (max, min)
```

Fig. 8. (a) Sample program with removable blocks. (b) Execution trace with removable blocks.

Node	Slice	max	min	s	i
1^{1}	input (n,a);	1	{}	{}	{}
2^{2}	max := a[1];	{}	{}	{}	{}
3^{3}	min := a[1];	{1,2}	{}	{}	{}
4^{4}	i := 2;	{1,2}	{1,3}	{}	{}
5^{5}	s := 0;	{1,2}	{1,3}	{}	{4}
6^{6}	$i \leq n$	{1,2}	{1,3}	{5}	{4}
7^{7}	max < a[i]	{1,2,4,6}	{1,3,4,6}	{1,4,5,6}	{1,4,6}
8^{8}	max := a[i];	{1,2,4,6,7}	{1,2,3,4,6,7}	{1,2,4,5,6,7}	{1,2,4,6,7}
9^{9}	s := max;	{1,2,4,6,7,8}	{1,2,3,4,6,7}	{1,2,4,5,6,7}	{1,2,4,6,7}
10^{10}	min > a[i];	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6}
13^{11}	output(s);	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6}
14^{12}	i := i +2;	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6}
6^{13}	$i \leq n$	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6,14}
15^{14}	output (min, max)	{1,2,4,6,7,8,14}	{1,3}	{1,2,4,5,6,7,8,9,14}	{1,4,6,14}

Fig. 9. Execution trace of the program from Fig. 8a for input $n = 3$, $a = (1,2,3)$ together with dynamic slices computed during program execution for four variables.

(data and control dependencies) that occur during program execution shown in Fig. 1b. A dynamic slice for variable *max* at statements 15 is computed by tracing backwards the highlighted edges starting with edge (8,15) where 8 is the last definition of variable *max*. The resulting dynamic slice is shown in Fig. 2.

4.2. Non-executable dynamic slicing algorithms

For a given slicing criterion $C = (x,y^q)$, a non-executable dynamic slice contains statements that "influence" the variable of interest y^q during program execution for an input *x*. There is no assumption about the execution of the

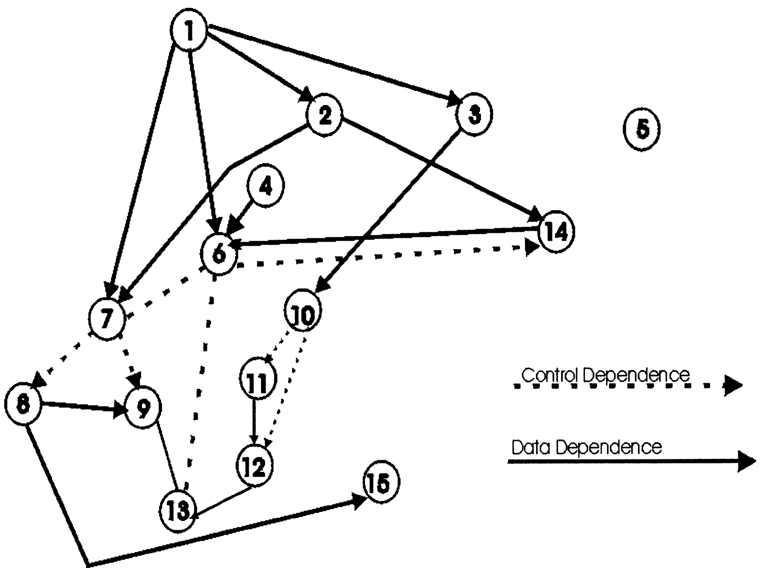


Fig. 10. Program dependence graph of the program of Fig. 1a.

dynamic slice. In most cases non-executable dynamic slices cannot be executed. Most of the existing methods of computation of a dynamic slice use the notion of data and control dependencies to compute non-executable dynamic program slices. Although, in general, these algorithms may compute ‘‘smaller’’ slices than the executable dynamic slices; the major problem is with the vague notion of program dependencies used to compute these slices.

4.2.1. Agrawal and Horgan dynamic slicing algorithms

Agrawal and Horgan proposed two methods of computation of non-executable dynamic program slices [2]. Both algorithms use data and control dependencies to compute the slices. The first algorithm is based on the property that different occurrences of the same statement may have different dependencies and it is possible that one occurrence contributes to the slice while another does not. In this approach they create a separate node for each occurrence of a statement in the execution history, with outgoing dependence edges to only those statements on which this specific occurrence of the statement occurrence is dependent. The dependency graph for this algorithm is shown in Fig. 11. Each node in the new dependence graph, called a dynamic dependence graph, will have at the maximum one outgoing edge for each variable used at the statement. In general, the problem with the dynamic dependence graph is that the number of nodes which have to be analyzed is unbounded. The number is equal to the number of statements in the execution trace. The dynamic slice computed by this algorithm for the sample program of Fig. 1a for the input ($n = 3$, $a = 1,2,3$) is shown in Fig. 3. Notice that the following statements are no longer included in the dynamic slice (3,5,9,10,11,12,13,14), since there exists no dependency to the variable *max* at node 15. By not including node 14 in the slice, the algorithm computes a slice that is non-executable (non-terminating program execution).

Another approach by Agrawal and Horgan presented in [2], called Reduced Dynamic Dependence Graph (RDDG), does not require the recording of the execution trace and restricts the number of nodes in a dynamic dependence graph. For the Reduced Dynamic Dependence Graph the size of the dynamic dependency graph is no longer a function of the number of corresponding nodes in the execution trace. This approach creates a new node in the dependence graph only if another node with the same transitive dependencies does not already exist. The problem with this approach is that in some cases it only creates a set of statements instead of an executable slice. The slice for the variable *max* at node 15 is the same as that computed by the third algorithm (shown in Fig. 3).

4.2.2. Gopal's dynamic slicing algorithm

Gopal's dynamic slicing algorithm [12] is based on a dynamic version of Bergeretti and Carre's information-flow relations [5]. For empty statements, assignments and statement sequences Gopal's dependence relations are exactly the same as for the static case. The information-flow relations for a conditional statement are derived from the statement itself, and from the statements that constitute its branches. Gopal defines specific rules for different kinds of statements, e.g. assignment, conditional statements and sequences of statements. For dynamic dependence relations, however, only the dependencies that arise during the actual program execution are taken into account. Under certain loop conditions, Gopal's algorithm may compute non-executable (non-terminating) slices.

4.2.3. Kamkar's dynamic slicing algorithm

Kamkar presents an algorithm [18] which is based on graph reachability in dependence graphs. This algorithm adapts the interprocedural slicing presented in [19], which is primarily concerned with procedure level slices. During

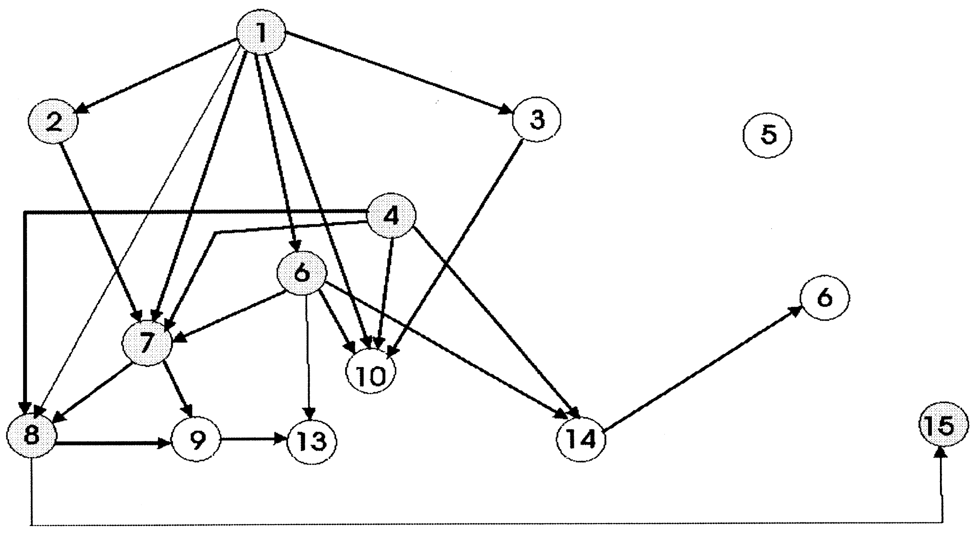


Fig. 11. Dynamic Dependence Graph (DDG) for the sample program from Fig. 1a.

program execution a dynamic dependence summary graph is constructed. The algorithm requires that a program is executed and its execution recorded to allow for the generation of an execution tree with dependence information. The slice is computed by traversing the execution tree backwards with respect to the dependencies. The algorithm by Kamkar can also be used for interprocedural slicing for structured programs with procedure calls. The slice calculated by Kamkar's algorithm is a non-executable slice.

5. Comparison of the existing dynamic slicing algorithms

This section compares the existing dynamic slicing methods and algorithms. The motivation for this comparison is to identify limitations and strengths of dynamic slicing methods and algorithms for sequential programs. A comparison of the dynamic slicing algorithms has been reported in [32]. In this section we discuss several important issues not discussed in [32] in order to have a better understanding of the existing methods of dynamic program slicing. In addition, we identify several dynamic slicing related research problems.

5.1. Executable versus non-executable dynamic slicing

In this section we contrast two types of dynamic program slicing methods: executable dynamic slicing and non-executable dynamic slicing. Executable dynamic slicing requires a dynamic slice to be executable and compute the same value for the variable of interest as in the original program for the same program input. Since the goal of executable dynamic slicing is very well defined, it is easy to understand the meaning of executable dynamic slices by programmers. In addition, it is possible to determine the correctness of the algorithms computing dynamic slices (it has been shown that some of the dynamic slicing algorithms may compute incorrect dynamic slices, e.g. for unstructured programs). Finally, since the goal for algorithms computing executable dynamic slices is precisely defined, it is also possible to compare them with respect to slice size, speed, memory requirements, etc.

The major motivation for the introduction of non-executable slicing was to compute smaller dynamic slices than executable dynamic slices because it is easier for programmers to localize faulty statements during debugging when slices are smaller. In general, a non-executable slice contains statements that ‘‘influence’’ the variable of interest, and programmers expect that a slice contains all faulty statement(s). Otherwise, a slice may be misleading to programmers in the process of debugging. The reduction of the slice size is a very important and challenging research issue not only for dynamic slicing but also for static slicing. Much research has recently been carried out to reduce the size of static and dynamic slices. One of the problems of the application of static and dynamic program slicing in program debugging is that program slices may ‘‘miss’’ faulty statements, i.e. faulty statements are not included in the slice. The sample faulty program of Fig. 12a demonstrates this problem. In this program statement 8 is a faulty statement. On most program inputs this program will compute an incorrect value for variable *max*. Fig. 12b shows a program slice (static or dynamic) for variable *max* at statement 15. Unfortunately, the computed slice does not contain the faulty statement. This example clearly illustrates that any attempt (method) to reduce the size of a slice has to take into account the fact that any reduction of the slice size may ‘‘slice out’’ more faulty statements than other slicing methods. Unfortunately, non-executable dynamic slicing may slice out more faulty statements than executable dynamic slicing. Consider the sample program of Fig. 1a and its execution trace of Fig. 1b. In this program, statement 14 ($i := i + 2$) is a faulty statement (in the correct program this statement should increment the value of i by one, i.e. $i := i + 1$). When the program is executed on input $n = 3$, $a = (1,2,3)$ it computes an incorrect value of variable $max = 2$ (correctly, $max = 3$). An executable dynamic slice for variable *max* is shown in Fig. 2 (this slice contains the faulty statement). However, the non-executable dynamic slice for variable *max* shown in Fig. 3 does not contain the faulty statement. For program debugging the challenge is not necessarily to minimize the size of the slice at any cost but rather to minimize the size of the slice without the elimination of faulty program statements.

Another major problem with non-executable dynamic slicing is that no precise goal for the computation of dynamic slicing is defined. As a result, it is not clear for programmers what exactly a non-executable slice represents, especially when it is not used for the purpose of program debugging. Since the goal for non-executable dynamic slicing is not well defined, it is not possible to determine the correctness of the existing algorithms computing non-executable slices. Clearly, when a non-executable slice is computed using one of the existing algorithms, there is no criterion that can be used to determine whether the computed slice is a correct one or not. A variety of different non-executable dynamic slicing algorithms have been developed, making it almost impossible to compare them with each other because each of the algorithms may compute slices with different properties.

Finally, the non-executable dynamic slicing methods use the notion of ‘‘dependence’’ to compute non-executable dynamic slices. Most algorithms use data and control dependencies for this purpose. It is believed that these dependencies are ‘‘sufficient’’ to compute dynamic slices. However, it is not clear whether this is actually the case. For example, it was a generally accepted opinion that the notion of control dependence is a well-defined concept. Unfortunately, it has recently been shown that using

```
1       input (n,a);
2       max := a[1];
3       min  := a[1];
4       i := 2;
5       s:= 0;
6       while i ≤ n do
        begin
7           if max < a[i] then
            begin
8               a[1]:=max;  /* correctly: max:=a[i]
9               s := max;
            end;
10          if min > a[i] then
            begin
11              min := a[i];
12              s := min;
            end;
13          output (s);
14          i := i +1;
        end;
15      output (max, min)
```

```
1       input (n,a);
2       max := a[1];
15      output (max, min)
```

Fig. 12. (a) Sample program. (b) A slice (dynamic or static) for variable *max* at 15.

control dependencies as defined in [10] may actually lead to incorrect slices for unstructured programs [4,25]. When control dependencies are used to compute non-executable dynamic slices for unstructured programs, it is unclear whether these slices are computed correctly or not.

5.2. Correctness

The correctness of most of the existing dynamic slicing algorithms has not been proven. As mentioned in the previous section, the correctness of algorithms for non-executable dynamic slicing does not make sense because the goal for these slices is not clearly defined. On the other hand, it is possible to prove the correctness of the algorithms for executable slicing. The first dynamic slicing algorithms [2,20] are dependence based algorithms and are limited for structured programs. However, the correctness of these algorithms has not been proven. It was believed that it is sufficient to take into account data and control dependencies to compute correct dynamic slices. These algorithms in most cases compute correct dynamic slices, but there are some cases for which these algorithms may actually compute incorrect dynamic slices even for structured programs. This is caused by the weakness of control dependencies. Consider the program of Fig. 13 which is executed on the program input $n = 1$, $a = (5)$. Notice that the program executes the repeat loop only once. The incorrect dynamic slice shown in Fig. 14 is computed by dynamic slicing algorithms [2,20] for variable *s* at statement 7. Notice that statement 5 is not included in the slice. The major reason for the incorrect computation of this dynamic slice is that control dependence between test node 6 and statement 5 never "occurs" during program execution. This of course can be easily "fixed" in these dynamic

```
1       input (n,a);
2       i:=1;
3       repeat
4           s:= s + a[i];
5           i := i+1;
6       until i ≥ n;
7       output(s)
```

Fig. 13. A sample program.

```
1       input (n,a);
2       i:=1;
3       repeat
4           s:= s + a[i];
6       until i ≥ n;
7       output(s)
```

Fig. 14. A dynamic slice of the program of Fig. 13 for variable *s* at 7.

slicing algorithms. But since the correctness of these algorithms has never been proven, it is not clear whether there are more such cases for which these dependence based algorithms may compute incorrect dynamic slices.

The correctness of the forward algorithm [23] for structured programs has been proven and this algorithm correctly computes the dynamic slice for the program of Fig. 13 (this algorithm is limited to structured programs). In [25] it has been shown that the dependence based dynamic slicing algorithms will in most cases compute incorrect dynamic slices for unstructured programs. The removable block based algorithms have been proposed to compute dynamic slices for arbitrary programs, and their correctness has been proven [25].

5.3. Size of dynamic slices

Since non-executable slices are not well defined, any comparison of different algorithms with respect to the size of slices for executable and non-executable slices is meaningless because these slices have different properties. Moreover, any comparison with respect to slice sizes between non-executable slicing algorithms may also be meaningless. However, algorithms for executable dynamic slicing can be compared with respect to the computed slices.

It is believed that all algorithms for executable slices compute identical dynamic slices (if computed correctly) for structured programs. However, this has never been proven. One of the research challenges is to show that, actually, these algorithms compute the same dynamic slices. The situation is quite different for dynamic slicing algorithms for unstructured programs. Currently, there are only two dynamic slicing algorithms for unstructured programs [25]. It has been shown that the extended algorithm can compute significantly smaller dynamic slices. One of the major research challenges is to develop new algorithms that will reduce the size of dynamic slices preserving, however, the properties of the executable slices. If some of these properties are relaxed, one has to carefully investigate the consequences of this relaxation on the dynamic slicing and its application.

5.4. Space and time complexity

Space complexity of different dynamic slicing algorithms was discussed in [32]. Therefore, in this section we provide a brief summary of the space complexity of dynamic slicing algorithms. In general, execution trace based algorithms [20,25] have to store the execution trace and therefore their space complexity depends on the size of a particular execution. These methods are appropriate for relatively short executions. Non-execution based algorithms [2,23] do not require storing of the execution trace and their space complexity is bounded. The space complexity of these algorithms is a function of the size of the program (number of statements) and the number of variables.

Time complexity of dynamic slicing algorithms has not been studied analytically and experimentally. It has been shown that the size of a dynamic slice may vary significantly with respect to the variables for which dynamic slices are computed and with respect to different program inputs [33]. Similarly, our experience has shown that the time to compute dynamic slices may vary significantly with respect to the variables and program inputs. One of the major research challenges is to compare the existing slicing algorithms analytically and experimentally. We have performed some preliminary experiments, in which we have compared the time complexity of execution based algorithms with non-execution based algorithms. The initial results indicate that the execution trace based algorithms outperform the forward algorithms for computation of individual dynamic slices. On the other hand, the forward approach outperformed the execution trace algorithms in situations where a large number of dynamic slices were supposed to be computed for the same execution. This is a major research challenge to perform experiments in order to compare not only the space complexity of the slicing algorithms but also their time complexity.

6. Conclusions

Dynamic program slicing refers to a collection of program slicing methods that are based on program execution and may significantly reduce the size of a program slice because run-time information, collected during program execution, is used to compute program slices. Dynamic program slicing was originality proposed only for program debugging, but its application has been extended to program comprehension, software testing, and software maintenance. Different types of dynamic program slices, together with algorithms to compute them, have been proposed in the literature. In this paper we have presented a classification of existing dynamic slicing methods and discuss the algorithms to compute dynamic slices. In the second part of the paper, we compare the existing methods of dynamic slice computation. In particular, we have compared two major types of dynamic program slicing: executable program slicing and non-executable program slicing. In addition, we have discussed the issues of correctness of the existing dynamic slicing algorithms, the slice size, etc. The motivation for this comparison is to identify limitations and strengths of dynamic slicing methods and algorithms for sequential programs, and identify dynamic slicing related research problems.

Acknowledgements

This research was partially supported by NSF Research Initiation Award grant CCR-9308895.

References

[1] H. Agrawal, Towards automatic debugging of computer programs, technical report SERTC-TR-40-P, Purdue University, 1989.
[2] H. Agrawal, J. Horgan, Dynamic program slicing, in: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990, pp. 246–256 (SIGPLAN Notices 25(6))
[3] H. Agrawal, R. DeMillo, E. Spafford, Debugging with dynamic slicing and backtracking, Software—Practice and Experience 23(6) (1993) 589–616.
[4] T. Ball, S. Horwitz, Slicing programs with arbitrary control-flow, in: Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, Lecture Notes in Computer Science, Vol. 749, Springer, Berlin, 1993, pp. 206–222.
[5] J-F. Bergeretti, X. Carre, Information-flow and data-flow analysis of while-programs, ACM Transactions on Programming Languages and Systems 7(1) (1985) 37–61.
[6] D. Binkley, K. Gallagher, Program Slicing, Advances in Computers, Vol. 43, Academic Press, New York, 1996, pp. 1–52.
[7] J. Cheng, Slicing concurrent programs—a graph-approach, in: P. Fritzson (Ed.), Proceedings of the First International Workshop on Automated and Algorithmic Debugging, Lecture Notes in Computer Science, Vol. 749, Springer, Berlin, 1993, pp. 232–245.
[8] J.-D. Choi, B. Miller, R. Netzer, Techniques for debugging parallel programs with flowback analysis, ACM Transactions on Programming Languages and Systems 13(4) (1991) 491–530.
[9] E. Duesterwald, R. Gupta, M. Soffa, Distributed slicing and partial reexecution for distributed programs, in: Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, 1992, pp. 329–337.
[10] K. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9(5) (1987) 319–349.
[11] K. Gallagher, J. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17(8) (1991) 751–761.
[12] R. Gopal, Dynamic program slicing based on dependence relations, in: Proceedings of the Conference on Software Maintenance, 1991, pp. 191–200.
[13] R. Gupta, M. Harrold, M. Soffa, An approach to regression testing using slicing, in: Proceedings of the Conference on Software Maintenance, 1992, pp. 299–306.
[14] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, in: Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation, Atlanta, GA, 1988.
[15] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12(1) (1990) 26–61.
[16] S. Horwitz, T. Reps, The use of program dependence graphs in software engineering, in: Proceedings of the Fourteenth International Conference on Software Engineering, 1992.
[17] D.T. Huynh, Y. Song, Forward dynamic slicing in the presence of structured jump statements, in: Proceedings of ISACC'97, pp. 73–81.
[18] M. Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, Ph.D. thesis, Linköping University, 1993.
[19] M. Kamkar, P. Fritzson, N. Shahmehri, Three approaches to interprocedural dynamic slicing, Microprocessing and Microprogramming 38 (1993) 625–636.
[20] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29(3) (1988) 155–163.
[21] B. Korel, J. Laski, Dynamic slicing in computer programs, The Journal of Systems and Software 13(3) (1990) 187–195.
[22] B. Korel, R. Ferguson, Dynamic slicing of distributed programs, Applied Mathematics and Computer Science Journal 2(2) (1992) 199–215.
[23] B. Korel, S. Yalamanchili, Forward derivation of dynamic slices, in: Proceedings of the International Symposium on Software Testing and Analysis, Seattle, 1994, pp. 66–79.
[24] B. Korel, Computation of dynamic slices for programs with arbitrary control-flow, in: Second International Workshop on Automated and Algorithmic Debugging, St. Malo, France, 1995.
[25] B. Korel, Computation of dynamic slices for unstructured programs, IEEE Transactions on Software Engineering 23(1) (1997) 17–34.
[26] B. Korel, J. Rilling, Dynamic program slicing in understanding of program execution, in: Proceedings of the 5th International Workshop on Program Comprehension, 1997, pp. 80–90.
[27] B. Korel, J. Rilling, Application of dynamic slicing in program debugging, in: Third International Workshop on Automated Debugging (AADEBUG'97), 1997, pp. 59–74.
[28] J. Lyle, M. Weiser, Experiments on slicing-based debugging tools, in: Proceedings of the First Conference on Empirical Studies of Programming, 1986, pp. 187–197.
[29] K. Ottenstein, L. Ottenstein, The program dependence graph in a software development environment, in: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984, pp. 177–184 (SIGPLAN Notices 19(5)).
[30] G. Rothermel, M.J. Harrold, Selecting tests and identifying test coverage requirements for modified software, in: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, 1994, pp. 169–184.
[31] T. Shimomura, The program slicing technique and its application to testing, debugging and maintenance, Journal of IPS of Japan 9(9) (1992) 1078–1086.
[32] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3(3) (1995) 121–189.
[33] G. Venkatesh, Experimental results from dynamic slicing of C programs, Transactions on Programming Languages and Systems 17(2) (1995) 197–216.
[34] M. Weiser, Programmers use slices when debugging, Communications of ACM 25 (1982) 446–452.
[35] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10(4) (1984) 352–357.
[36] L. White, H. Leung, Regression testability, IEEE Micro (1992) 81–85.