JOURNAL OF APPLIED QUANTITATIVE METHODS

Reliability and Quality Control – Practice and Experience

SOFTWARE QUALITY VERIFICATION THROUGH EMPIRICAL TESTING

Ion IVAN[1]
PhD, University Professor, Department of Economic Informatics
Academy of Economic Studies, Bucharest, Romania
Author of more than 25 books and over 75 journal articles in the field of software quality management, software metrics and informatics audit. His work focuses on the analysis of quality of software applications.
E-mail: ionivan@ase.ro , **Web page:** http://www.ionivan.ro

Adrian PIRVULESCU
BRD - Groupe Société Générale, Bucharest, Romania
Bachelor Degree in Economic Computer Science from
Academy of Economic Studies, Bucharest, Romania

E-mail: u4adrian@yahoo.com

Paul POCATILU
PhD, University Lecturer, Department of Economic Informatics
Academy of Economic Studies, Bucharest, Romania

E-mail: paul.pocatilu@ie.ase.ro

Iulian NITESCU
Student of Faculty of Cybernetics, Statistics and Economic Computer Science,
Academy of Economic Studies, Bucharest, Romania

E-mail: iulian.nitescu@yahoo.com

Abstract: *Included is research that contributes to raising the quality of programs written in C/C++. Empirical testing was tackled. The empirical nature is characterized by the partial quality of its elements, the absence of systematic behavior in the process and the idea of random attempts at program behavior. Empirical testing methods are used the program as a black box view, as well as for the source code. Software testing at source level pursues raising the tree-like coverage associated with the code. There are known indicators for quantifying the test methods and measuring their efficiency upon programs by an empirical approach, as well as measuring the program quality level.*

Key words: *software testing; empirical measurements; software quality; indicators*

1. Introduction

This section presents fundamental concepts for software testing.

The objective for testing is to establish the unconformities between the specification and the final software product. The performance of the product is brought out by more comprehensive testing, if the product is well constructed. If the software product is not well constructed, the depth of testing brings out deficiencies in the source code.

Included in the test objectives is testing whether or not all the data is read. If we have a file with *n* articles, we have to check whether all the *n* articles are read. If we have a matrix with *m* lines and *n* columns, we check whether the *m* lines and *n* columns are used in calculations and if the *m* * *n* elements are also used in calculations. For partial testing, lists of elements that are not part of the processing process are constructed.

The test has to establish if the processing is done as stated in the specifications. The processing algorithm implies a series of steps. The test checks whether for each element in the file, matrix or sequence, all the steps are applied.

The testing methodology has the particularity of checking whether the processing is complete and correct. There is a series of control keys. Intermediate and final results are checked to see if the imposed criteria are satisfied.

Through the testing process all the unconformities between what the software product has to offer and what is written in its specification are established. In practice, the following situations are encountered:

- correct specification, correct software; the analyst understood the problem at hand and the formalized definition included in the specification is correct and complete; the programmers have followed the criteria in the specification and each part in the specification has a correspondence to a module or code sequence in the program;
- correct specification, incorrect software; the analyst understood the problem at hand and the formalized definition included in the specification is correct and complete, but the programmers did not follow the criteria in the specification;
- incorrect specification, correct software; the analyst did not understand the problem at hand and the specification definition is incorrect or incomplete; the programmers intervened upon the specification requirements, thus the final program is correct;
- incorrect specification, incorrect software; the analyst did not understand the problem at hand and the specification definition is incorrect or incomplete; the programmers have followed the criteria in the specification and each part in the specification has a correspondence to a module or code sequence in the program resulting in the program functioning incorrectly or incompletely.

In all cases, the testing re-establishes the truth, for the purpose of bringing the software product back on track towards the defined development, for the objective it was made.

Syntax errors appear during compilation. These are grouped on different levels - from warnings to fatal errors. Construction errors appear at link editing, runtime and result interpretation. Empirical testing has a partial behavior and is carried out in the following stages: analysis, projection, programming and module integration. Empirical testing is an auto-validation process as well as a global process. Disadvantages of empirical testing are

related to the work volume and the impossibility of improving over a certain limit of software quality.

Empirical testing is carried out by the program makers and then by the program users. As a starting point, it has input data and expected results. In the case where there are correlations between results (unvaried elements), empirical testing will have to emphasize the extent to which these correlations are made.

Empirical testing can be oriented towards a positive aspect, which emphasizes what the program computes, or the extent to which the program computes what it was meant to, or a negative aspect, in which case the control examples are chosen in a way that will bring out what the program doesn't do. Control examples that make up the empiric lot for testing reflect the testing process developer's capacity.

Empirical testing is necessary for software products sold on key, without clear documentation or those belonging to a class that doesn't include rigorous testing. Also, empirical testing is specific in those situations where the user always solves the desired problems with the same data structures and there are no variations to the size and data grouping accuracy. Empirical testing is not specific to software development, integration and reusing.

Any problem to be solved can be expressed by different levels of complexity. Each level has a specific volume and an input data structure. Empirical testing does not assume a gradual approach, complete to those levels, but the introduction of test examples, in the extent to which these appear in books, sale of products or by partially copying files from an extended database. The test examples are the actual problems to be solved.

If the multitudes of test examples are systematically constructed form a mosaic, the empirical test examples can be compared to a mosaic that is missing enough plates. However, the theme, subject and characters can be intuited or reconstructed.

The classical approaches looking at software testing from (Myers GJ, 1979; Beizer B, 1990), as well as Hutcheson ML 2003; Patton R. 2001), touch on the empirical nature of software testing.

In (Ivan I, Pocatilu P 1999), empirical testing was looked at from the programs as black boxes view, without taking into account the source code. In (Ivan I, Teodorescu L, Pocatilu P, 2000), research results are presented on the way in which software quality can be improved through testing.

This article develops empirical testing on software and shows the way in which, through empirical testing, the quality level of software is influenced. For measuring these levels, indicators are proposed for quantifying the testing process, as well as for measuring the quality level associated with the program.

In the article there is research done through the CNCSIS *Framework for the estimation of object oriented prototypes software testing costs* and *Models for the estimation of e-business application' s costs* grants.

Section 2 looks at empirical testing of software through the black box prism. In section 3 we look at empirical testing at the source code level. In section 4, methods for quantifying the testing process are presented. Section 5 is dedicated to measuring software quality. Section 6 presents a series of experimental results obtained by the authors. Conclusions for the research carried out are presented in section 7.

2. The black-box approach to programs

The program has to be viewed as a black-box (Fig. 1). From documentation, from the way in which interfaces are conceived, comes the input data structure. The way in which processing is done, what processing is done, what the secondary effects are, do not represent an essential part of empirical testing.

Figure 1. The program viewed as a black-box

The objective of empirical testing is showing that the program is good, working or not good, in which case the situations where the program does not offer required results are identified.

Empirical testing in the case of the black bock approach is concentrated upon the following three important areas:

- input data level
- processing level
- output data level (results).

The input data level is used to check if the program accepts as input data, data that defines the problem. Situations are identified where the demand of data exceeds the supply, the demand is less than the supply and in the best case, where there is equality between what the programs wants as input and what is offered.

At the processing level, all the algorithm steps are completely traversed, with an interruption at a certain point of execution or in different points, with connections between offered data and the point at which interruption occurs.

At output level – program results – what is wanted is the identification of structurally incomplete results, structurally complete results but incorrect, and also, the situation where results are good qualitatively without being able to further comment upon their effective correctitude.

In the case where the program is considered as a black box, it is imperative to carry out a study on the qualitative nature of the processing level, as there is no access to the program components, to the algorithms.

Numerous programs computing economical problems (accounting books, forecasts) for variables such as Gross National Produce (GNP), price (Pr), would only have to deal with strictly positive numbers. This is why the appearance of negative or null values in a forecast model for estimating GNP or Pr indicates the existence of some processing errors, or errors in the conceptual scheme of the model.

The black box associated with the program allows the bookkeeping of the functional part of the program - what processing is carried out, what processing is not carried out or is not correctly carried out.

For example, the program *PRELM* is considered, which carries out a series of processes leading to a matrix with a particular structure (Fig. 2).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & a & x & y & x+y \\ 0 & 1 & 0 & 0 & a & u & w & u+w \\ 0 & 0 & 1 & 0 & a & u & w & u-w \\ 0 & 0 & 0 & 1 & a & u^2 & w & 0 \\ 4a & -1 & \frac{1}{a} & 0 & a & 0 & 1 & 0 \\ 3a & -1 & \frac{2}{a} & 0 & a & 0 & 0 & 0 \\ 2a & -1 & \frac{3}{a} & 0 & a & 0 & 0 & 0 \\ a & -1 & \frac{4}{a} & 1 & a & 0 & 0 & 0 \end{bmatrix}$$

Figure 2. Resulting matrix for the *PRELM* program

For the following set of input data: **a=1**, **b=2**, **c=3**, **x=10**, **y=11**, **u=100** and **w=110**, if program P displays a table, the first things to check are:

- if the 4th order identity sub-matrix exists in the upper left corner of the matrix;
- if the 5th column is populated exclusively by the a value;
- if the 3rd order null sub-matrix exists in the bottom right corner;
- if -1 is on the 2nd column on lines 5 to 8;
- if on line 1, columns 6 and 7, there are values for x, respectively y, and in the 8th column there is their sum;
- if on line 2 and 3, columns 6 and 7 there is the u value, respectively w, on the 8th column, line 2 there is their sum, and their difference should be on column 8, line 3;
- if 1 is on the 4th column, line 5, and 0 on lines 6, 7 and 8;
- if on the 1st column, on lines 5, 6, 7 and 8 there is the a value multiplied by 4, 3, 2, respectively 1;
- if on column 3, lines 5, 6, 7 and 8 there are values for 1/a, 2/a, ¾, respectively 4/a.

Not knowing the program, the control examples are collected randomly, at most introducing the criteria for the currently appearing situation (in the processing) frequency. At first these control examples have small dimensions, to permit manual verifications. For example the function that computes the inverse of a matrix will be called with a 4 lines by 4 columns matrix as input data. In the absence of possibilities for verifying $A*A^{-1}=U$, where:

- A is the matrix to invert;
- A^{-1} is the inverse of A;
- U is the identity matrix (or unit matrix)

the example is either taken from a book where values for A and A^{-1} have been presented, or is constructed ad-hoc.

Empirical testing also takes into account examples of specific situations. Coming back to the inverse of a matrix function, the input is supplied as a matrix with two identical lines and the behavior of the program is examined.

In the case where program P is integrated into an application for current usage, empirical testing consists of constructing copies of files that are already being exploited and extracting/actualizing information from these copies (Fig. 3). In this case the current database is protected, eliminating the risk of uncontrolled deterioration of it.

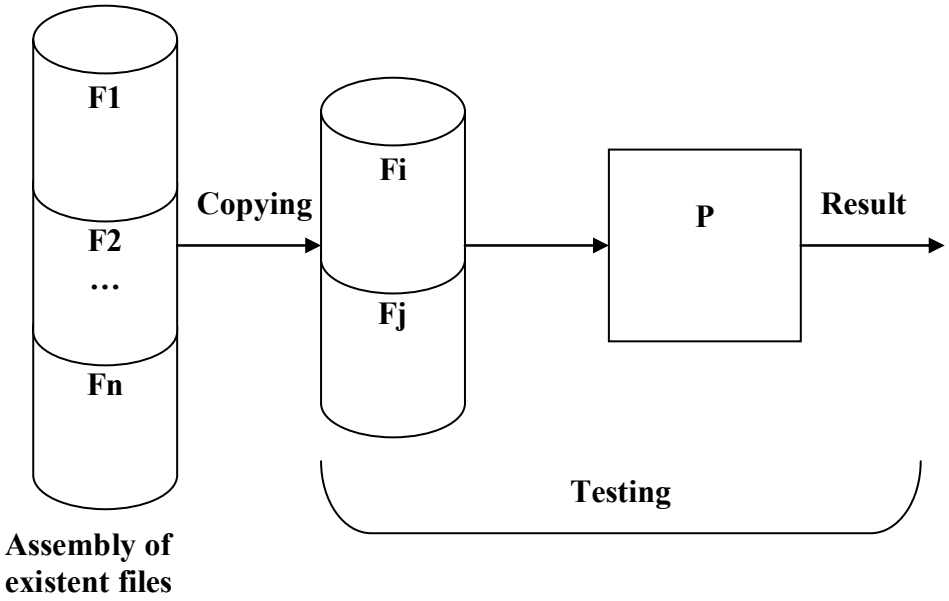

Figure 3. Empirical testing of a program to integrate into an application in current use

Empirical testing has to be carried out so that is can produce sufficient information that will lead to accepting or rejecting the use of this program for current use.

The program viewed as a black box is appreciated for carrying out the desired processing or not doing so. For the multitude of test examples considered, appreciation by *Yes* or *No* will suffice. Weighted percentages for correct results and total number of program runs are also described.

Consider the program that carries out certain tasks and the following test examples: E_1, E_2, ..., En. After running the program with test data, in *k* situations correct results were obtained, and in *k-n* situations incorrect results were obtained. If the *k-n* test examples, in which incorrect results were obtained, belong to a limited group of topologies, it can be concluded that the *k* situations cover a diverse and large enough area of different problem types. However, if the *k* examples belong to a single group of problems then it can be concluded that the program cannot cover a wide enough area of situations.

In the case where the specifications were not correctly understood, the basis for describing the algorithm was incorrect, but the program is accepted even though in reality the results are not 100% accurate because of erroneous foundations (specifications). When the results printed in books are incorrect, even though the program is correct, it may be rejected.

3. Structural approach to programs

Any construction can have a graph structure associated with it in which the nodes are instructions, sequences of instructions or procedures. The arcs show the succession of instruction execution, and succession of procedures.

There are situations where tree structures are associated with programs. In figure 4, there is a representation of the *PMIN4* program tree structure, whose objective is to determine the minimum value between a, b, c and d.

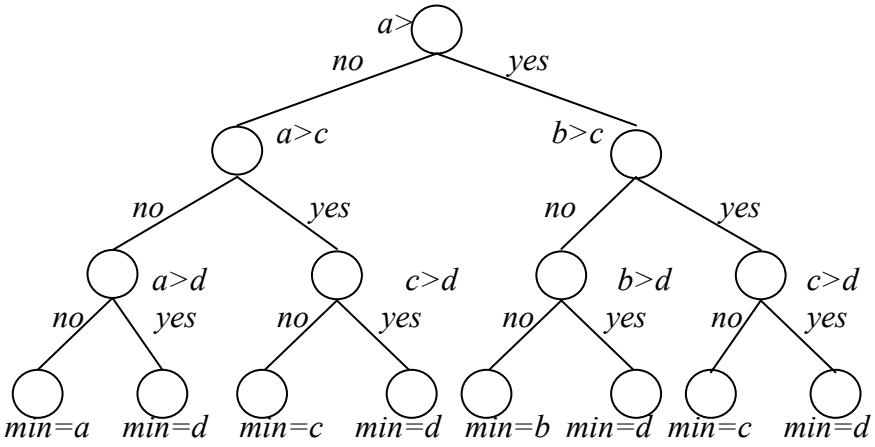

Figure 4. Tree structure associated with the *PMIN4* program for choosing the minimum element.

Consider the program *PGEN2M* written in C/C++ that generates two matrixes and prints them on the screen:

```
void main()
{
    int i,j,n,x[10][10],y[10][10];
        printf("n=");
        scanf("%i",&n);
        for (i=0;i<n;i++)
            for (j=0;j<n;j++)
            {
                    x[i][j]=i+j;
                    y[i][j]=i*j;
                }
for (i=0;i<n;i++)
                for (j=0;j<n;j++)
                printf("%i ",x[i][j]);
        for (i=0;i<n;i++)
    for (j=0;j<n;j++)
                printf("%i ",y[i][j]);
     }
```

The graph associated with *PGEN2M* is represented in figure 5.

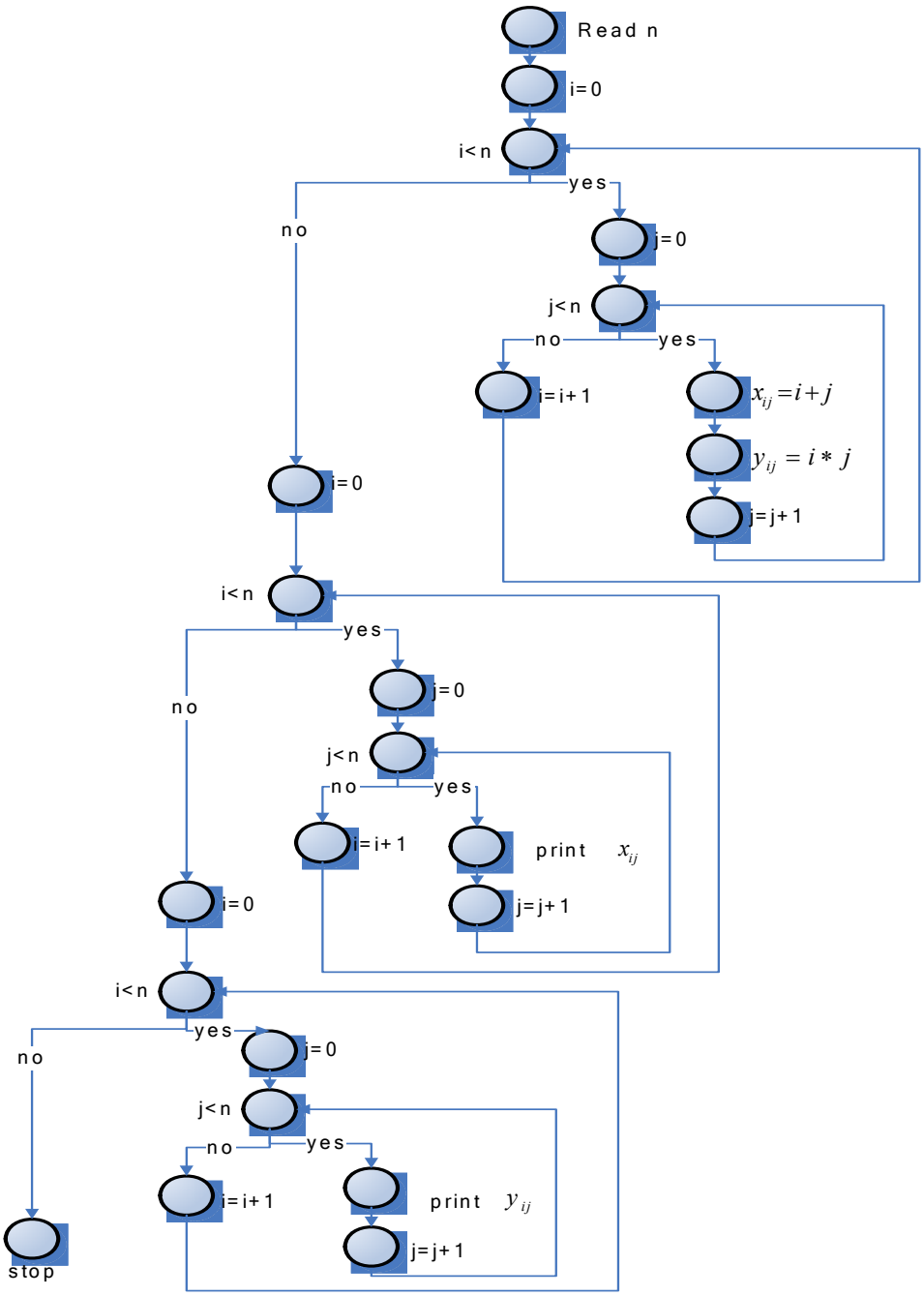

Figure 5. The graph associated with *PGEN2M* that generates two matrixes

To completely test the program means to define test data that will pass by every node in the graph at execution. In the above example, for $1 \leq n < 10$ all the nodes in the graph are visited. If the condition $1 \leq n < 10$ is true, for values $n<1$ and $n \geq 10$, the error part in figure 6 is traversed.

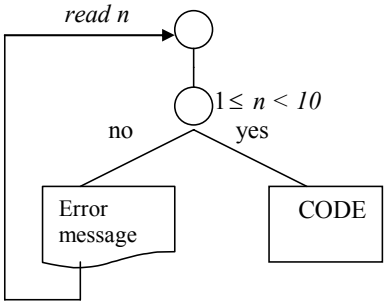

Figure 6. Program sequence for validating that variable n is within the limits

If a software product is organized on modules arranged in a tree-like structure (Fig. 7), to test the product would mean to define such sets of test data that will activate all branches of the tree, line by line.

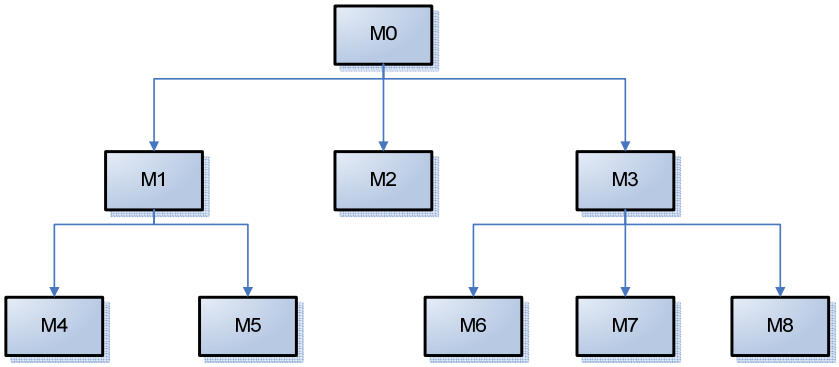

Figure 7. Software organized in modules

The multitude of paths for the program in figure 7 is:

- d1: {M0,M1,M4}
- d2: {M0,M1,M5}
- d3: {M0,M2}
- d4: {M0,M3,M6}
- d5: {M0,M3,M7}
- d:6 {M0,M3,M8}.

In the situation where a software product has a different structure than the tree-like one, through adequate transformations a tree structure is obtained. For example, in figure 9, by the multiplication of the M_4 module, a tree structure is obtained starting from the one in figure 8.

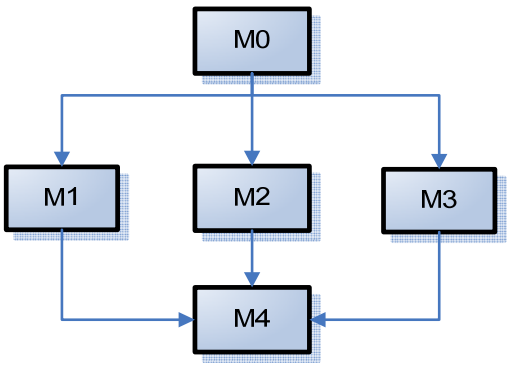

Figure 8. Graph structure of software product

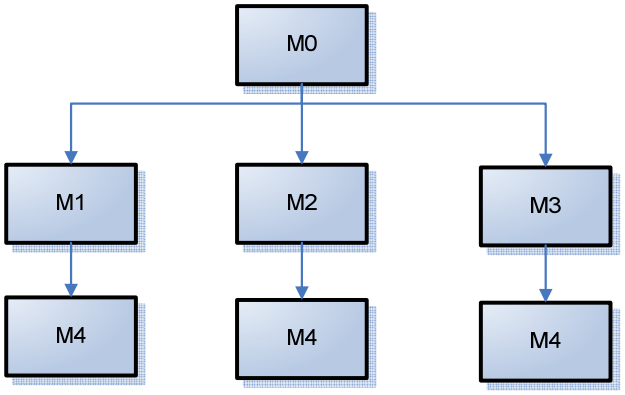

Figure 9. Tree structure obtained by the multiplication of the module

Even in the case of structures where there are conditional loops, a tree structure is assigned using conventions for processing the cyclic behavior. For example, the graph associated with the program that evaluates the expression $e = \min_{0 \le i \le 2} \{x_i\}$ is shown in figure 10.

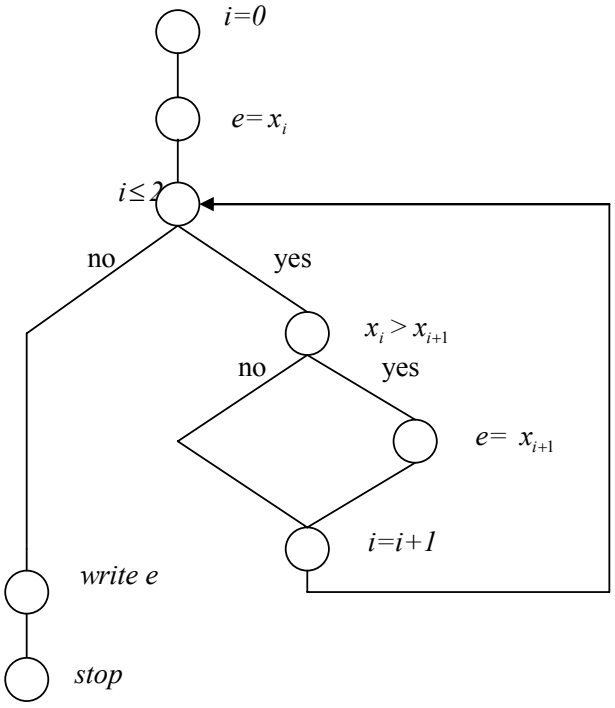

Figure 10. Graph with tree structure with known number of loops

For the graph with a known number of loops from figure 10, the tree structure associated with it is presented in figure 11. The arcs represented by dotted lines are the non activated instructions of the program.

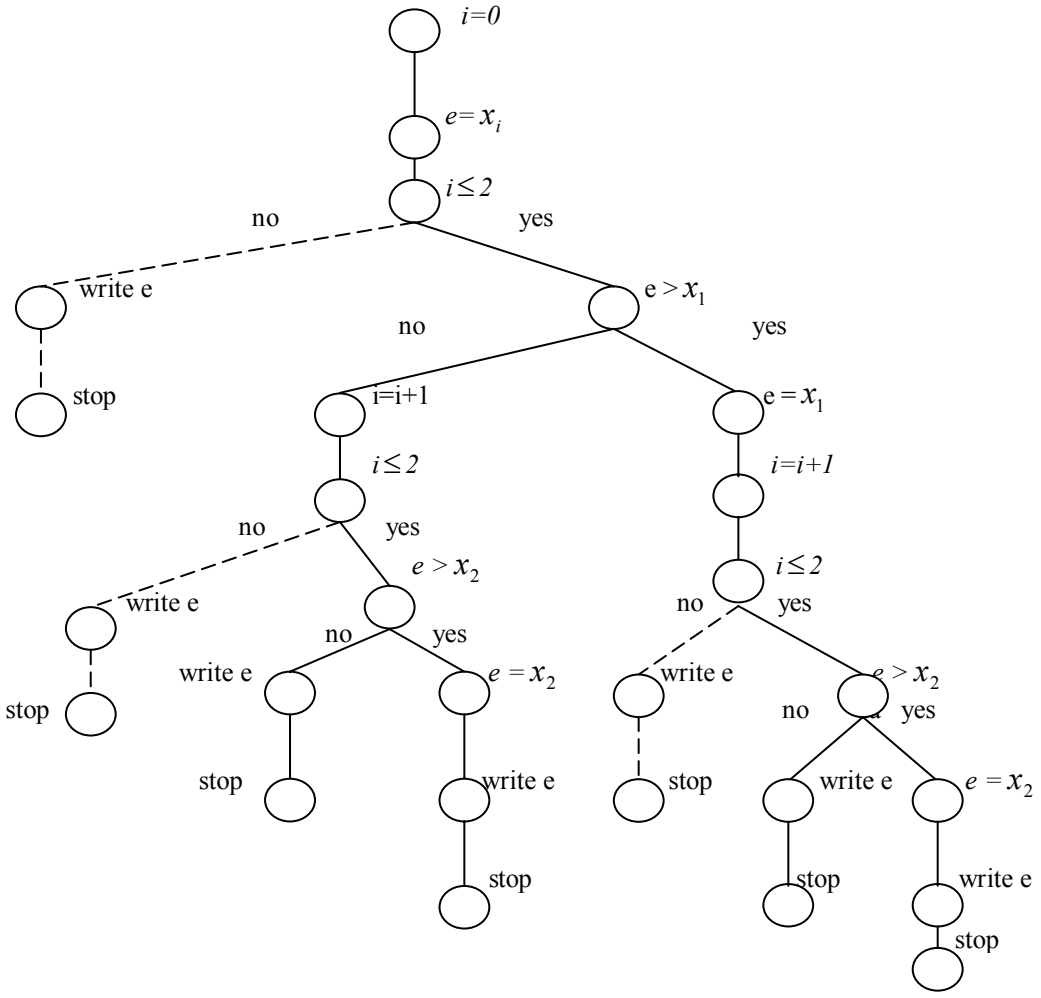

Figure 11. Tree structure corresponding to a finite number of loops

For an unknown number of loops, corresponding to the following sequence:

```
s=0;
for (i=0;i<n;i++)
      s+=x[i];
printf("%i",s);
```

the resulting structure is as shows in figure 12.

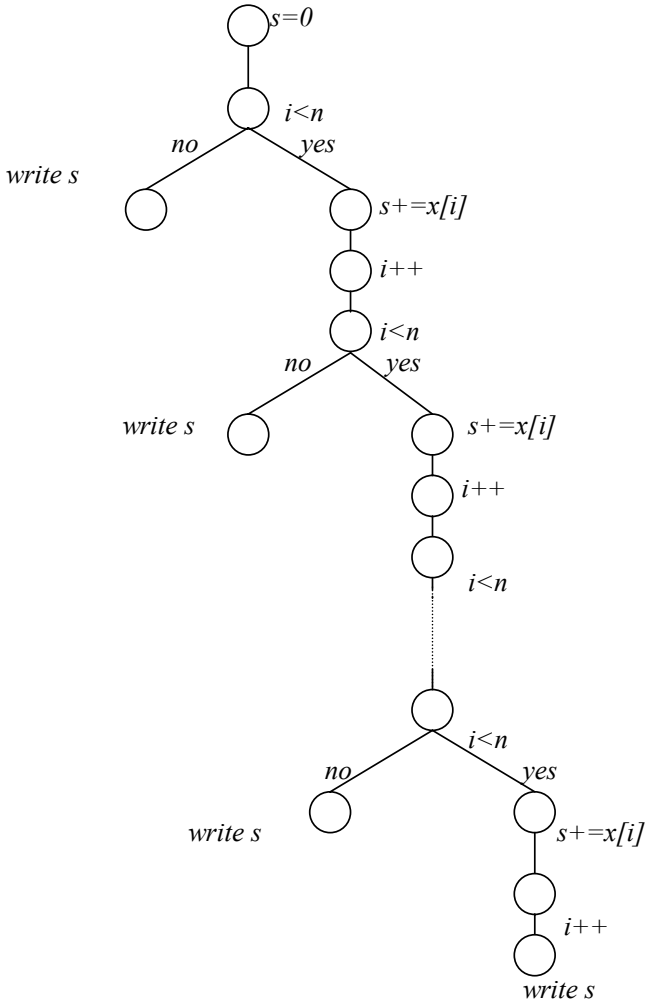

Figure 12. Tree structure representing an undefined number of loops

Working on a tree structure allows for carrying out more complete testing. Programs that solve more complex problems in economics, industry, transportation and commerce have a lot of levels, and the number of leafs for the tree structure is of the order of thousands, which in turn means generating test data in the order of thousands.

4. Quantifying testing processes

Consider a tree structure *S* associated with program P, organized on *k* levels. At the first level, the root level, there is a single node n_1. At the second level there are n_2 nodes, at the third level there are n_3 nodes, and so on. On the last level, where the leafs are, the number of nodes is n_k.

The total number of nodes N_T associated with the structure is therefore:

$$N_T = \sum_{i=1}^{k} n_i$$

The *number of data sets* N_{set} represents an important indicator because it offers an overview on the volume of processing specific to the testing.

The *diversity of test sets* D_{set} is an indicator that shows the measure of how the testing process has the capacity to cover an area as wide as possible of the tree. There is a maximum diversity and a relative diversity. The maximum diversity D_{max} represents the number of leafs in the tree. The relative diversity D_{rel} is defined as:

$$D_{rel} = \frac{D_{set}}{D_{\max}}$$

If D_{rel} converges to 1, it means that the testing process is complete.

The *testing level* L_t shows the position of the last node in the tree reached. L_{max} represents the level at which the leafs of the tree are situated. The relative level L_{rel} shows the degree of depth traversal of the tree:

$$L_{rel} = \frac{L_t}{L_{\max}}$$

The *degree of coverage* G_a shows the weighted percentage of nodes from the tree reached in the testing process N_a in the total number of nodes N_T of the tree structure:

$$G_a = \frac{N_A}{N_T}$$

The relative activation frequencies of leafs in the tree structure are tightly tied to the specifics of the problem to be solved.

For the tree structure in figure 13, the leafs *a, b, c, d* and *e* have the activation frequencies f_a, f_b, f_c, f_d and f_e.

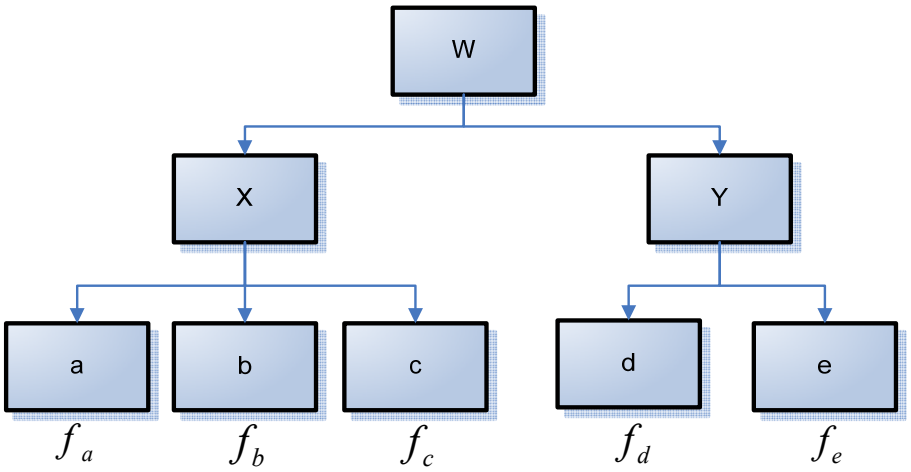

Figure 13. Tree structure organized on 3 levels

The *weighted coverage degree* G_{ap} is calculated as follows:

$$G_{ap} = \frac{\sum_{i=1}^{n} \alpha_i f_i}{\sum_{i=1}^{n} f_i}$$

where:

- f_i represents the activation frequency of node *i* in the tree structure, and
- α_i is 1 if node *i* is activated in the testing process and 0 otherwise.

This indicator assumes an analysis of the input data for the problem that is currently solved for each beneficiary. The testing process takes into account this information, however, by limiting resources it has it's own strategy that only in special cases overlaps with the real mode of exploitation of the program, as it happens for example when the testing is done for passenger flights software, for nuclear reactors, for space flights, where risks have a major significance.

There are a number of ways to interpret the test results. In a first variant, the concept all or nothing is used. With this method, after testing of program P with data sets $SD_1, SD_2, ..., SD_k$ is complete, the qualifier β_i is considered, with $\beta_i = 1$ (accepted) if for data set SD_i the program P being tested gives correct and complete results, and $\beta_i = 0$ as rejected, if after testing program P with SD_i data set there are errors. Table 1 is constructed:

Table 1. Test results using the accepted/rejected qualifier for the program

Data set	Qualifier β
SD_1	β_1
SD_2	β_2
…	…
SD_i	β_i
…	…
SD_k	β_k
Total	$T = \sum_{i=1}^{k} \beta_i$

The ratio $G_C = \frac{T}{k}$ is calculated which corresponds to the weighted percentage of the data sets whose results were correct. The data sets differ in structure and generate different effects with respect to the processing. If data set SD_i activates certain sequences in the program with complexity C_i, then the ratio $G_{CP} = \frac{\sum_{i=1}^{k} \beta_i C_i}{\sum_{i=1}^{k} C_i}$ is calculated, which allows for a better overview of the test process.

These indicators are used in section 6 for establishing the way in which program *PEC2* was tested.

5. Software quality planning

The experience in using and developing software products imposes rules on planning for its quality. There are numerous software products currently in use.

Consider *m* domains of usage: $D_1, D_2, ..., D_m$. Domain D_i contains programs $P_{i1}, P_{i2}, ..., P_{ir}$ which are permanently used. Each has its own quality level, that is $IQ_{i1}, IQ_{i2}, ..., IQ_{ir}$. IQ_{ij} is the aggregate indicator for the quality of program P_{ij}. While using program P_{ij} the advantages and disadvantages the program has are highlighted due to the initial quality level the program was endowed with. Consider:

- IQ_{ij}^{ef} - the effective quality level of program P_{ij}
- IQ_{ij}^{pl} - the planned quality level of program P_{ij}
- IQ_{ij}^{ne} - the user required necessary quality level for program P_{ij}.

If $IQ_{ij}^{ef} > IQ_{ij}^{pl} > IQ_{ij}^{ne}$ while the program is in use, the user has a high level of satisfaction, thus program P_{ij} is offering special facilities or special behaviors, above the expectations of the user.

If $IQ_{ij}^{ef} > IQ_{ij}^{pl} = IQ_{ij}^{ne}$, the user is satisfied that the product is functioning with $IQ_{ij}^{ef} > IQ_{ij}^{ne}$. If $IQ_{ij}^{ef} > IQ_{ij}^{pl} < IQ_{ij}^{ne}$ it means that at the planning stage, the user's needs have not been fully studied. If $IQ_{ij}^{ef} < IQ_{ij}^{pl} < IQ_{ij}^{ne}$, then the situation is at it's worst.

It follows that experience comes in to correct levels IQ^{pl} so that $(IQ_{ij}^{pl})_t < (IQ_{ij}^{pl})_{t+1} < ... < (IQ_{ij}^{pl})_{t+k}$. When a new software product is constructed for use in domain D_i at a moment $t+k$, the demands for $(IQ_{ij}^{pl})_{t+k}$, $j = 1,2,...,r$ are analyzed, and decisions are made to work in planning with average or maximum levels.

In the hypothesis where maximum levels are used and in the case where $IQ_{ij}^{ef} \geq IQ_{ij}^{pl} \geq IQ_{ij}^{ne}$, for products $P_{i1}, P_{i2}, ..., P_{ir}$, the quality characteristics $C_1, C_2, ..., C_r$ are considered, which are measured as in table 2:

Table 2. Measuring effective levels of quality characteristics

Program	C_1	C_2	...	C_j	...	C_r
P_{i1}	·	·	·	·	·	·
P_{i2}	·	·	·	·	·	·
...	...	...	...	...	...	...
P_{ik}	·	·	·	α_{kj}^{i}	·	·
...	...	...	...	...	...	...
P_{ir}	·	·	·	·	·	·

In table 2, variables α_{kj}^{i}, represent the level of quality characteristics C_j for program P_{ik}. The maximum levels are chosen for characteristics $\alpha_j^{\max} = \max_{i \le k < r} \{\alpha_{kj}^{i}\}$ where *r* represents the number of programs from domain D_i. It follows that the planned levels for the new product are for characteristics $C_1, C_2, ..., C_r$, respectively $\alpha_1^{\max}, \alpha_2^{\max}, ..., \alpha_r^{\max}$. Natural selection principles also apply in the software field.

For the problem on calculating the inverse of a matrix, characteristics C_1 – complexity and C_2 – robustness are considered, together with programs PX_1, $PX_2, ..., PX_{10}$, for which data is collected in the following tables:

Table 3. Marks associated with the qualifiers associated with characteristics C_1 and C_2

Qualifier α_{kj}^{i} C1	**Qualifier C2**	**Marks**
very high	very good	10
high	good	7
average	satisfying	5
low	unsatisfying	2

Qualifiers used for complexity and robustness are presented in table 4.

Table 4. Quality characteristics associated with the program for inverting a matrix

Program	C_1	C_2
PX_1	10	7
PX_2	5	7
PX_3	7	7
PX_4	2	5
PX_5	5	5
PX_6	10	10
PX_7	7	10
PX_8	5	7
PX_9	7	2
PX_{10}	7	5

Consider the programs for which the qualifier for complexity is very high or high, and also for which the qualifier for robustness if very good or good, and then calculate the mean complexity, and mean robustness. These then become planned levels for the type of program that deals with inverting matrixes.

For the importance coefficient $p_1=0.4$ associated to complexity and $p_2=0,6$ associated to robustness, the aggregate indicator for quality C_a can be calculated for the ten programs, from which the results in table 5 can be obtained.

Table 5. Aggregate indicator for quality

Program	C_1	C_2	$C_a = p_1 * C_1 + p_2 * C_2$
PX_1	10	7	8,2
PX_2	5	7	6,2
PX_3	7	7	7
PX_4	2	5	3,8
PX_5	5	5	5
PX_6	10	10	10
PX_7	7	10	8,8
PX_8	5	7	6,2
PX_9	7	2	4
PX_{10}	7	5	5,8

On the basis of the aggregate indicator for quality the planned level for program quality is identified. This means that inside software companies, the program behavior is noted, so that classes can be made as homogeneous as possible on different problem types, and to be able to obtain the planned levels.

In the case of some products, their behavior with users is noted, measurements of effective characteristics are taken, and thus, levels that become planned levels are associated through similarities to other applications with the same level of complexity or that are in the same area of usage.

6. Experimental results

For easing the development of the experimental part, the well known problem of solving a second order equation has been chosen. The testing of any other program is done in a similar way.

The program considered reads three floating point numbers a, b and c. These are interpreted as the coefficients for any equation of order <=2 of the form $ax^2 + bx + c = 0$. Program *PEC2* calculates the solutions to the equation in the case that they exist, complex or real, or shows a message corresponding to different situations - if the problem doesn't have a solution or is undetermined.

The code for program *PEC2* is written in C/C++:

```
#include "stdafx.h"

#include <stdio.h>
#include <math.h>
void ecuatie(float a,float b,float c,float *x1,float *x2,float *re1, float *im1,
                                    float *re2,float *im2,int *path)
{
        float delta;
        if(a==0)
                if(b==0)
                        if(c==0)
                                *path=1;
                        else
                                *path=2;
                else
                        if(c==0)
                        {
                                *x1=0;
```

```
                        *path=3;
                }
                else
                {
                        *x1=-c/b;
                        *path=4;
                }
        else
                if(b==0)
                        if(c==0)
                        {
                                *x1=0;
                                *x2=0;
                                *path=5;
                        }
                        else
                                if (-c/a>0)
                                {
                                        *x1=sqrt(-c/a);
                                        *x2=-sqrt(-c/a);
                                        *path=6;
                                }
                                else
                                {
                                        *im1=sqrt(c/a);
                                        *im2=-sqrt(c/a);
                                        *path=7;
                                }
                else
                        if(c==0)
                        {
                                *x1=0;
                                *x2=-b/a;
                                *path=8;
                        }
                        else
                        {
                                delta=b*b-4*a*c;
                                if(delta>0)
                                {
                                        *x1=(-b+sqrt(delta))/(2*a);
                                        *x2=(-b-sqrt(delta))/(2*a);
                                        *path=9;
                                }
                                else
                                        if(delta==0)
                                        {
                                                *x1=-b/(2*a);
                                                *x2=*x1;
                                                *path=10;
                                        }
                                        else
                                        {
                                                *re1=-b/(2*a);
                                                *im1=(sqrt(-delta))/(2*a);
                                                *re2=-b/(2*a);
                                                *im2=(-sqrt(-delta))/(2*a);
                                                *path=11;
                                        }
                        }
}

void main()
{
        char s[200], s1[200];
        float a,b,c,x1,x2,re1,re2,im1,im2;
        int path;
        FILE *f, *f1;
        printf("Nume fisier de intrare: ");
        gets(s);
```

```
printf("Nume fisier de iesire: ");
gets(s1);
f=fopen(s, "r");
f1=fopen(s1, "w");
while  (!feof(f))
{
        fscanf(f,"%f %f %f", &a, &b, &c);
        ecuatie(a,b,c,&x1,&x2,&re1,&im1,&re2,&im2,&path);
        switch (path)
        {
        case 1: fprintf(f1, "Nedeterminare\n");break;
        case 2: fprintf(f1, "Ecuatia nu are solutii\n");break;
        case 3: fprintf(f1, "%f\n",x1);break;
        case 4: fprintf(f1, "%f\n",x1);break;
        case 5: fprintf(f1, "%f %f\n",x1,x2);break;
        case 6: fprintf(f1, "%f %f\n",x1,x2);break;
        case 7: fprintf(f1, "%fi %fi\n",im1,im2);break;
        case 8: fprintf(f1, "%f %f\n",x1,x2);break;
        case 9: fprintf(f1, "%f %f\n",x1,x2);break;
        case 10:fprintf(f1, "%f %f\n", x1,x2);break;
        case 11:printf(f1,"%f+%fi%f+%fi\n",re1,im1,re2,im2);break;
        }
}
fclose(f);
fclose(f1);
}
```

The tree structure in figure 14 is associated to program *PEC2*:

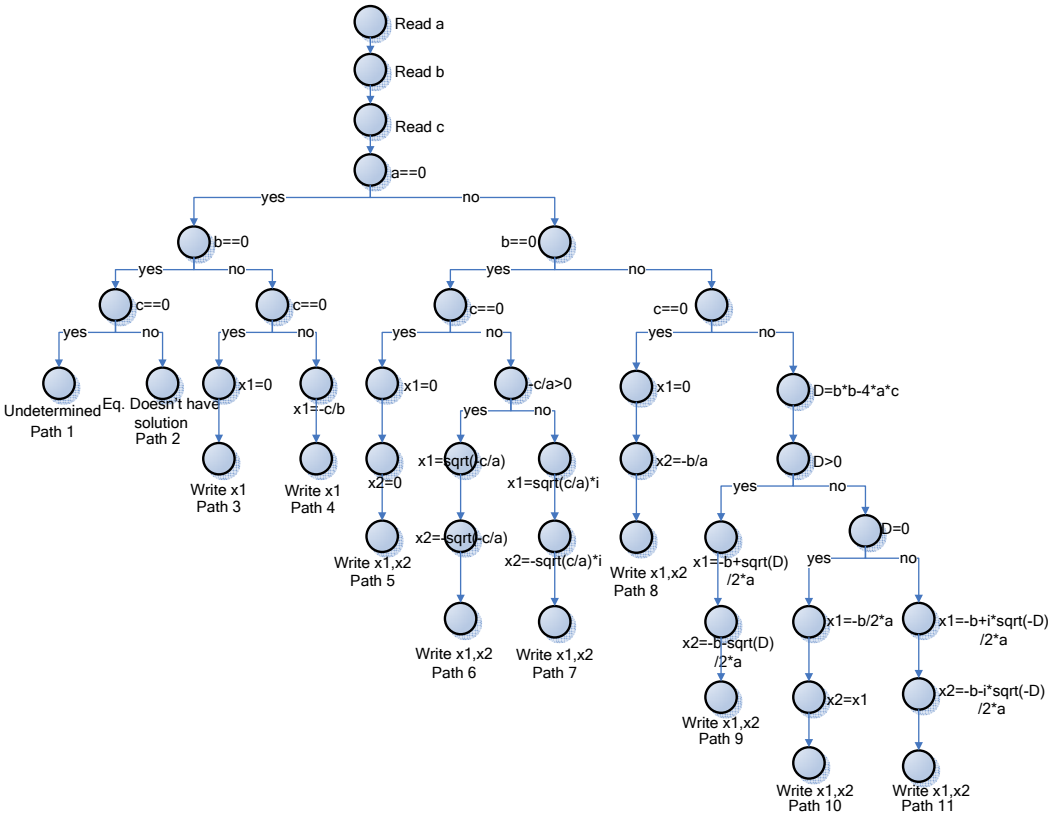

Figure 14. Tree structure associated with program *PEC2* for calculating the solution to a second order equation.

The tree structure has *k* =12 levels. The number of nodes for each level is n_1=1; n_2= 1; n_3= 1; n_4=1; n_5=2; n_6=4; n_7=8; n_8=7; n_9=6; n_{10}=5; n_{11}=3; n_{12}=2. The total number of nodes is N_T=31 nodes and there are 11 paths corresponding to the 11 leafs.

The complexities of the paths from the root to the leafs are calculated using Halstead's formula: $C = n_1 \log_2 n_1 + n_2 \log_2 n_2$, with n_1= number of operands and n_2 = number of operators. The total complexity C_T, is calculated as a sum of the complexities for all the paths. In table 6 there are complexities calculated for each path.

Table 6. Complexities of paths for program *PEC2*

Path	Complexity
$path_1$	48
$path_2$	52,53
$path_3$	71,27
$path_4$	91,36
$path_5$	91,13
$path_6$	150,84
$path_7$	145,04
$path_8$	112,11
$path_9$	282,21
$path_{10}$	250,92
$path_{11}$	413,19
Total	**1708,6**

For the test process to be complete, it is necessary for the relative diversity of the data sets D_{rel} to be 1, which means that the diversity of test sets D_{set} has to cover the maximum diversity corresponding to the number of leafs. For example, if the test data sets from table 7 are considered, these cover the whole tree area, so that the degree of coverage G_a is 100%. In this case, the degree of depth traversal L_{rel} is equal to 1.

Table 7. Test data sets associated to program *PEC2*

Data set	Associated values
SD_1	(0,0,0)
SD_2	(0,0,7)
SD_3	(0,5,0)
SD_4	(0,2,4)
SD_5	(1,0,0)
SD_6	(1,0,-3)
SD_7	(2,0,1)
SD_8	(1,2,0)
SD_9	(1,-1,-2)
SD_{10}	(1,2,1)
SD_{11}	(1,1,1)

For calculating the weighted degree of coverage G_{ap}, the activation frequencies of the tree leafs have to be settled. For this, state variables are introduced into the program,

which count the activation of leaf nodes. These indicators show to what extent the program testing is conclusive.

To accomplish the program testing, the test data sets are read from a text file as input, and the obtained results are saved in an output text file. A complete test for an application is impossible to accomplish both theoretically and practically. Tests that maximize the probability of discovering important processes in the application have to be devised.

To accomplish the program testing, the test data sets are read from a text file as input, and the obtained results are saved in an output text file. The obtained results by running the program with the data from table 7 are presented in figure 15.

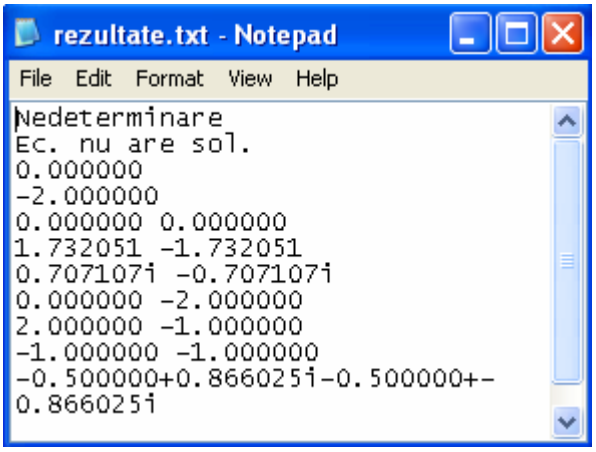

Figure 15. Results obtained from running PEC2 with the considered test data

The test is run, and table 8 results:

Table 8. Results from testing program *PEC2*

Data set	Qualifier β	Complexity
SD_1	1	48
SD_2	1	52,53
SD_3	1	71,27
SD_4	1	91,36
SD_5	1	91,13
SD_6	1	150,84
SD_7	1	145,04
SD_8	1	112,11
SD_9	1	282,21
SD_{10}	1	250,92
SD_{11}	1	413,19
Total	**11**	**1708,6**

The weighted indicator G_C is calculated for data sets whose results have been correct using the qualifier accepted/rejected and the indicator G_{CP} on the basis of path

complexity, thus the level of program quality results. On the basis of the results we have $G_C = G_{CP} = 1$, which means that the test is correct and complete.

7. Conclusions

The software quality of input data, results, processes, is emphasized by testing. Empirical testing has a special role because it is the only way to check the quality of very complex software applications. At present there is a lack of software to assist the symbolic testing for such applications. The correctness is automatically emphasized for very restrictive classes of applications.

Empirical testing is the practitioner's instrument for seeing how good or how bad a software product, database or the result of his/her application is. The only thing that needs to be done is to find techniques and methods based on empirical testing, that are built in such a way as to maximize the efficiency of the software testing process.

The empirical nature is characterized by partial quality of its elements, the absence of systematic behavior in the process and the idea of random attempts of program behavior.

The accumulated experience and joining the effort with the test results are the fundamentals of improvement in empirical testing. The dynamic behavior is concerned with the number of data sets, their diversity, and the summation of transformations that are produced in the testing process to obtain as much information as possible about the quality of the application.

In the future, experimental results and data series from tests will have to be included in models for software and database costs.

Empirical testing is at the hands of all users. Beta versions of software products are empirically tested on a very large user base.

Empirical testing was used in research for an informatics system for crediting operations in a bank. This system had a very high complexity.

References

1. Beizer, B. **Software Testing Techniques – Second Edition,** Van Nostrad Reinhold, New York, 1990
2. Cazan D., Ivan I. **Metrici de calitate ale sistemelor informatice,** Revista Informatica Economica, vol. 8, nr. 3, pp.123—128, September 2004
3. Chan, W. K., Chen T. Y., Tse, T. H. **An Overview of Integration Testing Techniques of Object-Oriented Programs,** Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science, (ICIS 2002), Mt. Pleasant, Michigan, 2002
4. Chen, T.Y., Yu, Y.T. **On the expected number of failures detected by subdomain testing and random testing,** IEEE Transactions on Software Engineering, vol. 22, No. 2, February 1996, pp 109--119.
5. Hutcheson M.L. **Software Testing Fundamentals: Methods and Metrics,** John Wiley & Sons, 2003
6. Ivan, I, Pocatilu, P. **Testarea Software Orientat Obiect,** Editura INFOREC, Bucharest, 1999
7. Ivan, I., Popescu, M., Sinioros, P., Simion, F. **Metrici Software,** Editura INFOREC, Bucharest, 1999
8. Ivan, I., Teodorescu, L., Pocatilu, P. **Cresterea calitatii software prin testare,** Revista Q-media, vol. 2, nr. 5, pp. 20—24, 2000

9. Ivan, I., Pocatilu, P., Stanca, C., Mihai, T. **Data Certification,** Proceeding of the 2000 MIT Conference on Information Quality, 2000
10. Ivan, I., Pocatilu, P., Sinioros, P. **Testarea aplicatiilor de e-business,** Proceedings of SIMPEC 2000 International Conference, vol. 2, Brasov, Transylvania University, pp. 172—177, November 2000
11. Ivan, I., Toma, C. **Testarea interfetelor om-calculator,** Revista Romana de Informatica si Automatica, vol. 13, nr. 2, pp. 22—29, 2003
12. Ivan, I., Pocatilu, P. **Testarea automata a aplicatiilor software specializate,** Revista Informatica Economica, vol. VIII, nr. 2, pp.116—120, June 2004
13. Ivan, I., Boja, C. **Metode statistice in analiza software,** Editura ASE, Bucharest, 2004
14. McGregor, J.D. **Quality Assurance: An overview of testing,** Journal of Object-Oriented Programming, vol.9, No. 8, 1997
15. McGregor, J.D. **Testing – Is It a Phase, an Activity or a Lifestyle?**, Journal of Object-Oriented Programming, Vol. 13, No. 1, March/April 2000, pp. 36--39
16. Myers, G. J.**The Art of Software Testing,** John Wiley & Sons, New York, 1979
17. Myers, G. J. **The Art of Software Testing, Second Edition, Revised and Updated by Tom Badgett and Todd M. Thomas with Corey Sandler,** John Wiley & Sons, 2004
18. Olaru, M. **Managementul calitatii,** Editura Economiaă, Bucharest, 1999
19. Pocatilu, P., Ungureanu, D. **Managementul procesului de testare software,** Revista Romana de Informatica si Automatica, vol. 13, nr. 2, 2003, pp. 15-21
20. Pocatilu, P. **Costurile testarii software,** Editura ASE, Bucharest, 2004
21. Pocatilu, P. **Testarea programelor Java cu JUnit,** Informatica Economica, vol. IX, nr. 2(34), 2005, pp. 51-55, June 2005
22. Pocatilu, P. **Using test cases in distributed application testing,** in Proceedings of SIMPEC 2005 International Conference, Brasov, May 20-21, 2005, pp. 255--261
23. Patton, R. **Software testing,** SAMS Publishing House, USA, 2001

[1] Ion IVAN has graduated the Faculty of Economic Computation and Economic Cybernetics in 1970, he holds a PhD diploma in Economics from 1978 and he had gone through all didactic positions since 1970 when he joined the staff of the Bucharest Academy of Economic Studies, teaching assistant in 1970, senior lecturer in 1978, assistant professor in 1991 and full professor in 1993. Currently he is full Professor of Economic Informatics within the Department of Economic Informatics at Faculty of Cybernetics, Statistics and Economic Informatics from the Academy of Economic Studies. He is the author of more than 25 books and over 75 journal articles in the field of software quality management, software metrics and informatics audit. His work focuses on the analysis of quality of software applications. He is currently studying software quality management and audit, project management of IT&C projects. He received numerous diplomas for his research activity achievements. For his entire activity, the National University Research Council granted him in 2005 with the national diploma, Opera Omnia.
He has received multiple grants for research, documentation and exchange of experience at numerous universities from Greece, Ireland, Germany, France, Italy, Sweden, Norway, United States, Holland and Japan.
He is distinguished member of the scientific board for the magazines and journals like:
- Economic Informatics; - Economic Computation and Economic Cybernetics Studies and Research; - Romanian Journal of Statistics
He has participated in the scientific committee of more than 20 Conferences on Informatics and he has coordinated the appearance of 3 proceedings volumes for International Conferences.
From 1994 he is PhD coordinator in the field of Economic Informatics.
He has coordinated as a director more than 15 research projects that have been financed from national and international research programs. He was member in a TEMPUS project as local coordinator and also as contractor in an EPROM project.