Appl. Math. and Comp. Sci., 1993, vol.3, No.2, 341–355

PROGRAM CHANGE ANALYSIS

BOGDAN KOREL*

Software maintenance involves making changes to a program to correct errors, to improve efficiency, or to extend the program functionality. For large programs, understanding the influence of program changes can be very aggravating. Program change analysis is a method of analyzing the differences between the original program and its modified version in order to determine the scope and effect of a modification. The goal of program change analysis is to provide support for programers to understand program modifications (e.g., the influence of a modification on the selected program outputs) or to understand possible unintended influences introduced by the modification. In addition, the change analysis is used to automatically detect anomalies or errors related to the modification in order to indicate possible problems with the modification. The program change analysis presented in this paper is based on the program dependence analysis. First, different classes of possible modifications that are made to the program are identified. Then, based on the type of modification, program dependence analysis is used to determine the affected parts of the program and possible unintended influences. A prototype of the change analysis tool has been recently implemented for Pascal programs.

1. Introduction

The maintenance phase of the Systems Life Cycle has been estimated to consume 60 to 80 percent of the total life cycle cost of the typical system (Schneidewind, 1987). Therefore, developing techniques and tools that enhance the software maintenance phase could be very cost effective. Using proper maintenance techniques can increase the effectiveness of detecting a greater percentage of errors leading to a higher quality of the software product. Software maintenance involves making changes to a program to correct errors, to improve efficiency, or to extend the program functionality. For large programs, understanding the influence of program changes can be very aggravating. It is almost impossible for a programer to trace possible affects of a modification in large programs. Program change analysis is a method of analyzing the differences between the original program and its modified version in order to determine the scope and effect of a modification. The goal of program change analysis is to provide support for programers to understand the

* Department of Computer Science, Wayne State University, Detroit, MI 48202, USA, E–mail: bmk@cs.wayne.edu

effects of program modifications (e.g., the influence of a modification on program output or other parts of the program), to understand possible unintended influences introduced by the modification, etc. In addition, the change analysis is used to automatically detect anomalies or errors related to the modification in order to indicate possible problems with the modification. For example, the change analysis may determine that the modification does not influence an output variable that was incorrect in the original program on some program input. This is, obviously, an indication of an erroneous modification since the modification was supposed to change the value of the output variable on this input. The intention of change analysis is to draw the programer's attention to anomalies in the modified program. While these are not necessarily erroneous conditions, it is possible that many of these anomalies are the result of erroneous modifications.

In the related research mainly (backward) program slicing (Weiser, 1984) has been used for software maintenance (Gallagher and Lyle, 1991; Schneidewind, 1987; Yau and Kishimoto, 1987). We believe, however, that for the change analysis in software maintenance this type of program slicing is of limited use. In addition, formal approach for change analysis has been proposed in (Moriconi and Winkler, 1990). However, because of serious limitations of this approach (e.g., difficulty of automatic analysis for realistic programs), the approach does not seem to be of practical value.

In this paper we propose to use the notion of forward slicing and input–output relation analysis for program change analysis. The program change analysis presented in this paper is based on the program dependence analysis. First, different classes of possible modifications that may be made to the program are identified. Then, based on the type of modification, program dependence analysis is used to determine the affected parts of the program (and possible unintended influences) or to detect potential anomalies related to modifications.

An experimental prototype of the change analysis tool which supports software maintenance has been recently implemented. This tool automatically identifies types of modification by comparing the modified program and its old version. Depending on the modification type, the tool computes the influenced areas of the program based on static analysis (Ferrante *et al.*, 1987; Horowitz *et al.*, 1990; Korel, 1987). Users can query the tool to perform different types of analysis in order to better understand the possible effect of modifications and to detect possible anomalies. This tool allows programers to concentrate only on those parts of the program that relate to the modification. The tool has been implemented for a subset of Pascal programming language.

In the next section, we present program dependence concepts. Section 3 discusses types of program modifications and their influence on the program. In section 4, we introduce forward slicing and input–output analysis and show how they are used in change analysis. Finally, in conclusions future research is presented.

2. Program Dependence Concepts

To facilitate the presentation, we define some of the terminology that will be used in this paper. For the sake of simplicity, we restrict our analysis to a subset of structured Pascal–like programming language constructs, namely: sequencing, if–then–else and while statements. These restrictions are merely to simplify the presentation, and the general theory can be extended to complete programs written in any procedural language. A *flow graph* of program Q is a directed graph $C = (N, E, s, e)$ where (1) N is a set of nodes, (2) E is a binary relation on N (a subset of $N \times N$), referred to as a set of edges, and (3) s and e are, respectively, unique entry and unique exit nodes, $s, e \in N$. A node in N corresponds to the smallest single–entry, single–exit executable part of a statement in Q that cannot be further decomposed; such a part is referred to as a node. A single node corresponds to an assignment statement, an input or output statement, or the <expression> part of an if–then–else or while statement, in which case it is called a conditional node. An *edge* $(n_i, n_j) \in E$ corresponds to a possible transfer of control from node n_i to node n_j. For instance, (3,4), (7,8), and (7,13) are edges in the program of Figure 1. A *path* P in a control flow graph is a sequence $P = <n_{k_1}, n_{k_2}, ..., n_{k_q}>$ of nodes, such that $n_{k_1} = s$, and for all i, $1 \leq i < q$, $(n_{k_i}, n_{k_{i+1}}) \in E$. A use of variable y is a node n in which this variable is referenced. A use can be a conditional node, an assignment node, or an output node. A definition of variable y is a node n which assigns a value to that variable. A definition can be an assignment node or an input node. Let USE(n) be a set of variables whose values are used in node n and let DEF(n) be a set of variables whose values are defined in n. The sets USE and DEF for each node in a program can be identified by static analysis. An input variable of a program is a variable that appears in an input statement, e.g., read(x). Similarly, an output variable is a variable that appears in an output statement, e.g., write(x).

Program dependence concepts. The program dependences (Bergeretti and Carre, 1985; Ferrante *et al.*, 1987; Horowitz *et al.*, 1990; Korel, 1987) are broken into two kinds of dependences namely the data dependence and control dependence.

Data Dependence. The data dependence is represented by data flow (definition–use chain). More formally, given nodes n_i and n_j, n_j is *data dependent* on n_i by variable v iff (1) $v \in$ USE(n_j), (2) $v \in$ DEF(n_i), and (3) there exists a control path from n_i to n_j along which v is not modified. For instance, in the program of Figure 1, node 11 is data dependent on node 6 because variable i is defined in node 6 and used in node 11 and there exists a control path from 6 to 11 along which i is not modified.

Control Dependence. The control dependence is defined between conditional nodes and the nodes which can be chosen to execute by the conditional nodes. For the sake of presentation, we define control dependence only for if–then–else and while–statements (a method for finding control dependences for arbitrary programs is given in (Ferrante *et al.*, 1987)).

```
1. program Main
2.   read(A);
3.   read(B);
4.   read(n);
5.   call init(mn, mx, s);
6.   i := 1;
7.   while i <= n do
8.     call max(A[i], mx);
9.     call min(B[i], mn);
10.    call sum(s, B[i]);
11.    i := i + 1;
12.  endwhile;
13.  write(mn);
14.  write(mx);
15.  write(s);
16. end;

17. procedure init(x, y, z)
18.   x := 0;
19.   y := 0;
20.   z := 0;
21. return;

22. procedure max(x, y)
23.   if x > y then
24.             y := x;
25. return;

26. procedure min(x, y)
27.   if x < y then
28.             y := x;
29. return;

30. procedure sum(x, y)
31.   x := x + y;
32. return;
```

The program is supposed to compute the minimal element (represented by output variable mn) in input array B, the sum of all elements (represented by output variable s) of input array B, and the maximal element (represented by output variable mx) in input array A.

Fig. 1. A sample program with procedures.

a) **if** n_i **then** B_1 **else** B_2
   n_j is control dependent on n_i iff n_j appears in the control subgraph of B_1 or B_2.
b) **while** Z **do** B;
   n_j is control dependent on n_i iff n_j appears in the control subgraph of B.

The control dependences capture the dependence between conditional nodes and nodes which can be chosen to execute or not execute by these conditional nodes. For instance, nodes 9 and 10 in the program of Figure 1 are control dependent on node 7, but node 13 is not control dependent on node 7.

Program Dependence Graph. The dependence relations can be represented as a directed graph, where vertices represent nodes, and edges represent data and control dependences. This graph will be called the program dependence graph (Ferrante *et al.*, 1987; Horowitz *et al.*, 1990; Korel, 1987). More formally we give its definition as follows: A Program Dependence Graph of a program P is a graph $G = (N, A)$, such that N is a set of nodes in program P, and $A = \{(n_i, n_j) \in N \times N |\ n_j$ is Data or Control dependent on $n_i\}$.

System Dependence Graph. The system dependence graph (Horowitz *et al.*, 1990) is an extension of the program dependence graph used to represent programs with procedures. The system dependence graph combines the dependence graphs for the individual procedures with additional edges and nodes representing procedure calls. Procedures introduce new types of nodes into the dependence graph.

For each procedure definition, an "entry" node and a "return" node are introduced. These nodes represent an entry to the procedure and exit from the procedure, respectively. In addition, special types of nodes are introduced that represent formal and actual parameters. With each procedure calling node associated are nodes called *actual–in* and *actual–out* nodes. These nodes represent actual input and output parameters of the procedure call. Similarly, with each procedure entry node associated are nodes called *formal–in* and *formal–out* nodes. These nodes represent formal input and output parameters of the procedure. Directed edges in the system dependence graph are added that connect actual–in nodes with formal–in nodes and edges that connect formal–out nodes with actual–out nodes. Those edges model the process of passing parameters (input data) from the procedure call into the procedure and passing output data from the procedure to the procedure call. Additionally, new control edges are introduced that model control dependence between (1) a calling node and the procedure entry node, (2) a calling node and formal–in/formal–out nodes, and (3) a procedure entry node and all nodes inside of the procedure (including actual–in/actual– out nodes). *A* sample system dependence subgraph for the program of Figure 1 is given in Figure 2.

We say that node n_i has *influence* on node n_j iff there exists a path in the system dependence graph from n_i to n_j. For example, from the dependence subgraph of Figure 2, it is easy to see that node 3 has influence on node 13.

3. Program Change Analysis

Program change analysis is a method of analyzing the differences between the original program and its modified version in order to determine the scope and effect of a modification. The goal of program change analysis is to provide support for programers to understand the effects of program modifications (e.g., the influence of a modification on program output or other parts of the program), to understand possible unintended influences introduced by the modification, etc. Let P be an original program and Q be its modified version. Intuitively, a program modification has influence on output variable y if there exists a program input for which the final value of y computed by P is different from the final value computed by Q. Although it is undecidable whether a program modification actually leads to a change in behavior, it is possible to determine a safe approximation of the set of changed computations. To compute this information, we use a dependence–graph representation of programs (Ferrante *et al.*, 1987; Horowitz *et al.*, 1990).

In addition, the change analysis can be used to automatically detect anomalies or errors related to the modification in order to indicate possible problems with the modification. The intention of the change analysis is to draw the programer's attention to anomalies in the modified program. While these are not necessary erroneous conditions, it is possible that many of these anomalies are the result of erroneous modifications.

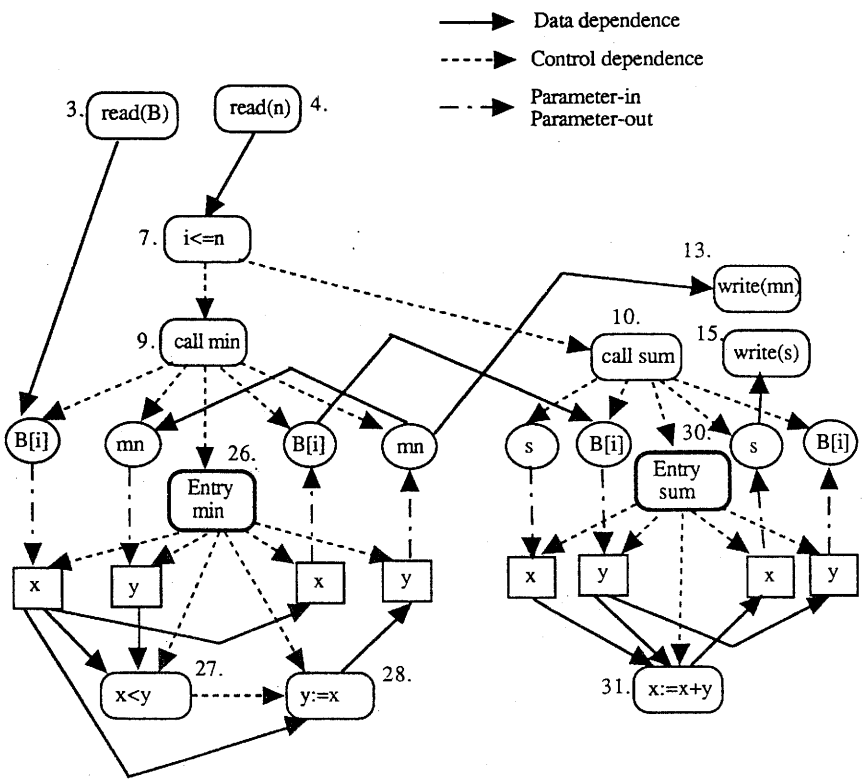

Fig. 2. System dependence subgraph of the program in Figure 1. Square nodes represent formal–in/formal–out nodes, circle nodes represent actual–in/actual–out nodes.

Depending on the different types of maintenance (perfective maintenance, corrective maintenance, or adaptive maintenance) different types of analysis can be performed. In perfective maintenance changes are made to the program because those changes should change, for example, the effectiveness of the program; in this type of maintenance, the functional behavior of the program should not change. In corrective maintenance the major reason for making changes to the program is to correct detected faults. In this type of maintenance, the behavior of the program is expected to be different, especially the incorrect output. In adaptive maintenance, changes are required because of changes in the environment of the program; in adaptive maintenance, the specification of the program is usually changed.

Modifications can be identified on different level of granuality, for example, modifications can be done on an assignment statement level, a predicate level, if–then–else

and while–statement level, procedure level, variable level, etc. Each modification may affect directly either a value of a variable or a flow of control at some point in the program. The first step is to identify for each modification the influenced variables or flow of control. This is a starting point of the change analysis. When the influenced variables (or control flow) for each modification are identified, we can determine, in the next step, those parts of the program which are influenced by those variables or modified test predicates. For this purpose, program dependence analysis is applied. Different modifications for different type of maintenance may require different types of change analysis because each modification can introduce new influences between program elements or cause the removal of others influences.

In what follows we present sample modifications on an assignment–statement level and test–predicate level, and, then, illustrate directly influenced program elements (variables or control flow). Let P be an original program and Q its modified version.

Modifications of an *Assignment Statement*:

a) *Right–hand side is modified.*
In this type of modification the right hand side of an assignment statement is modified. In this modification, a variable on the left–hand side of the assignment statement is influenced. For example, for the following modification

Initial	Modified
$y := x + z$	$y := x - z$

variable y is directly influenced, i.e., each time when the modified assignment statement is executed in Q the value of y may be different from the value of y in the original program P.

b) *Left–hand side is modified.*
In this type of modification the left hand side of an assignment statement is modified. In here two variables are influenced. For example, for the following modification:

Initial	Modified
$y := x + z$	$v := x + z$

variables y and v are directly influenced, i.e., values of those variables in the modified program Q may be different from values in the original program P. Although both variables y and v are influenced by this type of modification, they are influenced differently. Variable v is directly influenced by the modified statement in Q. On the other hand, variable y is not influenced any more by this assignment statement in Q.

Modification of a *Test Predicate*:

Initial	Modified
if $x < y$ **then** $z := 1$	**if** $x <= y$ **then** $z := 1$

In this type of modification, a flow of control is directly influenced, i.e., all nodes that are control dependent on the modified conditional node are influenced by this modification. In the above example, the execution of "$z := 1$" statement is influenced. A more extensive list of possible modifications and influenced variables (and control flow) are presented in Appendix A.

In summary, each modification can affect directly either variables or a flow of control at some point in the program. If the modification affects variables in the program, we distinguish three sets of variables associated with this modification. Let n be a node before modification and n' be its modified version.

A set V = DEF(n) $\bigcup$ DEF(n') represents all influenced variables by the modification.

A set V_1=DEF(n') - DEF(n) represents newly influenced variables, i.e., variables that were not influenced by node n (before the modification) but are influenced by modified node n'.

A set V_2=DEF(n) - DEF(n') represents variables not influenced any more by the modified node, i.e., variables that are not influenced by modified node n' but were influenced by node n (before modification).
These sets are used in the change analysis presented in the next section.

4. Program Dependence–Oriented Change Analysis

We now review a concept of forward slicing and input–out relation that are used in our change analysis.

Forward Slicing. A program slice (Weiser, 1984) with respect to a program point p and variable v consists of all statements and predicates of the program that might affect the value of v at point p. Program slicing has extensively been used for debugging, testing, reverse engineering, and software maintenance. We believe, however, that for the change analysis in software maintenance program slicing is of limited use. In this paper we propose to use the notion of forward slicing (Horowitz *et al.*, 1990) for program change analysis. The forward slice FS(p, v) of a program with respect to a program point p and variable v consists of all statements and predicates of the program that might be influenced by the value of x at point p. For a given program modification at point p that directly affects variable v, forward slice FS(p, v) represents all statements that are influenced by this modification, i.e., those statements that might manifest different behavior.

In order to find forward slice we use program dependence analysis, i.e., a system dependence graph. More formally we find this forward slice as follows:

$$\mathrm{FS}(p,v) = \mathrm{USE}'(p,v) \cup \{n \in N | \text{ there exists } Z \in \mathrm{USE}'(p,v) \text{ such that there exists a path in the system dependence graph from } Z \text{ to } n\}$$

where $\mathrm{USE}'(p,v) = \{n \in N | v \in \mathrm{USE}(n)$ and there exists a control path in the program's control flow graph from point p to n along which v is not modified$\}$, and $\bigcup$ is a set union.

For example, for the program of Figure 1, a forward slice for variable B at point 3 can be derived from the system dependance subgraph of Figure 2:

$$\mathrm{FS}(3,B) = \{9,10,13,15,26,27,28,30,31\}.$$

Let X be a conditional node. Similarly for forward slicing, we introduce a set $\mathrm{INF}(X)$ of nodes that might be influenced by X. More formally, we find this set as follows:

$$\mathrm{INF}(X) = \{n \in N | \text{ there exists a path in the system dependence graph from } X \text{ to } n\}.$$

This set is used to determine the influenced parts of the program for modification affecting flow of control, e.g., modification in a test predicate.

Input–Output Relation Analysis. In this section we present an input–output relation analysis (Korel, 1987) that may be effective in detection of some types of anomalies or errors related to the modification. The main idea is to derive a relationship between program inputs and program outputs. Clearly, the goal is to identify the specific input elements of the program that might influence the specific output elements. This information may be helpful in providing better understanding of the program, and it can be used to check for possible discrepancies. Since in many programs the same input variables may appear in different input statements, each input element will be identified by a variable name and an input node in which the variable appears. Similarly, output elements are identified by a variable name and an output node in which this variable appears. For the sake of presentation we assume that only one input variable is specified in each input statement and, similarly, only one output variable is specified in each output statement.

The following specifies a method of finding whether a particular input element may influence a particular output element, by using system dependence graph: We say that an input variable x at input node I may influence an output variable y at output statement O iff $O \in \mathrm{FS}(I,x)$.

For example, it can be determined from the system dependence graph of Figure 2 that input variables n and B have influence on output variables mn and s. The following is the complete input–output relation for the program of Figure 1.

Output variable	Input variables
mn	B, n
s	B, n
mx	A, n

Program Change Analysis. Let P be an original program and Q its modified version, and let n be a node in P and n' be its modified version in Q. With this modification three sets are associated: set V that represents all influenced variables by the modification, set V_1 that represents newly influenced variables, and set V_2 that represents variables not influenced any more by the modified node. For example, suppose that statement 28 in the program of Figure 1 has been modified from "$y := x$" to "$x := y$". Based on this modification, we can determine that $V = \{x, y\}$, $V_1 = \{x\}$, $V_2 = \{y\}$.

In order to evaluate the effect of the modification on the program, programers may need to have different types of information related to this modification. In what follows we present a list of possible analyses that can be performed to support programers in this evaluation process.

1. Influenced nodes

In order to determine the influence of the modification on the other parts of the program, forward slicing is used for all influenced variables:

$$\bigcup_{v \in V} \mathrm{FS}(n', v)$$

The influenced program parts are represented by a union of all forward slices of variables influenced by the particular modification at node n. This union represents all nodes that may manifest different behavior in Q. For instance, for our example,

$$\mathrm{FS}(28, x) \cup \mathrm{FS}(28, y) = \{9, 10, 13, 15, 26, 27, 30, 31\}.$$

Notice that

$$\mathrm{FS}(28, x) = \{9, 10, 13, 15, 26, 30, 31\}$$

$$\mathrm{FS}(28, y) = \{9, 13, 26, 27\}.$$

Those slices are derived from the system dependence graph presented in Figure 3.

2. Newly influenced nodes

In change analysis it is important to isolate newly influenced parts of the program. This is important for programers to determine the scope of influence and possible unintended influences, i.e., whether the "right" parts of the program are influenced, and that unexpected parts are not influenced. Newly influenced nodes can be found as follows:

$$\bigcup_{v \in V_1} \mathrm{FS}(n', v) - \bigcup_{v \in V_2} \mathrm{FS}(n', v)$$

Clearly, this set represents all nodes that were not influenced by node n in the original program P but are now influenced by n' in the modified program Q.

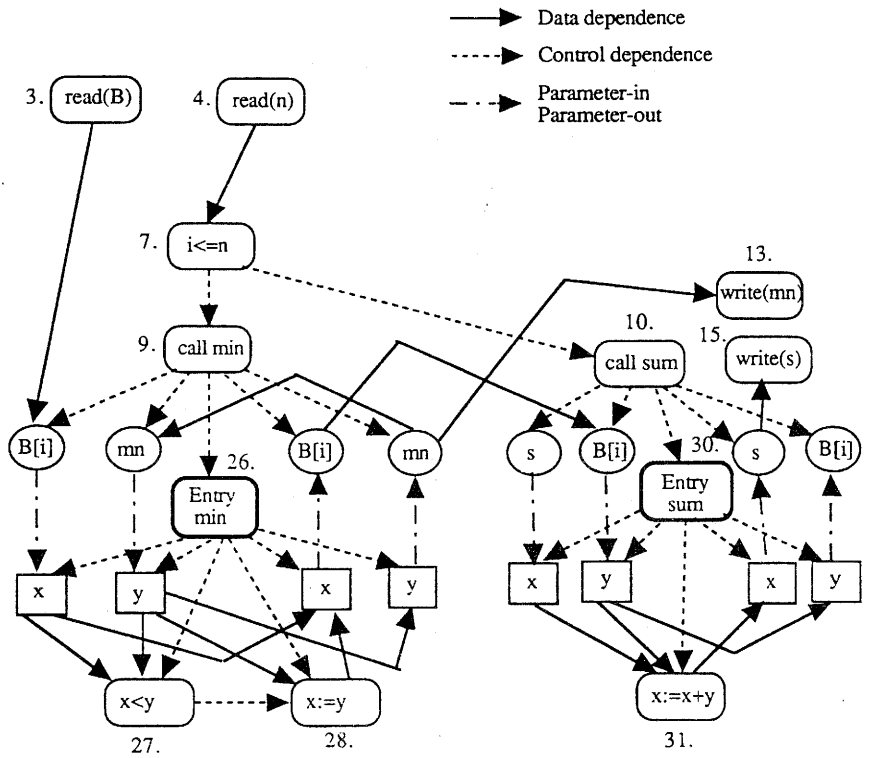

Fig. 3. System dependence subgraph of the modified program of Figure 1.

For our example, newly influenced area is computed as:

$$\mathrm{FS}(28, x) - \mathrm{FS}(28, y) = \{10, 15, 30, 31\}$$

This information may help a programer to determine possible unintended influences introduced by this modification, i.e., in this case the modification has a "suspicious" influence on procedure sum and "write(s)". Careful analysis by the programer may determine an erroneous modification. In the change analysis tool that has been developed, this information is displayed in the program text by highlighting statements corresponding to the newly influenced nodes (see Figure 4).

3. Not influenced nodes

Similarly to the newly influenced nodes, it is important to isolate those nodes that are no longer influenced by the modified node. This information is computed as follows:

$$\bigcup_{v \in V_2} \mathrm{FS}(n', v) - \bigcup_{v \in V_1} \mathrm{FS}(n', v)$$

Clearly, this set represents all nodes that were influenced in node n in the original program P but are not influenced any more by n' in the modified program Q. For our example, those nodes are computed as

$$\mathrm{FS}(28, y) - \mathrm{FS}(28, x) = \{13, 27\}.$$

This information helps a programer to understand that the output node "write(mn)" is not influenced any more by the modified node 28.

4. Input–Output relation

When the input–output relation analysis is applied to the modified program in our example, it is determined that there is a change in the input–output relation with respect to output variable mn.

Before Modification		After modification	
Output	Input	Output	Input
mn	B, n	mn	n

The change in the program functionality indicates to the programer a possible erroneous modification.

```
1.  program Main
2.    read(A);
3.    read(B);
4.    read(n);
5.    call init(mn, mx, s);
6.    i := 1;
7.    while i <= n do
8.      call max(A[i], mx);
9.      call min(B[i], mn);
10.     call sum(s, B[i]);
11.     i := i + 1;
12.   endwhile;
13.   write(mn);
14.   write(mx);
15.   write(s);
16. end;

17. procedure init(x, y, z)
18.   x := 0;
19.   y := 0;
20.   z := 0;
21. return;

22. procedure max(x, y)
23.   if x > y then
24.             y := x;
25. return;

26. procedure min(x, y)
27.   if x < y then
28.             y := x;
29. return;

30. procedure sum(x, y)
31.   x := x + y;
32. return;
```

Fig. 4. Highlighted newly influenced program statements.

5. Conclusions

The goal of change analysis presented in this paper is to analyze the possible difference in behavior between the original program and its modified version, i.e., this analysis tries to answer the question as to whether a program modification

leads to the desired change in program behavior, and, at the same time, it does not cause an undesirable change in behavior in other parts of the program. The purpose of program change analysis presented in this paper is twofold: (1) to detect possible erroneous modifications or anomalies related to those modifications, and (2) to support programers in the process of understanding the overall effect of modifications on the program. An experimental prototype of the change analysis tool which supports change analysis has been recently implemented for Pascal programs.

References

Bergeretti J.F. and Carre B.A. (1985): *Information-flow and data-flow analysis of while-programs.* — ACM Trans. Program. Languages and Systems, v.7, No.1, pp.37–61.

Ferrante J., Ottenstein K. and Warren J. (1987): *The program dependence graph and its use in optimization.* — ACM Trans. Program. Languages and Systems v.9, No.3, pp.319–349.

Gallagher K. and Lyle J. (1991): *Using program slicing in software maintenance.* — IEEE Trans. Software Engineering, v.SE–17, No.8, pp.751–761.

Horowitz S., Reps T. and Binkley D. (1990): *Interprocedural slicing using dependence graphs.* — ACM Trans. Programming Languages and Systems, v.12, No.1, pp.26–60.

Korel B. (1987): *The program dependence graph in static program testing.* — Information Processing Letters, v.24, No.2, pp.103–108.

Moriconi M. and Winkler T. (1990): *Approximate reasoning about the semantic effects of program change.* — IEEE Trans. Software Engineering, v.SE–16, No.9, pp.980–992.

Schneidewind N. (1987): *The state of software Maintenance.* — IEEE Trans. Software Engineering, v.SE–13, No.3, pp.303–310.

Weiser M. (1984): *Program slicing.* — IEEE Tran. on Software Engineering, v.SE–10, pp.352–357.

Yau S.S. and Kishimoto Z. (1987): *A Method for Revalidating Modified Programs in the Maintenance Phase.* — Proc. *COMPSAC–87 Conference*, pp.272–277.

Received December 7, 1992
Revised March 19, 1993

APPENDIX A

Sample Program Modifications

			Influenced
1. Assignment Statement			
a. Modification of the right–hand–side.			
Initial	Modified		
$y := x + z;$	$y := x - z;$	$V = \{y\},\ V_1 = V_2 = \{\ \}$	
b. Modification of the left–hand side.			
$y := x + z;$	$v := x + z;$	$V = \{y, v\},\ V_1\{v\},\ V_2 = \{y\}$	
c. Insertion			
$z := 1;$	$z := 1;$		
$s := x + 4;$	$y := x + z;$	$V = V_1 = \{y\},\ V_2 = \{\ \}$	
	$s := x + 4;$		
d. Deletion			
$z := 1;$	$z := 1;$		
$y := x + z;$	$s := x + 4;$	$V = V_2 = \{y\},\ V_1 = \{\ \}$	
$s := x + 4;$			
2. Test Node			
if $x < y$ then $z := 1;$	if $x = y$ then $z := 1;$	flow of control	
3. If–statement			
a. Insertion			
	if $x < y$ then $z := 1;$	flow of control	
b. Deletion			
$v := 1;$	$v := 1;$		
if $x < y$ then $z := 1;$	$w := x;$	$V = V_2 = \{z\},\ V_1 = \{\ \}$	
$w := x;$			

4. Procedure call

a. Insertion

empty statement $\{n'\}$ call sum$(x1, x2)$ $V = V_1 = \{\text{DEF}(n')\}$, $V_2 = \{\ \}$

b. Deletion

$\{n\}$ call sum$(x1, x2)$ empty statement $V = V_2 = \{\text{DEF}(n)\}$, $V_1 = \{\ \}$

Where, set V represents all influenced variables by the modification, set V_1 represents newly influenced variables, and set V_2 represents variables not influenced any more by the modified node.