Handling the Unstructured Statements in the Forward Dynamic Slice Algorithm[1]

Csaba Faragó, Tamás Gergely

Research Group on Artificial Intelligence, Hungarian Academy of Sciences
Aradi vértanúk tere 1., H-6720 Szeged, Hungary, +36 62 544145
h633185@stud.u-szeged.hu, gertom@inf.u-szeged.hu

Abstract. Different program slicing methods are used for debugging, testing, reverse engineering and maintenance. Slicing algorithms can be classified as static slicing and dynamic slicing methods. In applications such as debugging the computation of dynamic slices is more preferable since it can produce more precise results. Tibor Gyimóthy et al. introduced a new so called "forward dynamic slice" algorithm. It has a great advantage compared to other dynamic slice algorithms that the used memory locations are proportional to the number of different memory locations used by the program, which is in most cases much smaller than the size of the execution history. The execution time of the algorithm is linear in the size of the execution history. In this paper we introduce the handling of the jump statements in the `C` language (`goto`, `break`, `continue`).

1 INTRODUCTION

Program slicing methods are widely used for debugging, testing, reverse engineering and maintenance (e.g. [3], [7], [2], [4]). A slice consists of all statements and predicates that might affect the variables in a set V at a program point p [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*).

In many applications (e.g. debugging) the computation of dynamic slices is more preferable since it can produce more precise results (i.e. the dynamic slice is smaller than the static one). In this paper we are concerned with dynamic slicing.

In [5] Gyimóthy et al. introduced a method for the forward computation of dynamic slices (i.e. at each iteration of the process slices are available for

[1] This work was supported by the grant OTKA T25721

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). In [9] the handling of the procedures and pointers, and the implementation of the algorithm were shown. In this paper we introduce how to handle the jump statements in the C programs. This method was not described in [9]. Besides the statement `goto` it solves the problem of statements `break` and `continue` which are in fact special cases of the `goto` statement – they are simply reduce to it. The handling of the statement switch-case-dafault is also described.

The paper is organized as follows: in the next section the "forward dynamic slice" method is introduced, the handling of the jump statements is presented in the section 3 and in the last section a short summary is described.

2 FORWARD COMPUTING OF THE DYNAMIC SLICE

In some applications static program slices contain superfluous instructions. This is the case for debugging, where we have dynamic information as well. Hence debugging may require smaller slices, which improves the efficiency of the bug finding process ([1], [6]). The goal of the introduction of dynamic slices was to determine more precisely those statements that may contain program faults, assuming that the failure has been revealed for a given input.

```
     #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);
     }
```

Fig. 1. Example program

Consider the example program in figure 1. The static slice of this code with respect to the variable s at vertex 12 contains all the statements.

Prior to the desciption of a new dynamic slice algorithm we introduce some basic concepts and notations.

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 instructions which are in the same order as they have been executed, so $EH(j)$ 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 make use of the notion of *action*. It is a pair (i, j) which is written 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 same input above.

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 a set of the variables. For a slicing criterion a dynamic slice can be defined as the set of statements which may affect the values of the variables in V.

We apply a program representation which only considers the definition of a variable, and the use of variables, and direct control dependences. We refer to this program representation as a *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 are ready to 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. Firstly, we process each instruction in the execution history starting from the first executed statement. Then after processing an instruction $i.\ d : U$, we derive a set $DynSlice(d)$ that contains all those statements which affect d when instruction i has been executed. By applying the D/U program representation

$i.$	$d:$	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.	$o12:$	$\{s\}$

Fig. 2. D/U representation of the program

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 $LS(d)$. Put simply, the last definition of variable d is the serial number of the instruction where d is defined last (considering the instruction $i.\ d : U$, $LS(d) = i$). Clearly, after processing the instruction $i.\ d : U$ at the execution position j $LS(d)$ have the value 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. For example, if $EH(10) = 7$ (the current action is 7^{10}) then $LS(d) = 7$.

Now the dynamic slices can be determined as follows. Assume that we are running a program on input t. After an instruction $i.\ d:\ U$ has been executed at position p, $DynSlice(d)$ contains just those statements involved in the dynamic slice for the slicing criterion $C = (t, i^p, U)$. *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 $LS(d)$ 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 whether $LS(d)$ occurred at a former execution position instead of p (like the program line `x = x + y` in a loop).

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

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 [9].

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

Fig. 3. Dynamic slice algorithm

Now we illustrate the above method by applying it to 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 is the union of $DynSlice(o12)$ and $\{LS(o12)\}$. (See figure 4.)

3 HANDLING THE UNSTRUCTURED STATEMENTS

A problem which must be dealt with is how we should handle the jump statements in the dynamic slicing algorithm. In this section C-specific jump statements

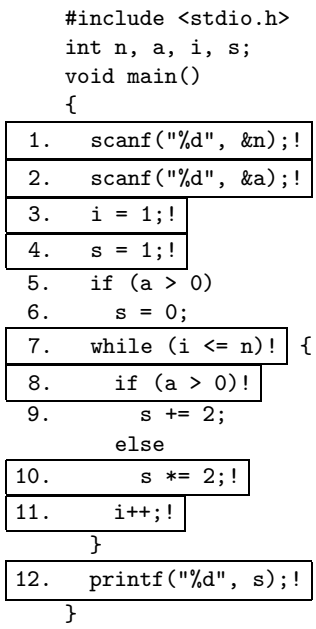

Fig. 4. The framed statements give the dynamic slice

are considered, but the method can be used in other programming languages, as well.

In the next subsection the handling of the `goto` statement is described, along with the `break`, `continue`, and `switch` statements.

3.1 The `goto` statement

Where the `goto` statement occurs, the D/U structure is built up as follows: so called *"label variables"* are introduced. Let the defined variable (d) be the prevously introduced label variable called the real name of the label. It could also be an ordinal number, but for sake of simplicity here we use the previous name. The use set (U) contains no "extra" variables, just the appropiate predicate variable and we will find that it can contain label variables, too.

The previously defined label variable is inserted the use set (U) of those statements which occur after the corresponding label within the function. It is important to do this to the end of the function, not only in the appropiate block.

If there are more labels, they are all handled in the same way. If the `goto` statement appears after the definition of the label, then of course it contains the just defined label variable. But this isn't a problem because in the execution history it appears as a former defined variable. It can be defined by itself or by another `goto` statement. If neither `goto` statement corresponding to a label executes during the program, the last definitios of the label statement remains

undefined so it won't affect the result of the dynamic slice. The result contains all of the defined labels.

When the `goto` is executed during the program and the dynamic slice contains at least one of the statements after the definition of the label, then the result will contain at least the previous corresponding `goto` (and of course its predicate dependencies transitively). So it often unnecessary increases the size of the dynamic slice. So using of many `goto` statements it makes hard to analyze the program.

A simple example is shown on figure 5, and its result is on figure 6. In spite of the result is computed at the first line, the second, the third and the forth statement also has effect of the final result, because if we change the value of variable `b` in the second statement, then the final result would be changed.

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`  int a,b;`	
1.	`  a=1;`	$a : \emptyset$
2.	`  b=1;`	$b : \emptyset$
3.	`  if (b==1)`	$p3 : \{b\}$
4.	`  goto l;`	$l : \{p3\}$
5.	`  a++;`	$a : \{a, l\}$
	`l:`	
6.	`  printf("%d",a);`	$o6 : \{a, l\}$
	`}`	

Fig. 5. A simple C program demonstrating how the `goto` statement works

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^4	$\{2, 3\}$
6^5	$\{1, 2, 3, 4\}$

Fig. 6. The results of program 5

A slightly bigger example is shown on figure 7, and its result on figure 8. In this example the weak point of handling the `goto` statement turns out. The result is correct.

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`  int i,j,k,l;`	
1.	`  k=0;`	$k : \emptyset$
2.	`  l=0;`	$l : \emptyset$
3.	`  i=0;`	$i : \emptyset$
	`l1:`	
4.	`  j=0;`	$j : \{l1\}$
	`l2:`	
5.	`  k=k+i+j;`	$k : \{k, i, j, l1, l2\}$
6.	`  l++;`	$l : \{l, l1, l2\}$
7.	`  j++;`	$j : \{j, l1, l2\}$
8.	`  if (j<2)`	$p8 : \{j, l1, l2\}$
9.	`  goto l2;`	$l2 : \{p8, l1, l2\}$
10.	`  i++;`	$i : \{i, l1, l2\}$
11.	`  if (i<2)`	$p11 : \{i, l1, l2\}$
12.	`  goto l1;`	$l1 : \{p11, l1, l2\}$
13.	`  printf("%d",k);`	$o13 : \{k, l1, l2\}$
	`}`	

Fig. 7. Another C program demonstrating how the `goto` statement works

3.2 The `break` statement

The `break` statement is equivalent to `goto` statement, which jumps out from the block of the appropiate `while`, `do...while`, `switch` or `for` statement to the first statement after this block. This problem can be solved as follows. The defined variable at every occurrence of the `break` statement should be an individual label variable. One form might be `break<Nr>`, where `<Nr>` is the ordinal number of the `break` statement within the program. All of the statements after the corresponding block are dependent on the previously defined label variable as introduced at the `goto` statement. Note that if a label is placed just after the corresponding block and the `break` is replaced with a `goto` which jumps to that label, then the result is the same.

An example of the `break` statement and its results are shown in figures 9 and 10 respectively.

3.3 The `continue` statement

Like the `break` statement we should define an individual `continue` named label variable. This might be denoted by `continue<Nr>`, where `<Nr>` is the ordinal number of the `continue` statement within the program. It is defined in statements where `continue` occurs. The dependent statements are statements from the beginning of the block of the appropriate `for`, `while` or `do...while` statement to the end of the function. So the `continue` statement is always dependent on itself.

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\emptyset$
5^5	$\{1, 3, 4\}$
6^6	$\{2\}$
7^7	$\{4\}$
8^8	$\{4, 7\}$
9^9	$\{4, 7, 8\}$
10^5	$\{1, 3, 4, 5, 7, 8, 9\}$
11^6	$\{2, 4, 6, 7, 8, 9\}$
12^7	$\{4, 7, 8, 9\}$
13^8	$\{4, 7, 8, 9\}$
14^{10}	$\{3, 4, 7, 8, 9\}$
15^{11}	$\{3, 4, 7, 8, 9, 10\}$
16^{12}	$\{3, 4, 7, 8, 9, 10, 11\}$
17^4	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
18^5	$\{1, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$
19^6	$\{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$
20^7	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
21^8	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
22^9	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
23^5	$\{1, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$
24^6	$\{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$
25^7	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
26^8	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
27^{10}	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
28^{11}	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
29^{13}	$\{1, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$

Fig. 8. The results of program 7

An example of the `continue` statement and its results are shown in figures 11 and 12 respectively.

3.4 The `switch` statement

After the handling `break` statement has been dealt with, then the handling of the `switch` statement is quite straightforward.

At place where the `switch` statement occurs a predicate variable is defined, just like those with `while` or `if`. All of the statements within the `switch` block are dependent on this predicate variable. If at least one statement within the switch block is included to the slice result, all of the `case` labels and the accidental `default` label are included. Here the `break` statements are handled with the previously described manner.

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`int a,b,i;`	
1.	`a=1;`	$a : \emptyset$
2.	`b=1;`	$b : \emptyset$
3.	`i=2;`	$b : \emptyset$
4.	`while (i>0) {`	$p4 : \{i\}$
5.	`b--;`	$b : \{p4, b\}$
6.	`i--;`	$i : \{p4, i\}$
7.	`if (b==0)`	$p7 : \{b\}$
8.	`break;`	$break8 : \{p7\}$
9.	`a++;`	$a : \{p4, a\}$
	`}`	
10.	`printf("%d",a);`	$o10 : \{a, break8\}$
	`}`	

Fig. 9. A simple C program demonstrating how the `break` statement works

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2, 3, 4\}$
6^6	$\{3, 4\}$
7^7	$\{2, 3, 4, 5\}$
8^8	$\{2, 3, 4, 5, 7\}$
9^{10}	$\{1, 2, 3, 4, 5, 7, 8\}$

Fig. 10. The results of program 9

An example of the `switch` statement and its results are shown in figures 13 and 14 respectively.

4 SUMMARY

Different program slicing methods are used for debugging, testing, reverse engineering and maintenance. Slicing algorithms can be categorized according to whether they use static slicing or dynamic slicing methods. In applications such as debugging the computation of dynamic slices is more preferable since it can produce more precise results.

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 dependencies called the Dynamic Dependence Graph

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`int a,b,i;`	
1.	`a=1;`	$a : \emptyset$
2.	`b=1;`	$b : \emptyset$
3.	`i=2;`	$b : \emptyset$
4.	`while (i>0) {`	$p4 : \{i, continue8\}$
5.	`b--;`	$b : \{p4, b, continue8\}$
6.	`i--;`	$i : \{p4, i, continue8\}$
7.	`if (b==0)`	$p7 : \{b, continue8\}$
8.	`continue;`	$continue8 : \{p7, continue8\}$
9.	`a++;`	$a : \{p4, a, continue8\}$
	`}`	
10.	`printf("%d",a);`	$o10 : \{a, continue8\}$
	`}`	

Fig. 11. A simple C program demonstrating how the `continue` statement works

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2, 3, 4\}$
6^6	$\{3, 4\}$
7^7	$\{2, 3, 4, 5\}$
8^8	$\{2, 3, 4, 5, 7\}$
9^4	$\{2, 3, 4, 5, 6, 7, 8\}$
10^5	$\{2, 3, 4, 5, 6, 7, 8\}$
11^6	$\{2, 3, 4, 5, 6, 7, 8\}$
12^7	$\{2, 3, 4, 5, 6, 7, 8\}$
13^9	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
14^4	$\{2, 3, 4, 5, 6, 7, 8\}$
15^{10}	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Fig. 12. The results of program 11

(DDG). The main disadvantage of these methods is that the size of the DDGs is unbounded, since it includes a distinct vertex for each occurrence of a statement.

In [9] new forward global method for computing dynamic slices of C programs were introduced. In parallel with program execution the algorithm determines the dynamic slices for any program instruction.

This paper mainly devoted itself to the handling of unstructured jump statements. A method of handling the `goto`, `break`, `continue` and `switch` statements

$i.$		$\langle d : U \rangle$	
	`#include <stdio.h>`		
	`void main() {`		
	`  int a,b;`		
1.	`  b=0;`	$b : \emptyset$	
2.	`  a=2;`	$a : \emptyset$	
3.	`  switch (a) {`	$p3 : \{a\}$	
	`  case 1:`		
4.	`  b=5;`	$b : \{p3\}$	
5.	`  break;`	$break5 : \{p3\}$	`case 2:`
6.	`  b=3;`	$b : \{p3\}$	
	`  case 3:`		
7.	`  b++;`	$b : \{p3, b\}$	
8.	`  break;`	$break7 : \{p3\}$	`default:`
9.	`  b=6;`	$b : \{p3\}$	
	`  }`		
10.	`  printf("%d",b);`	$o10 : \{b, break5, break7\}$	
	`}`		

Fig. 13. A simple C program demonstrating how the `switch` statement works

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^6	$\{2, 3\}$
5^7	$\{2, 3, 6\}$
6^8	$\{2, 3\}$
7^{10}	$\{2, 3, 6, 7, 8\}$

Fig. 14. The results of program 13

of the `C` programming language was described. The solution of the handling functions and pointers is described in [9].

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, actually, the absolute upper bound).

We have already developed a program, in which we implemented the forward dynamic slicing algorithm for the C language. Our assumptions about the memory requirements of the algorithm turned out to be well-founded. According to our preliminary test results it is indeed proportional to the number of different memory locations used by the program, which is much less than the size of the execution history.

References

1. Agrawal, H., DeMillo, R. A., and Spafford, E. H. Debugging with dynamic slicing and backtracking. *Software—Practice And Experience*, 23(6):589-616, June 1993.
2. Beck, J., and Eichmann, D. Program and Interface Slicing for Reverse Engineering. In *Proc. 15th Int. Conference on Software Engineering*, Baltimore, Maryland, 1993. IEEE Computer Society Press, 1993, 509-518.
3. Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimóthy, T. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems 1*, 4 (1992), 303-322.
4. Gallagher, K. B., and Lyle, J. R. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering* 17, 8, 1991, 751-761.
5. Gyimóthy, T., Beszédes, Á, and Forgács, I. An Efficient Relevant Slicing Method for Debugging. In *Proc. 7th European Software Engineering Conference (ESEC)*, Toulouse, France, Sept. 1999. LNCS 1687, pages 303-321.
6. Korel, B., and Rilling, J. Application of dynamic slicing in program debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97)*, Linkoping, Sweden, May 1997.
7. Rothermer, G., and Harrold, M. J. Selecting tests and identifying test coverage requirements for modified software. In *Proc. ISSTA'94* Seattle. 1994, 169-183
8. Weiser M. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4, 1984, 352-357.
9. Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J., and Gyimóthy T. *Dynamic Slicing Method for Maintenance of Large C Programs.* At the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001). Lisbon, Portugal, March 14-16, 2001.