Control Structure Oriented Software Testing

Prof. Ion IVAN PhD., Assist. Paul POCATILU
Economy Informatics Department, Academy of Economic Studies Bucharest
Anca Andreea IVAN, New York University, USA

This paper defines techniques for testing homogeneous procedures that form complex software products. First, we establish the volume of test data for each control structure. Then, we analyze the effects of the testing process for program sequences that include linear structures, alternative structures, including compound ones, repetitive structures, embedded structures, and procedures.
Keywords: *software testing, control structures, testing code, test data.*

1 Testing linear procedures

Linear procedures include sequences of instructions that are executed one after another one. There are no unconditional jumping instructions, no conditional jumping instructions, and no multiple alternative structures. A linear procedure looks like:

```
type procedure_name(type p1, type
p2,…, type pk)
{     I1
      I2
      I3
      …
      Ik
      …
      In
      return (return_value);
}
```

The instructions are executed in order, as shown in figure 1.

Fig. 1. Graph associated to the linear structure

The next step is to construct the matrix with instruction dependences (table 1). The elements in the matrix a_j are equal to '*' when there is no dependence between instruction i and instruction j. If instruction i depends on instruction j, the element of the matrix is set to 'd'.

Table 1. The matrix with instruction dependences

	I_1	I_2	…	I_{n-1}	I_n
I_1	*	*		*	*
I_2	d	*		*	*
…					
I_{n-1}	*	d		*	*
I_n	*	d	…	d	*

By analyzing the matrix with instruction dependences, it results that instruction I2, I3 and In-1 depend on the previous instructions, respectively I1, I2, and I3. In addition, the instruction In indirectly depends on the previous instructions I1, I2, …, In-1. For example, let's consider the ***Calcul()*** procedure that evaluates the expression $e=(a+b-a+5+8)^2$, using a series of intermediary subexpressions.

```
int Calcul(int a,int b,int
c,int d, int e, int f)
{
   a=3;          //    I1
   b=5;          //    I2
   c=a+5;        //    I3
   d=b-a;        //    I4
   f=c+8;        //    I5
   e=d=f;        //    I6
   return e*e;   //    I7
}
```

Table 2 shows the parameter domain for the ***Calcul()*** function. The matrix with instruction dependences is given in table 3.

Table 2. Parameter domain for the Calcul function

Parameter	Type	Domain
a,b,c,d,e,f	int	-2147483648…+2147483647

Table 3. The matrix with instruction dependences for the Calcul function

	I1	I2	I3	I4	I5	I6	I7
I1	*	*	*	*	*	*	
I2	*	*	*	*	*	*	
I3	d	*	*	*	*	*	
I4	d	d	*	*	*	*	
I5	d	d	d	*	*	*	
I6	d	d	*	d	d	*	
I7							

The cyclomatic complexity for the ***Calcul()*** procedure is CV(G)=1. There is only one succession of arcs in the procedure because there are no alternative or repetitive instructions. This leads to the existence of only one test data to execute all instructions.

A significant data set is constructed based on the matrix with instruction dependences. This set should highlight the fact that the procedure leads to correct and complete results (table 4).

Table 4. The set of test data for the ***Calcul()*** procedure

Data	a	b	C	d	f	e	e*e
D_1	0	0	5	0	13	13	169
D2	0	-13	5	-13	13	0	0
D3	1	1	6	0	14	14	196
D4	1	-1	6	-2	14	12	144
D5	-1	1	4	2	12	14	196

When analyzing and designing the test cases, the procedure testing cost is given by the expression:

$$C = C_1 + C_2 + C_3$$

where C_1 is the cost to build the table of variable domains, C_2 is the cost to build the matrix of dependences and C_3 is the cost to build the testing tests.

All these costs depend on the activity durations, the number of involved persons, and their salaries.

The costs of the test sets depend on the data volume and the way they were obtained. The cost of obtaining a test set by analyzing the problem specifications and the algorithm that solves the problem is bigger than the cost of obtaining the test set by analyzing the limit values.

The cost of obtaining the test sets, C3, is given by the formula: $C_3 = \sum_{i=1}^{N} c_i V_i$, where c_i is the cost of obtaining the test set for volume V_i. After running experiments and obtaining measurements, one can unites the resulted data such that the costs Ci and the volumes Vi are the results of work normalization.

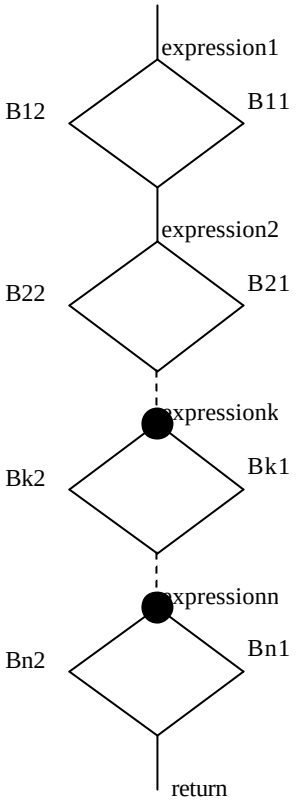
2. Testing the alternative procedures

The procedures that mainly contain ***if-then-else*** instructions and implement alternative structures are called ***if-then-else procedures*** or ***alternative procedures.***

Figure 2 a) illustrates the structure of an alternative procedure, and figure 2 b) presents the graph associated to such a procedure. Figure 3 presents the tree structure associated to it.

```
type procedure_name(type1 p1,
                    type2 p2,…,
                    typek pk)
{
        if(expression1)
        {
                B11
        }
        else
        {
                B12
        }
        …
        if(expressionk)
        {
                Bk1
        }
        else
        {
                Bk2
        }
        …
        if(expressionn)
        {
                Bn1
        }
        else
        {
                Bn2
        }
        return (return_value);
}
```

a)



b)

Fig. 2. The structure and the graph associated to an alternative procedure

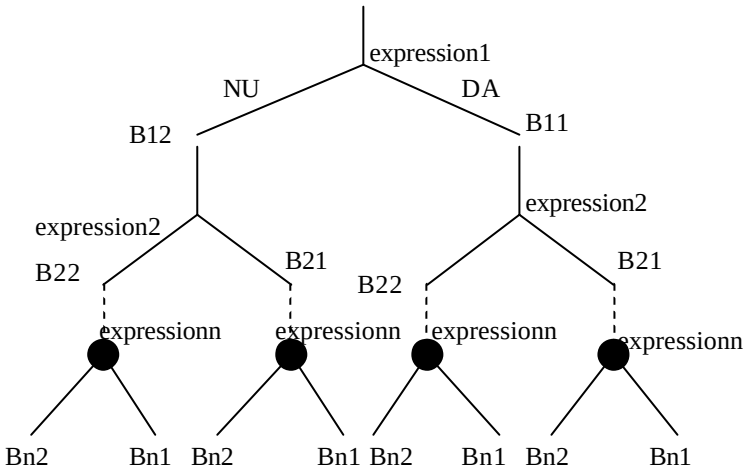


Fig. 3. Tree structure associated to an alternative procedure

The number of paths in an alternative procedure that contains n ***if-then-else*** instructions is 2^n.

Let's consider the function ***Minim()*** that computes the minimum between three integers:

```
int Minim(int a, int b, int c)
    {
        int min;
        min=a;
        if(min>b)
            min=b;
        if(min>c)
            min=c;
        return min;
    }
```

The graph associated with the function ***Minim()*** is presented in figure 4 a) and the tree structure is illustrated in figure 4 b).

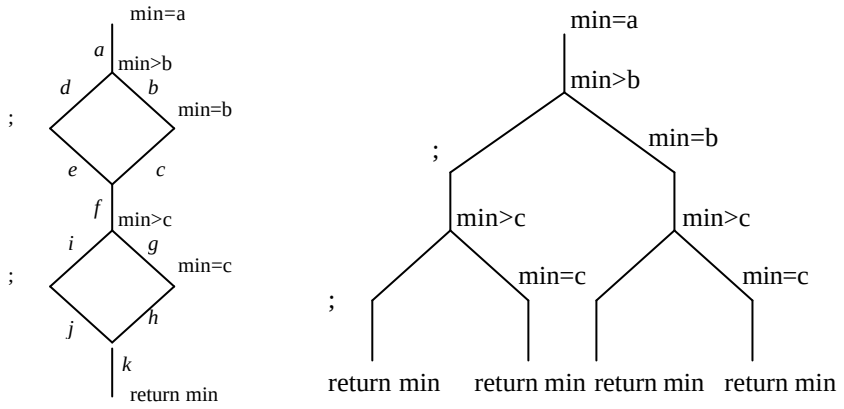


Fig. 4. The graph (a) and the tree structure (b) associated to the function Minim

The graph from figure 4 a) has the cyclomatic number equal to CV(G)=3. Thus, three test examples are chosen to execute all instructions in all possible paths.

Table 5. Test cases to cover all possible instructions and branches.

No.	Input	Path	Output	
1	3,2,1	**abcfghk**	c	1
2	2,3,1	a**de**fghk	c	1
3	1,3,2	ade**fij**k	a	1

The procedure is not very complex. Analyzing the conditions after the BRO [PRES00] strategy leads to the following possibilities.

C1: min>b
min=a

1) a=b F
2) a<b F
3) a>b T

C2: min>c
min=b

4) b=c F
5) b<c F
6) b>c T

min=a

7) a=c F
8) a<c F
9) a>c T

Table 6 presents the test cases chosen based on these results.

Table 6. Test data for the function ***Minim()***

No.	Covered situations	Inputs		Outputs	
1	1), 7)	a=b=c	1,1,1	a	1
2	1), 8)	a=b>c	2,2,1	c	1
3	1), 9)	a=b<c	1,1,2	a	1
4	2), 7)	a<b, a=c	1,2,1	a	1
5	2), 8)	a<b, a<c	1,2,3	a	1
6	2), 9)	a<b, a>c	2,3,1	a	1
7	3), 4)	a>b=c	2,1,1	b	1
8	3), 5)	a>b<c	2,1,3	b	1
9	3), 6)	a>b>c	3,2,1	c	1

If the alternative procedures lead to a very large number of possible paths, the paths are divided into a hierarchy of binary subtrees and the tests focus on these subtrees.

3. Repetitive procedures

The procedures that include a repetitive structure such as ***for***, ***while*** or ***do…while*** are called ***repetitive procedures*** and look like this:

```
type     procedure_name(type1
p1, type2 p2,…, typek pk)
{
      for(expr)
      {
            I1
            I2
            I3
            …
            In
      }
      return (return_value);
}
```

or

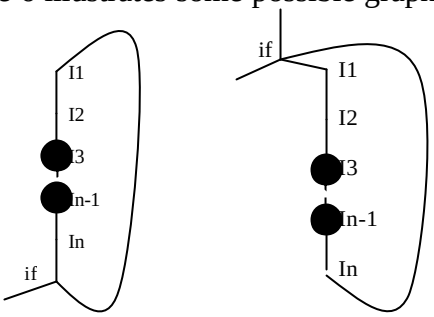
```
type     procedure_name(type1
p1, type2 p2,…, typek pk)
{
      while(expr)
      {
            I1
            I2
            I3
            …
            In
      }
      return (return_value);
}
```

or

```
type     procedure_name(type1
p1, type2 p2,…, typek pk)
{
      do
      {
            I1
            I2
            I3
            …
            In
      }
      while(expr);
      return (return_value);
}
```

The I1, I2, …, instructions are organized in a linear structure Si that can be repeatedly executed for a number of times. The number of repetitions depends on the control variable i that is evaluated in an relational expression. The reputed execution of the sequence generates a sequence of sequences Si1, Si2, …, Sim.

Depending on the position of the relational expression, the repetitive procedure is associated with one graph or another one. Figure 6 illustrates some possible graphs.



a) ***for***, ***while*** b) ***do…while***

Fig. 6. The graph associated to a repetitive procedure

The repetitive structure belonging to a procedure can be:

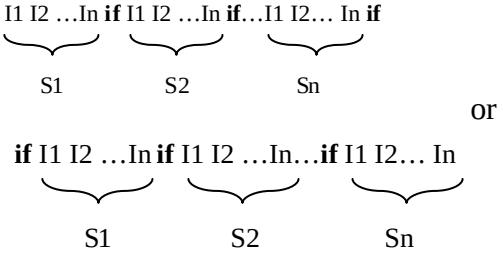
- simple, when there is only one repetitive sequence
- concatenated, when there are several repetitive sequences, one after another one
- embedded, when there are several repetitive structures, one inside another one.

Testing a repetitive procedure reduces to testing a sequence Si and the sequences Si1, Si2, …, Sim. The intermediary results obtained when testing each sequence Sij, j=1, 2, …m are as important as the results obtained from the final testing of the sequence of sequences. The testing results divide the sequences into two classes:

- the class of sequences executed according to the S' specifications;
- the class of sequences with errors S''.

In the case of the class S'', one needs to identify a cause. If there are several causes, one needs to analyze the matrix of precedence defined for the linear structure included in the repetitive block.

The graph associated to the sequence of sequences is:



The score of testing the sequence is computed using the formula Sij,.

$$a(Si_j) = \begin{cases} 1, \text{ correct execution} \\ 0, \text{ eroneous execution} \end{cases}$$

If the correctness degree,

$$C = \frac{\sum_{j=1}^{m} a(Si_j)}{m} \in [A, B],\ A,\ B \in [0,1],\ A < B$$

where A and B are given in the specifications, then the repetitive procedure is accepted in the finite product. The indicator C is used to compute the experimental reliability of the final software product.

The test data for the simple repetitive sequences and the independent concatenated sequences are chosen as follows:

- value 0, no iteration
- value n, the maximum number of iterations
- value k, where 0<k<n
- value n-1
- value n+1.

For the embedded repetitive sequences, the test data is chosen by first starting with the interior repetitive structure that will be tested with the values 0, n, k, n-1 and n+1 while the other repetitive structures are kept to the minimum value of iterations. Then, the process is continued towards the exterior for all other repetitive structures until the entire sequence of instructions associated to the embedded repetitive structure is tested.

Let's consider the procedure that computes the sum of the elements on a row in a matrix $A_{MN,}$ when all elements are integers. The result will be a vector with the number of elements equal to M, the number of rows in the matrix A.

The source code for the procedure is:

```
#define N 10
#define M 10

void   SumRows(int   a[M][N],
int m, int n, int *sum)
{
      for(int i=0;i<m;i++)
      {
            sum[i]=0;
            for(j=0;j<n;j++)

sum[i]+=a[i][j];

      }
}
```

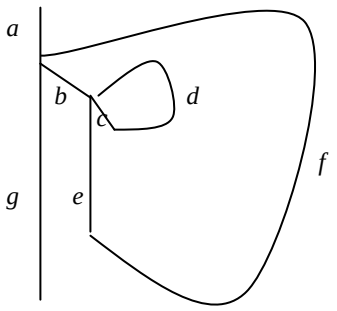


Fig. 7. The graph associated to the procedure ***SumRows()***

The cyclomatic complexity of the function ***SumRows()*** is CV(G)=3. Thus, three test cases should be enough to traverse all the instructions and independent branches. The test cases are presented in table 7.

Table 7. The test cases to cover all possible paths

No.	Inputs	Path	Outputs
1	a={{1,1,1},{10,10,10},{100,100,100}} m= 3 n= 3 sum={0,0,0}	abcdefg	sum={3,30,300}
2	a={{1,1,1},{10,10,10},{100,100,100}} m= 0 n= 3 sum={0,0,0}	ag	sum={0,0,0}
3	a={{1,1,1},{10,10,10},{100,100,100}} m= 3 n= 0 sum={0,0,0}	abefg	sum={0,0,0}

The test data for the function ***SumRows(),*** is chosen considering that the maximum number is 10 and it is equal to the size of the matrix.

m=1; n={0,1,3,9,10,11}

m={0,1,3,9,10,11}; n=1

The obtained test cases are presented in Table 8.

Table 8. The test cases for testing the repetitive structures

No.	Inputs	Outputs
1	a={{1}} **m= 1** **n= 1** sum={0}	sum={1}
2	a={{1,1,1}} m= 1 **n= 3** sum={0}	sum={3}
3	a={{1,…,1,1}} m= 1 **n= 9** sum={0}	sum={9}
4	a={{1,…,1,1}} m= 1 **n= 10** sum={0}	sum={10}
5	a={{1,…,1,1}} m= 1 **n= 11** sum={0}	sum={10} Error related to a[0][**10**] An error message is expected
6	a={{1},{10},{100}} **m= 3** n= 1 sum={0,0,0}	sum={1,10,100}
7	a={{1}, {1},…, {1},{1}} **m= 9** n= 1 sum={0,…,0,0}	sum={1,1,…, 1,1}
8	a={{1}, {1},…, {1},{1}} **m= 10** n= 1 sum={0,…,0,0}	sum={1,1,…, 1,1}
9	a={{1},{1},…,{1},{1}} **m= 11** n= 1 sum={0,…,0,0}	sum={1,1,…, 1} Error related to [**10**][0] and sum[10] An error message is expected

4. Procedure with calls

The object oriented programming technique leads to building procedures that contain calls to other procedures and functions.

The construction looks like:

```
type     procedure_name(type1
p1, type2 p2,…, typek pk)
{
      for(expr)
      {
            name1(param_list
_1);
            name2(param_list
_2);
            name3(param_list
_3);
            …
            na-
men(param_list_n);
      }
      return (return_value);
}
```

and is often seen in programs written by programmers experienced in using function libraries.

Testing procedures where the semantics of other procedure calls is not know requires a new approach. The parameters lists of the n procedure calls are used to generate the dependences where one aij element can take one of the values: I – if pj is an input parameter for the procedure Pi, E – if pj is an output parameter for the procedure Pi, S – if pj is a state parameter for the procedure Pi.

The element bij from the dependences between procedure calls is 1 if the output of the procedure Pj is directly used by the procedure Pi, and 0 otherwise. A procedure Pi is strongly dependent on the procedures that precede its call, if x=[0.72,1] of these procedures have a number of output parameters are used as inputs in the procedure Pi. A procedure Pi is strongly dependent of the procedures that come after it if x=[0.72,1] of these procedures use the output of the procedure Pi as input parameters. A procedure Pi is weakly dependent if 1-x of the procedures crate outputs for the procedure Pi. Testing requires building the

dependence graph and traversing this graph by covering all possible paths. Let's consider the procedure

```
int P0(int a, int b, int c)
{
      int x,y,z,w;
      P1(a,b,x);
      P2(a,c,y);
      P3(b,c,z);
      P4(x,y,z,w);
      return w;
}
```

The dependences matrix for the procedure ***P0()*** is presented in Table 9, the dependences graph is illustrated in Figure 8, and Figure 9 shows the graph associated to the procedure ***P0()***.

Table 9. The dependences matrix for procedure ***P0()***

Procedure / Procedure	P0()	P1()	P2()	P3()	P4()
P0()		1	1	1	
P1()					1
P2()					1
P3()					1
P4()	1				

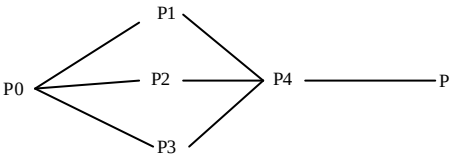


Fig. 8. The dependences graph

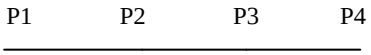


Fig. 9. The graph associated to the procedure P0()

The test cases for the procedure ***P0()*** are built depending on the semantics of each procedure call. Usually, one selects special values that belong to the parameter domain (0 or 1 or –1 or values closely located to the ends of the intervals). These values are then combined and used in testing by following rules specific to experience planning techniques from applied statistics area.

5. Conclusions

Testing different types of procedures requires building definitions for software constructs enabled for maximum cohesion. Each procedure is orthogonal to the other procedures from the software product. This means that each procedure achieves a well-defined processing type that cannot be found in the other procedures.

This type of homogeneous structures of procedures, specific to the sequences included in the procedure, creates a new approach for programming, characterized by redundancy control and orientation towards lower levels.

The specialization of classes of procedures leads to an increase in productivity of the testing process and to an efficient use of the middleware that assists this process.

Bibliography

[ARHI00] Arhire, Romulus – *Evaluating complex programming systems – PhD Thesis*, The Academy of Economic Studies, Bucharest, 2000

[BEIZ90] Beizer, Boris, *Software Testing Techniques – Second Edition*, Van Nostrad Reinhold, New York, 1990

[IVAN99] Ivan, Ion, Pocatilu, Paul – *Testing Object Oriented Software*, Inforec Publishing, Bucharest, 1999

[POCA01b] Pocatilu, Paul – *The Role of Software Testing in Increasing the Software Reliability*, Proceedings of "Software Reliability Workshop", Bucharest, 2001

[POCA02] Pocatilu, Paul – *The Costs of Software Testing*, Informatics Economics vol. VI, nr. 1, 2002, pp. 90-93

[PRES00] Pressman, Roger S. – *Software Engineering – A Practitioner's Approach, European Adaptation, Fifth Edition*, McGraw-Hill, 2000