# Dynamic Slicing Method for Maintenance of Large C Programs

Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik and Tibor Gyimóthy
Research Group on Artificial Intelligence, University of Szeged & HAS
Aradi vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544126
{beszedes,gertom,szabozs,csirik,gyimi}@cc.u-szeged.hu

## Abstract

*Different program slicing methods are used for maintenance, reverse engineering, testing and debugging. Slicing algorithms can be classified as static slicing and dynamic slicing methods. In several applications the computation of dynamic slices is more preferable since it can produce more precise results. In this paper we introduce a new forward global method for computing backward dynamic slices of C programs. In parallel to the program execution the algorithm determines the dynamic slices for any program instruction. We also propose a solution for some problems specific to the C language (such as pointers and function calls). The main advantage of our algorithm is that it can be applied to real size C programs, because its memory requirements are proportional to the number of different memory locations used by the program (which is in most cases far smaller than the size of the execution history—which is, in fact, the absolute upper bound of our algorithm).*

## Keywords

Software maintenence, reverse engineering, dynamic slicing

## 1. Introduction

Program slicing methods are widely used for maintenance, reverse engineering, testing and debugging (e.g. [6], [12]). A slice[1] consists of all statements and predicates that might affect the variables in a set $V$ at a program point $p$ [15], [8]. A slice may be an executable program or a subset of the program code. In the first case the behaviour of the reduced program with respect to a variable $v$ and program point $p$ is the same as the original program. In the second case a slice contains a set of statements that might influence the value of a variable at point $p$. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*) or compute those statements which influence the value of a variable occurrence for a specific program input (*dynamic slice*).

[1] In this paper we are dealing with backward slicing.

To determine whether a change at some place in a program will affect the behavior of other parts of a program is an important task of a software maintainer. *Decomposition slices* [6] are useful in making a change to a piece of software without unwanted side effects. The decomposition slice of a variable $v$ consists of all statements that may affect the value of $v$ at some point; it captures all computations of a variable and is independent of program location. One advantage of using decomposition slices is that after making a modification in a program the unaffected part of the program can be determined. Therefore program slicing is very useful to reduce the cost of *regression testing* [2]. Slicing-based techniques can be used in *program understanding* e.g. to locate *safety critical code* that may be interleaved throughout the entire system [5]. A slicing approach is able to identify all code that contributes to the value of variables that might be part of a safety critical component.

One of the problems in *reverse engineering* consists of understanding the current design of a program and the way this design differs from the original design. Program slices can be used to assist for this type of re-abstraction. A program can be represented as a lattice of slices ordered by the *is-a-slice-of* relation and this lattice can guide an engineer towards places where reverse engineering efforts should be concentrated ([5], [3]). Static slicing approaches have been used to the applications mentioned above. However, for realistic programs static slices may be very large due to the complexities in determining precise data- and control dependences. One of the problems is the static resolution of aliases due to procedure calls, pointer variables and array references. On the other hand, aliasing can be easily resolved by observing execution behavior for dynamic slicing. The dynamic slicing approach can be used to approximate static slices by constructing a union of program slices for each variable in the program over a large number of test runs [14].

Different dynamic slicing methods are introduced in e.g.

[10], [1]. In [1] Agrawal and Horgan presented a precise dynamic slicing method, which is based on the graph representation of the dynamic dependences. This graph, called Dynamic Dependence Graph (DDG) includes a distinct vertex for each occurrence of a statement. A dynamic slice created from the DDG with respect to a variable contains those statements that actually had an influence on this variable. (We refer to this slice as the *DDG slice*.) The major drawback of this approach is that the size of the DDG is unbounded. Although Agrawal and Horgan suggested a method for reducing the size of the DDG [1], even this reduced DDG may be very huge for an execution which has many different dynamic slices [13]. Therefore, the method of Agrawal et al. could not be applied for real size applications where for a given test case millions of execution steps may be performed.

In [7] we have introduced a method for the forward computation of backward dynamic slices (i.e. at each iteration of the method all slices are available for all variables at the given execution point). However, the presented method was applicable only to "toy" programs (i.e. with one entry procedure and with only scalar variables and simple assignment statements). The main contribution of the current paper is that this basic algorithm is extended for slicing real C programs. The extended algorithm presents the solution for handling the pointers, function calls (interprocedural slicing) and jump statements (the latter, though, is not presented in this paper due to space constraints). The main advantage of our approach is that it can be applied to *real size* C programs as well, because the memory requirements of the algorithm are proportional to the number of *different* memory locations used by the program, and not to the size of the execution history (number of steps—instructions—during program execution). Our experiences and preliminary test results show that this number of different memory addresses is substantially lesser than the size of the execution history.

The paper is organized as follows. In the next section the basic concepts of dynamic slicing are presented. Section 3 provides an introductory description of our algorithm for simple C programs. The extension of the algorithm for real C programs is presented in detail in Section 4. Section 5 discusses relations to other work. We have already developed a prototype for the algorithm presented in the paper. Section 6 summarizes our experience in this implementation and the future research is also highlighted.

## 2. Dynamic slicing

The goal of the introduction of dynamic slices was to determine those statements more precisely that may contain program faults assuming that the failure has been revealed for a given input.

Prior to the description of different dynamic slicing approaches some background is necessary, which is demonstrated using the example in Figure 1 (a).

A feasible path that has actually been executed will be referred to as an *execution history* and denoted by $EH$. Let the input be $a = 0, n = 2$ in the case of our example. The corresponding execution history is $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 \rangle$. We can see that the execution history contains the instructions in the same order as they have been executed, thus $EH(j)$ gives the serial number of the instruction executed at the $j$th step referred to as *execution position* $j$.

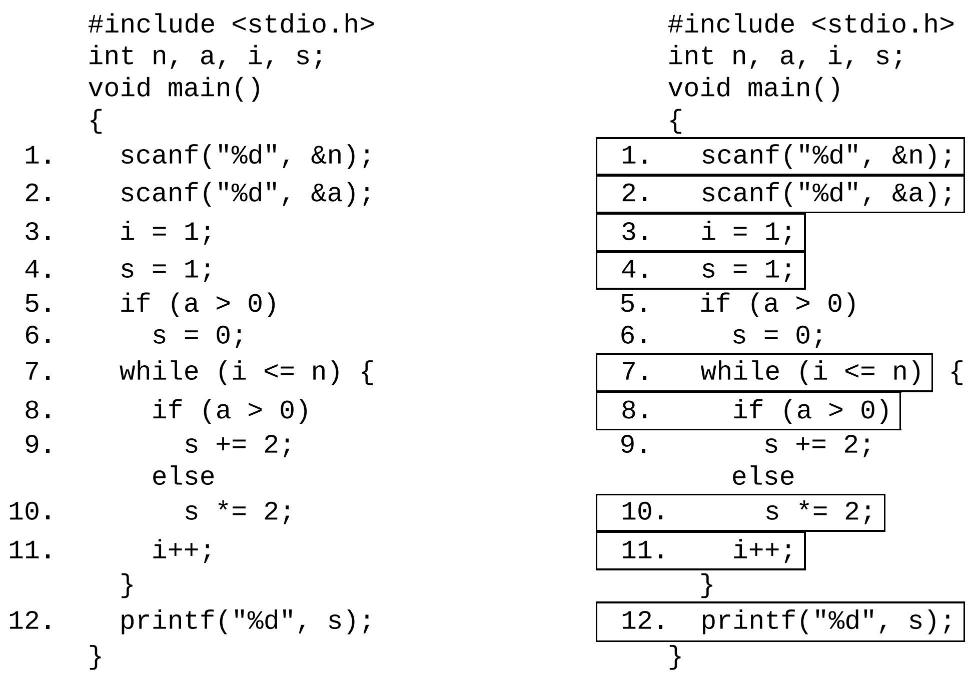
To distinguish between multiple occurrences of the same instruction in the execution history we use the concept of *action* that is a pair $(i, j)$, which is written down as $i^j$, where $i$ is the serial number of the instruction at the execution position $j$. For example, $12^{15}$ is the action for the output statement of our example for the input above.

We can define the *dynamic slicing criterion* as a triple $(\mathbf{x}, i^j, V)$ where $\mathbf{x}$ denotes the input, $i^j$ is an action in the execution history, and $V$ is the set of the variables for which the dynamic dependences should be computed.

Agrawal and Horgan [1] defined dynamic slicing as follows: given an execution history $H$ of a program $P$ for a test case $t$ and a variable $v$, the dynamic slice of $P$ with respect to $H$ and $v$ is the set of statements in $H$ whose execution had some effect on the value $v$ as observed at the end of the execution.

Agrawal and Horgan introduced a new method, which uses a Dynamic Dependence Graph (DDG) to take into account that the different occurrences of a given statement may be affected by different set of statements due to redefinitions of variables. In the DDG there is a distinct vertex for each occurrence of a statement in the execution history. Using this graph precise dynamic slice can be created. The main drawback of using the DDG is the size of this graph. The number of vertices in a DDG is equal to the number of executed statements, which is unbounded. To improve the size complexity of the algorithm Agrawal and Horgan suggested a method for reducing the number of vertices in the DDG. The idea of this method is that a new vertex is created only if it can create a new dynamic slice. Thus the size of this reduced graph is bounded by the number of different dynamic slices. It was shown in [13] that the number of different dynamic slices is in the worst case $O(2^n)$, where $n$ is the number of the statements.

If we compute a precise dynamic slice for the slicing criterion $(\langle \texttt{a=0}, \texttt{n=2} \rangle, 12^{15}, \texttt{s})$ using the DDG slicing method we get the dynamic slice of the program presented in Figure 1 (b).



```
       #include <stdio.h>
       int n, a, i, s;
       void main()
       {
 1.      scanf("%d", &n);
 2.      scanf("%d", &a);
 3.      i = 1;
 4.      s = 1;
 5.      if (a > 0)
 6.        s = 0;
 7.      while (i <= n) {
 8.        if (a > 0)
 9.          s += 2;
           else
10.          s *= 2;
11.        i++;
         }
12.      printf("%d", s);
       }
```

(a)

```
       #include <stdio.h>
       int n, a, i, s;
       void main()
       {
 1.      scanf("%d", &n);
 2.      scanf("%d", &a);
 3.      i = 1;
 4.      s = 1;
 5.      if (a > 0)
 6.        s = 0;
 7.      while (i <= n) {
 8.        if (a > 0)
 9.          s += 2;
           else
10.          s *= 2;
11.        i++;
         }
12.      printf("%d", s);
       }
```

(b)

**Figure 1. (a) A simple program. (b) The framed statements give the dynamic slice**

## 3. Forward computation of dynamic slices

For simplicity, we present our dynamic slice algorithm for C programs in two steps. First, the computation of the backward dynamic slice is described for programs with simple statements. Then this algorithm is extended to derive the dynamic slice for real C programs in Section 4.

Our algorithm is forward, which means that we obtain the necessary information (i.e. the dynamic slice for a given instruction) as soon as this instruction has been executed. As a consequence, our method is global, i.e. after the last instruction has been executed we obtain the dynamic slice for all the instructions processed previously. On the contrary, former methods involving backward processing compute the slices only for a selected instruction (and variables used at this instruction). Global slicing is very useful for testing and program maintenance.

Our algorithm does not necessitate a Dynamic Dependence Graph. Instead, we compute and store the set of statements that affect the currently executed instruction. This way we avoid any superfluous information (which may be unbounded).

Prior to the desciption of the algorithm some basic concepts and notations are overviewed and introduced. For clarity we rely on [10] but in some cases the necessary modifications have been made. We demonstrate our concepts for dynamic slicing by applying them on the example program in Figure 1.

We apply a program representation which considers only the *definition* and the *use* of variables and, in addition, it considers direct control dependences. We refer to this program representation as *D/U program representation*. An instruction of the original program has a D/U expression as follows:

$$i.\,d : U,$$

where $i$ is the serial number of the instruction and $d$ is the variable that gets a new value at the instruction in the case of assignment statements. For an output statement or a predicate $d$ denotes a newly generated "output variable"– or "predicate variable"–name of this output or predicate, respectively (see the example below). Let $U = \{u_1, u_2, ..., u_n\}$ such that any $u_k \in U$ is either a variable that is used at $i$ or a predicate-variable from which instruction $i$ is (directly) control dependent. Note that there is at most one predicate-variable in each $U$. (If the *entry* statement is defined, there is exactly one predicate-variable in each $U$.)

Our example has a D/U representation shown in Figure 2. Here $p5$, $p7$ and $p8$ are used to denote predicate-variables and $o12$ denotes the output-variable, whose value depends on the variable(s) used in the output statement.

Now we can derive the dynamic slice with respect to an input and the related execution history based on the D/U representation of the program as follows. We process each

| $i.$ | $d:$ | $U$ |
|---|---|---|
| 1. | $n:$ | $\emptyset$ |
| 2. | $a:$ | $\emptyset$ |
| 3. | $i:$ | $\emptyset$ |
| 4. | $s:$ | $\emptyset$ |
| 5. | $p5:$ | $\{a\}$ |
| 6. | $s:$ | $\{p5\}$ |
| 7. | $p7:$ | $\{i,n\}$ |
| 8. | $p8:$ | $\{p7,a\}$ |
| 9. | $s:$ | $\{s,p8\}$ |
| 10. | $s:$ | $\{s,p8\}$ |
| 11. | $i:$ | $\{i,p7\}$ |
| 12. | $o12:$ | $\{s\}$ |

**Figure 2. D/U representation of the program**

instruction in the execution history starting from the first one. Processing an instruction $i.\ d\ :\ U$, we derive a set $DynSlice(d)$ that contains all the statements which affect $d$ when instruction $i$ has been executed. By applying the D/U program representation the effect of data and control dependences can be treated in the *same way*. After an instruction has been executed and the related $DynSlice$ set has been derived we determine the *last definition* (serial number of the instruction) for the newly assigned variable $d$ denoted by $LS(d)$. Very simply, the last definition of variable $d$ is the serial number of the instruction where $d$ is defined last (considering the instruction $i.\ d\ :\ U$, $LS(d) = i$). Obviously, after processing the instruction $i.\ d\ :\ U$ at the execution position $j$ $LS(d)$ will be $i$ for each subsequent executions until $d$ is defined next time. We also use $LS(p)$ for predicates which means the last definition (evaluation) of predicate $p$.

Now the dynamic slices can be determined as follows. Assume that we are running a program on input $\mathbf{t}$. After an instruction $i.\ d\ :\ U$ has been executed at position $p$, $DynSlice(d)$ contains exactly the statements involved in the dynamic slice for the slicing criterion $C = (\mathbf{t}, i^p, U)$. $DynSlice$ sets are determined by the equation below:

$$DynSlice(d) = \bigcup_{u_k \in U} \Big(DynSlice(u_k) \cup \{LS(u_k)\}\Big)$$

After $DynSlice(d)$ has been derived we determine the value $LS(d)$ for assignment and predicate instructions, i.e.

$$LS(d) = i$$

Note that this computation order is strict, since when we determine $DynSlice(d)$ we have to consider $LS(d)$ occured at a former execution position instead of $p$ (consider the program line `x = x + y` in a loop).

We can see that during the dynamic slice determination we do not use a Dynamic Dependence Graph (which may be huge), but only a D/U program representation, which requires less space than the original source code and the method above creates the same dynamic slice as the application of the DDG in [2].

The formalization of the forward dynamic slice algorithm is presented in Figure 3.

```
program DynamicSlice
begin
    Initialize $LS$ and $DynSlice$ sets
    ConstructD/U
    ConstructEH
    for $j = 1$ to number of elements in EH
        the current D/U element is $i^j.\ d:\ U$
        $DynSlice(d) =$
            $\bigcup_{u_k \in U} \big(DynSlice(u_k) \cup \{LS(u_k)\}\big)$
        $LS(d) = i$
    endfor
    Output $LS$ and $DynSlice$ sets for the last definition of
        all variables
end
```

**Figure 3. Dynamic slice algorithm**

Note, that the construction of the execution history is achieved by instrumenting the input program and executing this instrumented code. The instrumentation procedure is discussed in Section 4.

Now we illustrate the above method by applying it on our example program in figure 1. for the execution history $\langle 1,2,3,4,5,7,8,10,11,7,8,10,11,7,12\rangle$. During the execution the following values are computed:

| Action | $d$ | $U$ | $DynSlice(d)$ | $LS(d)$ |
|---|---|---|---|---|
| $1^1$ | $n$ | $\emptyset$ | $\emptyset$ | 1 |
| $2^2$ | $a$ | $\emptyset$ | $\emptyset$ | 2 |
| $3^3$ | $i$ | $\emptyset$ | $\emptyset$ | 3 |
| $4^4$ | $s$ | $\emptyset$ | $\emptyset$ | 4 |
| $5^5$ | $p5$ | $\{a\}$ | {2} | 5 |
| $7^6$ | $p7$ | $\{i,n\}$ | {1,3} | 7 |
| $8^7$ | $p8$ | $\{p7,a\}$ | {1,2,3,7} | 8 |
| $10^8$ | $s$ | $\{s,p8\}$ | {1,2,3,4,7,8} | 10 |
| $11^9$ | $i$ | $\{i,p7\}$ | {1,3,7} | 11 |
| $7^{10}$ | $p7$ | $\{i,n\}$ | {1,3,7,11} | 7 |
| $8^{11}$ | $p8$ | $\{p7,a\}$ | {1,2,3,7,11} | 8 |
| $10^{12}$ | $s$ | $\{s,p8\}$ | {1,2,3,4,7,8,10,11} | 10 |
| $11^{13}$ | $i$ | $\{i,p7\}$ | {1,3,7,11} | 11 |
| $7^{14}$ | $p7$ | $\{i,n\}$ | {1,3,7,11} | 7 |
| $12^{15}$ | $o12$ | $\{s\}$ | {1,2,3,4,7,8,10,11} | 12 |

The final slice of the program can be obtained as the union of $DynSlice(o12)$ and $\{LS(o12)\}$.

## 4. Dynamic slicing of real C programs

In the previous section we have introduced an algorithm for forward computation of dynamic slices. For simplicity, the method was presented for simple C programs (only intraprocedural and only with scalar variables and assignment statements). In this section we extend our algorithm for real C programs. This means the solution of several problems, such as *pointers*, *function calls* and *jump statements*.

The necessity for handling the pointers prompts us to slightly extend the meaning of our slicing criterion. This means that, for example, if we want to compute the dynamic slice for a pointer dereference `*p`, we are actually seeking for dynamic dependences of a *memory location* (and not simply a variable, as in our original definition). (As we will see later, the slice for `*p` will include the dependences of the pointer `p` itself and the dereferenced memory location as well.)

We note, that the handling of arrays and structure members can be traced back to slicing of memory locations (as in the case of pointers).

The complete handling of C programs includes the handling of jump statements (`goto`, `switch`, `break`, `continue`). Our algorithm is capable of slicing such C programs too, however, due to space constraints we cannot present this technique here in detail (the full version of the mehod subsists in a technical report [4]).

Our method for slicing C programs involves the following main steps:

- First, by analyzing the input program a D/U representation is created based on static dependences in the program and the program is instrumented for creating the necessary runtime information.

- Next, the instrumented program is compiled and executed and this way a *trace of the execution* is created which contains the dynamic information needed by the dynamic slice algorithm (among others the $EH$). This is denoted by $TRACE$.

- Finally, the dynamic slice algorithm is executed for a certain slicing criterion using the previously created D/U representation and $TRACE$.

In the previous section the D/U representation is defined as $i.\ d : U$ for each program instruction $i$. For C programs, the D/U representation will contain a *sequence* of $d$ : $U$ *items* for each instruction as:

$$i.\ \langle (d_1 : U_1), (d_2 : U_2), \ldots \rangle$$

This is needed because in a C instruction (i.e. expression) several l-values can be assigned new values. Note that the sequence order is important, since $d$ values of a previous D/U item can be used by subsequent $U$ sets. This sequence order is determined by the "execution-order" (evaluation) of the corresponding subexpressions (the order of evaluation of subexpressions in C is not always defined by the language, however we can rely on the parsing sequence determined by the context-free grammar of C).

The defined variable $d$ and the used variables $u_k \in U$ can have several meanings. These are (see the example program in Figure 4):

- *Scalar variables*. These are the "regular" global or local variables (they have a constant address in the scope where they are declared).

- *Predicate variables*. Denoted by $pn$, where $n$ is the serial number of the predicate instruction as described in the previous section.

- *Output variables*. Denoted by $on$. By definition, output variables are a kind of "dummy" variables that are generated at those places where a set $U$ is used but no other regular variable takes any value from $U$. These include function calls with their return values neglected, single expression-statements with no side-effects and for simplicity some output statements in C (such as `printf`).

- *Dereference variables*. Denoted by $dn$, where $n$ is a global counter for each dereference occurrence. We use the notion of dereference variables where a memory address is used or gets a value through a pointer (or an array, structure member).

- *Function call argument variables*. Variables denoted by $arg(f, n)$, where $f$ is a function name and $n$ is the function argument (parameter) number. An argument variable is *defined* at the function call site and *used* at the entry point of the function.

- *Function call return variables*. Denoted by $ret(f)$, where $f$ is a function name. A return variable is *defined* at the exit point of the function and *used* at the function caller after returning.

In Figure 4. we can see an example C program and its statically computed D/U representation according to the notation described above.

In order to compute the dynamic slice, beside the static D/U representation we need to gather some dynamic information as well of the actual execution of the program. This is done by *instrumenting* the original program code[2] at all

[2]Basically, there are two alternatives for the type of instrumentation: source level and object-code level. In our algorithm source level instrumentation was chosen because of ease of portability to different platforms and of mapping the slice results to the original source code. However, some may argue that object level instrumentation could result in faster execution of instrumented code and also system and library calls could be handled more completely [14].

| $i.$ | | $\langle d : U \rangle$ |
|---|---|---|
| | `#include <stdio.h>` | |
| | `int a, b;` | |
| 1. | `int f(int x,int y) {` | $x : \{arg(f,1)\}$, $y : \{arg(f,2)\}$ |
| 2. | `  a += x;` | $a : \{a, x\}$ |
| 3. | `  b += y;` | $b : \{b, y\}$ |
| 4. | `  return x+2;` | $ret(f) : \{x\}$ |
| | `}` | |
| 5. | `int g(int y) {` | $y : \{arg(g,1)\}$ |
| 6. | `  a += y;` | $a : \{a, y\}$ |
| 7. | `  return y+1;` | $ret(g) : \{y\}$ |
| | `}` | |
| | `void main() {` | |
| | `  int s, *p;` | |
| 8. | `  s = 0;` | $s : \emptyset$ |
| 9. | `  scanf("%d", &a);` | $a : \emptyset$ |
| 10. | `  scanf("%d", &b);` | $b : \emptyset$ |
| 11. | `  p = &b;` | $p : \emptyset$ |
| 12. | `  while (*p < 10) {` | $p12 : \{p, d1\}$ |
| 13. | `    s += f(3,4);` | $arg(f,1) : \{p12\}$, $arg(f,2) : \{p12\}$, $s : \{s, ret(f), p12\}$ |
| 14. | `    s += g(3);` | $arg(g,1) : \{p12\}$, $s : \{s, ret(g), p12\}$ |
| | `  }` | |
| 15. | `  printf("%d", *p);` | $o15 : \{d2\}$ |
| 16. | `  printf("%d", s);` | $o16 : \{s\}$ |
| | `}` | |

**Figure 4. An example C program and its (static) D/U representation**

the relevant points in the program with certain dump-actions which will create a dump-file, the so-called $TRACE$ of the program. The instrumentation is performed in such a way, that after compiling the instrumented program it will behave identically to the original code. $TRACE$ contains all the necessary information about the actual execution (the $EH$ itself) and some other "administrative" information about the program and its execution as well, such as memory addresses of scalar variables, actual values of pointers, beginnings/endings of functions, etc.

After we have determined the D/U representation of the program and the $TRACE$ for a given execution we can derive the dynamic slice with respect to the given slicing criterion. We process the $TRACE$ from its beginning and perform different actions for each element. If the actual $TRACE$ element is an "administrative" element, then the necessary computations are made for the internal representations (such as maintaining a stack for the visible variables in block-scopes). If the element is an $EH$ element then the corresponding $i.\ \langle d : U \rangle$ sequence is processed as follows. For each D/U item in this sequence we determine the corresponding $d'$ and $u'_k \in U'$ "dynamic dependences" (i.e. memory locations and/or variables and predicates) and derive a set $DynSlice(d')$ that contains all the statements, which affect $d'$ when instruction $i$ has been executed and $LS(d')$ by the equations below:

$$DynSlice(d') = \bigcup_{u'_k \in U'} \Big( DynSlice(u'_k) \cup \{LS(u'_k)\} \Big) ,$$

$$LS(d') = i$$

Note, that this computation order is strict as seen in the previous section.

In the case of function calls the actual D/U sequence cannot be processed in a single iteration of the algorithm by processing a single $EH$ element "on the fly". In these cases the remaining sequence positions should be stacked and after the function has returned the remaining sequence items can be processed (see the formalized algorithm below).

The $d'$ and $u'_k$ values are determined from the corresponding $d$ and $u_k$ (statically computed) values as follows (the same conversions apply to $u_k$ as well):

- if $d$ is a *scalar variable* then $d'$ will be its memory location (this can be determined based on $TRACE$),
- if $d$ is a *dereference variable* then $d'$ will be the value (also a memory location) of the corresponding pointer (also from $TRACE$),
- if $d$ is a *predicate variable* then $d'$ is determined by supplementing the predicate variable $pn$ with a "depth-level" which corresponds to the depth of the *function-call stack*, denoted by $pn(k)$ (this is needed in the case of recursive functions because the same predicate should be considered as a different one in different instances of the same function),
- in all other cases $d$ and $d'$ are the same.

Now we can give a formalization of the dynamic slice algorithm for complete C programs in Figure 5.

We illustrate the above method by applying it on our example program shown in Figure 4 for the slicing criterion $(\langle$`a=2`$,$ `b=6`$\rangle, 15^{14},$ `*p`$)$. On this input we get the following execution history: $\langle 8, 9, 10, 11, 12, 13/1, 2, 3, 4/13, 14/5, 6, 7/14, 12, 15, 16\rangle$. The "dual" $EH$ elements $13/1$, $4/13$, $14/5$ and $7/14$ correspond to the function call/return and parameter passing "virtual statements". Parameter passing can be treated as two statements (but as one action), since first the caller puts the parameter on the function call stack and then the called function takes that value (returning a value can be interpreted similarly). The values computed during the execution are shown in Figure 6. (The numbers of the form `$xxxxxxx` are memory addresses supplied by $TRACE$.)

The final slice of the program can be obtained as the union of $DynSlice(o15)$ and $\{LS(o15)\}$, while the actual result is depicted in Figure 7. The slice contains the lines marked with the bullets in the first column. We can observe that the dynamic slice contains those statements, which influenced the value of memory location `$4347828` pointed by `p` (which is, in fact, the value of the scalar variable `b`).

As a point of interest, we can observe also that the dynamic slice for criterion $(\langle$`a=2`$,$ `b=6`$\rangle, 16^{15},$ `s`$)$ (the second column of bulleted lines) contains only those statements, which influence the value of scalar `s`, i.e. those statements, which influence the values of the two globals are not included, since `s` uses only the return values of the functions `f` and `g` (which are dependent only on constant values). Note, that this computation does not necessitates a separate execution of the algorithm, i.e. our method is "global" for a specific input (and $TRACE$) and slices of all variables at their last definition point can be determined simultaneously.

```
program DynamicSliceForC
begin
    Initialize $LS$ and $DynSlice$ sets
    ConstructD/U
    ConstructTRACE
    actual D/U item = nil
    for all lines of $TRACE$
        case the current line in $TRACE$ of
            function begin mark:
                push(actual D/U item)
                actual D/U item = nil
            function end mark:
                pop(actual D/U item)
            EH element:
                the current action in $EH$ is $i^j$
                actual D/U item = the first item in $i.\ \langle d : U\rangle$
            other:
                resolve the unresolved memory address
                    references in actual D/U item
        endcase
        while actual D/U item can be processed*
            compute $d'$ and $U'$ based on actual D/U item
            $DynSlice(d') =$
                $\bigcup_{u'_k \in U'} \left(DynSlice(u'_k) \cup \{LS(u'_k)\}\right)$
            $LS(d') = i$
            actual D/U item = the next item in $i.\ \langle d : U\rangle$
        endwhile
    endfor
    Output $LS$ and $DynSlice$ sets for the last definition
        of all memory locations
end
```

**This is true if: according to the static D/U there are no function calls at the actual D/U item position* **and** *item* $\neq$ *nil* **and** *item does not have any unresolved memory address references.*

**Figure 5. Dynamic slice algorithm for C programs**

## 5. Related work

Our method for the computation of dynamic slices significantly differs from the approach presented in [1]. This method takes into account that different occurrences of a statement may be affected by different set of statements. This approach uses a Dynamic Dependence Graph. Using this graph precise dynamic slice can be computed, but as mentioned earlier, the size of the DDGs may be unbounded. Agrawal and Horgan also proposed a reduced DDG method. The size of reduced graphs is bounded by the number of different dynamic slices (see Section 2 for details about the

| Action | $d'$ | $U'$ | $DynSlice(d')$ | $LS(d')$ |
|---|---|---|---|---|
| $8^1$ | `$6684144` | $\emptyset$ | $\emptyset$ | 8 |
| $9^2$ | `$4347824` | $\emptyset$ | $\emptyset$ | 9 |
| $10^3$ | `$4347828` | $\emptyset$ | $\emptyset$ | 10 |
| $11^4$ | `$6684148` | $\emptyset$ | $\emptyset$ | 11 |
| $12^5$ | $p12(1)$ | {`$6684148`, `$4347828`} | {10,11} | 12 |
| $13^6$ | $arg(f,1)$ | $\{p12(1)\}$ | {10,11,12} | 13 |
| $13^6$ | $arg(f,2)$ | $\{p12(1)\}$ | {10,11,12} | 13 |
| $1^6$ | `$6684060` | $\{arg(f,1)\}$ | {10,11,12,13} | 1 |
| $1^6$ | `$6684064` | $\{arg(f,2)\}$ | {10,11,12,13} | 1 |
| $2^7$ | `$4347824` | {`$4347824`, `$6684060`} | {1,9,10,11,12,13} | 2 |
| $3^8$ | `$4347828` | {`$4347828`, `$6684064`} | {1,10,11,12,13} | 3 |
| $4^9$ | $ret(f)$ | {`$6684060`} | {1,10,11,12,13} | 4 |
| $13^9$ | `$6684144` | {`$6684144`, $ret(f)$, $p12(1)$} | {1,4,8,10,11,12,13} | 13 |
| $14^{10}$ | $arg(g,1)$ | $\{p12(1)\}$ | {10,11,12} | 14 |
| $5^{10}$ | `$6684064` | $\{arg(g,1)\}$ | {10,11,12,14} | 5 |
| $6^{11}$ | `$4347824` | {`$4347824`, `$6684064`} | {1,2,5,9,10,11,12,13,14} | 6 |
| $7^{12}$ | $ret(g)$ | {`$6684064`} | {5,10,11,12,14} | 7 |
| $14^{12}$ | `$6684144` | {`$6684144`, $ret(g)$, $p12(1)$} | {1,4,5,7,8,10,11,12,13,14} | 14 |
| $12^{13}$ | $p12(1)$ | {`$6684148`, `$4347828`} | {1,3,10,11,12,13} | 12 |
| $15^{14}$ | $o15$ | {`$4347828`} | {1,3,10,11,12,13} | 15 |
| $16^{15}$ | $o16$ | {`$6684144`} | {1,4,5,7,8,10,11,12,13,14} | 16 |

**Figure 6. Values computed during the execution.**

DDG and reduced DDG methods). In [13] a simple program is presented (called $Q^n$) which has $O(2^n)$ different dynamic slices, where $n$ is the number of statements in the program. This example shows that even this reduced DDG may be very huge for some programs.

In [11] Korel and Yalamanchili introduced a forward method for determining dynamic program slices. Their algorithm computes executable program slices. In many cases these slices are less accurate then those computed by our forward dynamic slicing algorithm. (Executable dynamic slices may produce inaccurate results in the presence of loops [13].) The method of Korel and Yalamanchili is based on the notion of *removable blocks*. The idea of this approach is that during program execution, on each exit from a block the algorithm determines whether the executed block should be included in the dynamic slice or not.

Excellent comparison of different dynamic slicing methods can be found e.g in [13], [9].

## 6. Summary

Different program slicing methods are used for maintenance, reverse engineering, testing and debugging. Slicing algorithms can be classified as static slicing and dynamic slicing methods. In several applications the computation of dynamic slices is more preferable since it can produce more precise results. Experimental results from [14] show that the dynamic slice of a program can be expected to be less than 50% of the executed nodes and to be well within 20% of the entire program.

There have been several methods for dynamic slicing introduced in the literature, but most of them use the internal representation of the execution of the program with dynamic dependences called the Dynamic Dependence Graph (DDG). The main disadvantage of these methods is that the size of the DDGs is unbounded, since it includes a distinct vertex for each instance of a statement execution.

In this paper we have introduced a new forward global method for computing backward dynamic slices of C programs. In parallel to the program execution the algorithm determines the dynamic slices for any program instruction. We have also proposed a solution for some problems specific to the C language (such as pointers and function calls). The main advantage of our algorithm is that it can be applied to real size C programs, because its memory requirements are proportional to the number of different memory locations used by the program (which is in most cases far smaller than the size of the execution history—which is, in fact, the absolute upper bound of our algorithm).

We have already developed a prototype system, where we implemented our forward dynamic slicing algorithm for the C language. Our assumptions about the memory requirements of the algorithm turned out to be true according to our preliminary test results.

```
                                  | *p   s
-----------------------------------------
     #include <stdio.h>           |
     int a, b;                    |
                                  |
 1.  int f(int x,int y) {         |  •   •
 2.    a += x;                    |
 3.    b += y;                    |  •
 4.    return x+2;                |      •
     }                            |
                                  |
 5.  int g(int y) {               |      •
 6.    a += y;                    |
 7.    return y+1;                |      •
     }                            |
                                  |
     void main() {                |
       int s, *p;                 |
 8.    s = 0;                     |      •
 9.    scanf("%d", &a);           |
10.    scanf("%d", &b);           |  •   •
11.    p = &b;                    |  •   •
12.    while (*p < 10) {          |  •   •
13.      s += f(3,4);             |  •   •
14.      s += g(3);               |      •
       }                          |
15.    printf("%d", *p);          |  •
16.    printf("%d", s);           |      •
     }                            |
```

**Figure 7. The dynamic slices computed for the input** a=2 **and** b=6

We tested our system on several input programs (more than twenty) and several hundred separate executions for different slicing criterions. The average number of different memory locations was less than 10% of the size of execution histories. We computed a ratio for each test execution and after that we computed the average value (for large execution histories the ratios were less than 3%). The largest execution history consisted of more than 1 million actions.

Our future goal is to obtain empirical data on the size of dynamic slices in large C programs. We selected a set of C programs and a large number of test cases for each program. We are going to construct a union of dynamic slices for each variable in the program over all test runs to obtain an approximation of decomposition slices. As we mentioned earlier, decomposition slices are very useful in software maintenance and reverse engineering. We also wish to extend our approach to compute dynamic slices of C++ programs.

## Acknowledgements

The authors wish to acknowledge the work of Csaba Faragó including the implementation and testing of the presented algorithm.

This work was supported by the grants OTKA 23307 and T25721.

## References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *SIGPLAN Notices No. 6*, pages 246–256, 1990.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, Montreal, Canada, 1993.

[3] J. Beck. Program and interface slicing for reverse engineering. In *Proceeding of the Fifteenth International Conference on Software Engineering*, 1993. Also in *Proceedings of the Working Conference on Reverse Engineering*.

[4] Á. Beszédes, T. Gergely, Zs. M. Szabó, Cs. Faragó, and T. Gyimóthy. Forward computation of dynamic slices of C programs. Technical Report TR-2000-001, RGAI, 2000.

[5] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43, 1996. Marvin Zelkowitz, Editor, Academic Press San Diego, CA.

[6] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[7] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference (ESEC)*, pages 303–321, Toulouse, France, Sept. 1999. LNCS 1687.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[9] M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Systems Software*, 31:197–214, 1995.

[10] B. Korel and J. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[11] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, Washington, Aug. 1994.

[12] G. Rothermer and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of ISSTA'94*, pages 169–183, Seattle, Washington, Aug. 1994.

[13] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[14] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, Mar. 1995.

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