AUTOMATIC TEST DATA GENERATION USING DATA FLOW INFORMATION

Abdelaziz Khamis
Dept of Comp. & Inf. Sciences, Cairo University

Reem Bahgat
Dept of Computer Science, Cairo University

Rana Abdelaziz
Dept of Comp. & Inf. Sciences, Cairo University

Abstract: This paper presents a tool for automatically generating test data for Pascal programs that satisfy the data flow criteria. Unlike existing tools, our tool is not limited to Pascal programs whose program flow graph contains read statements in only one node but rather deals with read statements appearing in any node in the program flow graph. Moreover, our tool handles loops and arrays, these two features are traditionally difficult to handle in test data generation systems. This allows us to generate tests for larger programs than those previously reported in the literature.

Keywords: *Software testing, Automated test generation, Test adequacy criteria, Dataflow criteria*

Özet: Bu makalede veri akış kriterini sağlayan Pascal programları için otomatik test verisi üreten bir yazılım programı sunulmuştur. Mevcut programların aksine, bizim programımız Pascal programında tek bir düğümdeki okuma komutuna bağlı kalmamakta, bunun yerine herhangi bir düğümde bulunan okuma komutuyla ilgilenmektedir. Ayrıca test veri üretim sistemlerinde incelenmesi zor olan çevrim ve dizileri de ele almaktadır. Bu metod, literatürde mevcut programlardan daha kapsamlı programlar için test üretimini mümkün kılmaktadır.

Anahtar Kelimeler: *Yazılım testi, Otomatik test üretimi, Test uygunluk kriteri, Veri akış kriteri*

1. INTRODUCTION

Program testing is the most commonly used method for demonstrating that a program accomplishes its intended purpose. It involves selecting elements from the program's input domain, executing the program on these test cases, and comparing the actual output with the expected output. Since the input domain is usually too large for exhaustive testing to be practical, the usual procedure is to select a relatively small subset, which is in some sense representative of the entire input domain.

An important problem in software testing is deciding whether or not a program has been tested enough. Test adequacy criteria were developed to address this problem. A test data adequacy criterion is a set of rules that is used to determine whether or not sufficient testing has been performed. In section 2, we illustrate the basic notions underlying adequacy criteria, overview the major categories of these criteria, and conclude section 2, with selecting a test adequacy criterion to be used in our approach.

Once a test adequacy criterion has been selected, the question that arises next is how to go about creating a test data that is good with respect to that criterion. Since this can be difficult to do by hand, there is a need for automatic test data generation. In section 3, we overview the most recent procedure for structural-oriented test data generators namely, dynamic domain reduction procedure (DDR). This paper presents a new procedure for use in structural-oriented generators. In section 4, we present our new procedure which addresses the shortcomings of the DDR procedure.

2. Test Data Adequacy Criteria

The software testing literature contains two different, but closely related, notions associated with the term test data adequacy criteria [1]. First, an adequacy criterion is considered to be a stopping rule that determines whether sufficient testing has been done so that it can be stopped. As a stopping rule, an adequacy criterion can be formalized as a function C that takes a program p, a specification s, and a test data t and gives a truth value *true* or *false*. *C(p,s,t) = true means* that t is adequate for testing p against specification s according to the criterion C, otherwise t is inadequate.

Second, test data adequacy criteria provide a measure of test quality. As measurement, an adequacy criterion can be formalized as a function C that takes a program p, a specification s, and a test data t and gives a degree of adequacy which is a real number in the interval [0,1]. $C(p,s,t) = r$ means that the adequacy of testing the program p by the test t with respect to the specification s is of degree r according the criterion C. The greater the real number r, the more adequate the testing.

There are various ways to classify adequacy criteria. One of the most common is by the *source of information* used in the adequacy criteria. Hence, an adequacy criterion can be either *specification-based or program-based* adequacy criteria. A specification-based adequacy criterion specifies the required testing in terms of identified features of the specification of the program under test, so that a test set is adequate if all the identified features have been fully exercised.

A program-based adequacy criterion specifies the required testing in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised. An example of a program-based adequacy criterion is path adequacy [2]: If a program ***P*** is represented by a flowchart, then a *path* in ***P*** is a finite sequence of nodes $<n_1, \ldots, n_k>$ $k \geq 2$ such that there is an edge from ni to n_{i+1} for i = 1,2,...,k-1 in the flowchart representing ***P***. Test set ***T*** is path adequate for P, if for every path ***p*** in **P**, there is some ***t*** in ***T*** which causes p to be traversed.

Another classification is by the *underlying testing approach.* There are two basic approaches to program testing namely, *structural testing and fault-based testing.* Structural testing focuses on the coverage of a particular set of elements in the structure of the program or the specification.

Fault-based testing focuses on detecting faults in the program. An adequacy criterion of this approach is some measurement of the fault detecting ability of test sets. Mutation is an example of the fault-based testing approach. Mutation testing is based on the assumption that a program will be tested if all simple faults are detected and removed. Simple faults are introduced into the program by mutation operators [3].

The source of information used in the adequacy measurements and the underlying approach to testing can be considered as two dimensions of the space of test adequacy criteria. A test adequacy criterion can be classified by these two aspects. In this paper we consider only one group of the program-based structural adequacy criteria namely, data-flow criteria. Our selection is based on experimental comparisons of the fault-detecting ability of several test data adequacy criteria [4], [5], [6].

2.1 Data-Flow Criteria

Before we define data flow criteria we give a brief introduction to the *flow-graph model* of a program. A flow graph is a directed graph that consists of a set N of nodes and a set $E \subseteq N \times N$ of directed edges between nodes. Each node is either a statement node, representing a linear sequence of computations, or a condition node, representing the predicate that controls the conditional or the repetitive statements. Each edge is represented by an ordered pair $<n_1,n_2>$ of nodes and represent flow of control from node n_1 to node n_2. If n_1 is a condition node, and n_2 is a statement node then the corresponding edge is labelled by either 't' or 'f', denoting the values true and false respectively. In a flow graph there is a begin node and an end node where the computation starts and finishes, respectively. Every node in a flow graph must be on a path from the begin node to the end node.

Example 1. The following Pascal program computes the number of odd and even numbers in a list of input numbers ending with the value zero.

```
program  test;
var x , e , o : integer ;
begin
   e := 0 ;
   o := 0 ;
   read( x ) ;
   while x <> 0 do
           begin
              if ( x mod 2 ) = 0
                 then e := e + 1
                 else  o := o +1 ;
              read( x )
           end ;
    write( e , o )
end .
```

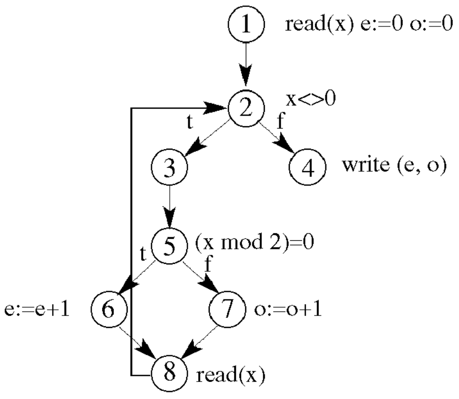


Figure 1. Flow graph for program in Example 1.

Figure 1 is an example of flow graph. It should be noted that in the literature there are a number of slight differences in presenting such graphs. However, adequacy criteria can be defined independently of such differences.

Now, we introduce the way that data flow information is added into the flow graph model of the program under test. Data flow test adequacy analysis focuses on the occurrences of variables within the program. Each variable is classified as either a definition occurrence or a use occurrence. A definition occurrence of a variable is where a value is associated with the variable. A use occurrence of a variable is where the value of the variable is referred. Each use occurrence is further classified as a computational use (*c-use*) or a predicate use *(p-use)*. If the value of the variable is used to decide whether a predicate is true for selecting execution paths, the occurrence is called a predicate use. Otherwise, the occurrence is called a computational use.

Rapps and Weyuker have proposed a family of testing adequacy criteria based on data-flow information [7]. Frankl and Weyuker later extended the definitions of these criteria [8]. Their criteria require that test data be included which cause the traversal of sub-paths from a variable definition to either some or all of the *p-uses*, *c-uses*, or their combination. However, empirical evidences show that the *all-uses* criterion is the most effective criterion compared to the other data flow criteria [9], [10]. All-uses criterion requires that test data be included which causes the traversal of at least one sub-path from each variable definition to every p-use and every c-use of that definition.

A definition-use association is a triple <d, u, v>, where d is a node in the program's flow graph in which a variable v is defined, and u is an edge or a node in which v is used, and there is a path from d to u on which v is not redefined. If we apply the all-uses criterion on example 1, the definition-use associations will be:

<1, (2,3), x>
<1, (2,4), x>
<1, (5,6), x>
<1, (5,7), x>
<8, (2,3), x>
<8, (2,4), x>
<8, (5,6), x>
<8, (5,7), x>

3. Automatic Test Data Generation

Test data generation is the process of creating program inputs that satisfy some t esting criterion. The problem of automatic test data generation has been examined by a number of researchers [11], [12], [13], [14]. As pointed out in [11], test data generators can be categorised into three *groups: structural-oriented test data generators attempt* to cover certain structural elements in the program, *specification-oriented test data generators* generate test data from a formal description of the input domain, and *random* test data *generators* create test data according to some distribution of the input domain without satisfying any test criterion.

In this paper we focus on structural-oriented test data generators. For this category of generators, the test data generation problem is defined as follows. Let n_g be a node in the flow graph of a program P with input domain D, n_g is called the goal node. The test data generation problem is: *find a program input t $\in$ D such that when P is executed on t, n_g will be reached.* To extend this problem to include the all-uses data flow criterion, the goal n_g becomes the node that contains a definition of a variable x, and the requirement is added that after n_g is reached, the node containing the use of x (n_u) must be reached, with the further restriction that the sub-path from n_g to n_u must not contain another definition of x.

As far as we know, Offutt, Jin, and Pan in [14], present the most recent procedure, called *dynamic domain reduction procedure (DDR),* which addresses most of the problems in the existing structural-oriented test data generators. In this section we give an overview the DDR procedure and define some of its weaknesses.

3.1 Overview the dynamic domain reduction procedur e

The DDR procedure starts with several pieces of information: a flow graph, the initial domains for all input variables, and two nodes representing the initial and goal nodes. The first step is that a finite set of paths from the initial to the goal node is determined. Then each path is analyzed in turn. The path is traversed, and symbolic execution is used to progressively reduce the domains of values for the input variables. When choices for how to reduce the domain must be made, a search process is used to split the domain of some variable in an attempt to find a set of values that allow the constraints to be satisfied. Finally, a test case is chosen arbitrarily from within the reduced input domains.

The DDR procedure suffers from several shortcomings that prevent it from working in some situations and hamper its applicability in practical situations. These include three main problems. First, The DDR procedure deals with programs whose flow graphs have only one node containing read statements. Second, it assumes that the domains of input variables have only discrete values. A third problem is that the DDR procedure has partially solved the problems of arrays and loops.

4. OurApproach for Test Data Generation

This paper presents a novel procedure for automated test data generation, which overcomes the weaknesses of the DDR procedure that are stated above. Our procedure is based on the idea of dividing the input domain of the program under test into subsets, called sub-domains, then requiring the test cases to include elements from each sub-domain. The following concepts and assumptions are used in our procedure.

The *domain of a variable* is the set of all possible values the variable can have. The domain is either continuous or discrete, depending on the variable's type declaration. Our procedure assumes that the intersection between the variables' domains is not empty. The domain of the program consists of all the domains of its variables.

The *Sub-domains of a program* are subsets produced by dividing the domain of the program. When the all-uses data flow criterion is used to divide the domain of the program, each sub-domain would consist of all the inputs that execute any path from a particular definition of a variable v to a particular use of v without any intervening redefinition of v.

The *sub-domain of a path* is a predicate that restricts the space of the program variables to certain domains. If input data that satisfy the sub-domain of a path exist, the path is said to be an execution path, and that data can be used to test the program. If the sub-domain of the path cannot be satisfied, the path is said to be infeasible.

The *start nodes* in the program flow graph are the begin node and any other node that contains a read statement. For the program flow graph in figure 1, the start nodes are node-1 and node-8.

An *independent variable* in a program is an input variable that doesn't depend on the value of another input variable, while a *dependent variable* is an input variable that depends on the value of another input variable. A *dependent path* is a path from a node containing a read statement for an independent variable to a node containing read statements for a dependent variable.

4.1 Overview of our new procedur e

Our procedure is quite difficult to clearly describe. We proceed by first giving a high level overview of the procedure, followed by an illustrative example, then a detailed description of the procedure.

The main steps of our procedure are:

1- Produce the table of definition-use associations for the input variables.
2- Identify two types of paths, namely: paths from the start nodes to the definition nodes and paths from the definition nodes to the use nodes.
3- For each path identified in step 2, determine its sub-domain, and randomly select an element from each input variable's domain such that they satisfy the sub-domain of the path.

In example 1, assume the domain of x is [-50··50]. Using the criterion all-uses, our procedure would produce the following:

Definition-use association	**Sub-domains**	**Test cases**
(1,(2,3),x)	x<>0	x = 40
(1,(2,4),x)	x=0	x = 0
(1,(5,6),x)	(x<>0)and((x mod 2)=0)	x = -44
(1,(5,7),x)	(x<>0)and((x mod 2)<>0)	x = 19
(8,(2,3),x)	(x<>0)and((x mod 2)=0)	x = -28
	(x<>0)and((x mod 2)<>0)	x = 21
	(x<>0) and((x mod 2)=0)	x = 14
	(x<>0)and((x mod 2)<>0)	x = 33
	(x<>0)	x = -27
(8,(2,4),x)	(x<>0)and((x mod 2)<>0)	x = 29
	(x<>0)and((x mod 2)=0)	x = 38
	(x<>0)and((x mod 2)<>0)	x = 31

	(x<>0)and((x mod 2)=0)	x = 32
	(x=0)	x = 0
(8,(5,6),x)	(x<>0)and((x mod 2)=0)	x = 26
	(x<>0)and((x mod 2)<>0)	x = -17
	(x<>0)and((x mod 2)=0)	x = -18
	(x<>0)and((x mod 2)<>0)	x = 39
	(x<>0)and((x mod 2)=0)	x = -40
(8,(5,7),x)	(x<>0)and((x mod 2)<>0)	x = 41
	(x<>0)and((x mod 2)=0)	x = -48
	(x<>0)and((x mod 2)<>0)	x = 9
	(x<>0)and((x mod 2)=0)	x = 44
	(x<>0)and((x mod 2)<>0)	x = 23

This example has one input variable and there are no dependent variables. At triple (1, (2,4), x), node-1 (def-node) contains readln(x) and it is the start node, the path from node-1 (def-node) to edge(2,4) (use edge) is 1-2-4 has the sub-domain (x=0), i.e. this triple is associated with the sub-domain (x=0). 0 is an element from [-50,50] which satisfies the sub-domain (x=0).

At triple (8, (2,3), x), node-8 (def node) contains readln(x), and the start nodes are node-1 and node-8. Paths from the start nodes to the def-node, node-8, are: (**1**) The 1-2-3-5-6-8, with sub-domain ((x<>0) and ((x mod 2)=0)) and -28 is an element selected from [-50,50] that satisfies the sub-domain, (**2**) The path 1-2-3-5-7-8 with sub-domain ((x<>0) and ((x mod 2)<>0)), and 21 is an element selected from [-50,50] which satisfies the sub-domain, (**3**) The path 8-2-3-5-6-8 with sub-domain ((x<>0) and ((x mod 2)=0)) and 14 is an element selected to satisfy the sub-domain, and (**4**) The path 8-2-3-5-7-8 with sub-domain ((x<>0)and((x mod 2)<>0)) and 33 is an element satisfying the sub-domain. Finally, the path from the def-node, node-8, to the use-edge (2,3) is 8-2-3, with sub-domain (x<>0), and -27 satisfies the sub-domain. Hence, the triple (8, (2,3), x) is associated with the test case -28, 21,14,33, -27. Note that this test case may not lead to loop termination, we will continue this example in section 4.3.

4.2 Detailed description of the procedur e

Figure 2 shows an abstract algorithm of our procedure. The algorithm uses modules for producing the definition-use association table, finding all the dependent paths in the control flow graph, generating the sub-domain of a given path, and randomly selecting elements from the program domain to satisfy a given sub-domain.

In figure 3, we overview the algorithm for handling loops that contain read statements. The algorithm is based on the idea of identifying the progress variables in the considered loop, finding their initial values, and finally updating their values using the progress statements. Figure 3 can be simplified to Figure 4, when the progress variables are variables in the read statements.

```
Find the table of definition-use association for the input variables.
dep-paths := all the dependent paths in the control flow graph;
dep-sd_i := sub-domain-of (dep-paths_i);
T := empty set of test cases;

For each triple (d, u, v) in the table Do
  Begin
   If the node d contains read (v) Then
     Begin
          t := empty test case;
          For each start node i Do
               For each path from the start node i to the node d, namely p_id Do
                    Begin
                         sd_id := sub-domain-of (p_id);
                         t_id := rand-select( sd_id);
                         Add t_id to t;
                    End
          P_du := paths from the node d to the node u;
          sd_du := sub-domains-of (P_du);
          t_du := rand-select(sd_du);
          Add t_du to t;
     End
   Else If the node d contains v : = A Then
          Begin
               P_id := paths from every start node i to node d;
               P_du := paths from node d to node u;
               sd_d := sub-domains-of (P_id);
               sd_u := sub-domains-of (P_du) after replacing v by A;
               sd := sd_d and sd_u;
               t := rand-select(sd)
          End;
   Add t to T
End
Function rand-select (sd)
Begin
  el := randomly selected element from each input variable's domain to satisfy sd;
  IF sd is a sub-domain for a path of the second type Then
     For each dependent sub-domain dep-sd_i Do
        IF el satisfies dep-sd_i Then
           Add to el randomly selected elements from dependent variables'domains;
  rand-select := el;
End
```

Figure 2. The Test Data Generation Procedure

Figure 5, shows our strategy for handling sub-domains that contain non-input variables. The strategy is based on the idea of applying the all-uses data flow criterion on these non-input variables, to identify the nodes that contain their definitions.

Our strategy for handling arrays is shown in Figure 6. The strategy is based on the idea of checking the index of the array to see whether it is an input variable or not.

4.3 Handling Loops Containing Read Statements

The problem with analyzing loops is that some test cases may not lead to loop termination. In example 1, the first test case , x = 44, does not terminate the loop. This problem is not faced if the loop body does not contain read statements. Our strategy for handling the loops which contain read statements is shown in Figure 3.

```
 Identify the progress variables in the considered loop;
 For every test case, t Do
  Begin
      T := t;
      Find values of the progress variables  for the considered test case, t;
      While the loop condition Do
        Begin
           p:= path from the first node of the loop to
                 node containing the progress statement;
           sd := sub-domain-of (p);
           t := rand-select(sd);
           Add t to T;
           Update the values of the progress variables using progress statements;
        End
  End
```

Figure 3. The loop handling procedure

The procedure in Figure 3 can be simplified to be as in Figure 4, when the progress variables are variables in the read statement. Applying the simplified procedure to example 1, where the condition of the loop is (x<>0), the procedure adds x=0 to all the test cases that do not have x=0.

```
For every test case, t Do
    Begin
        T := t;
        If the loop condition Then  t := rand-select(not condition of loop);
        Add t to T;
    End
```

Fig 4. Simplification of the loop handling procedure

4.4 Handling sub-domains depending on non-input variables

Instead of walking through the program flow graph to determine the value of a non-input variable in a sub-domain, our procedure applies all-uses data flow criterion on that variable to identify the nodes which contain its definitions. Then, depending on whether the statement definition is a progress statement or not, our procedure finds a new sub-domain by replacing the non-input variable by its assigned expression. The strategy to find the new sub-domains is shown in Figure 5. If one of the new sub-domains still depends on another non-input variable then the above process is repeated until the new sub-domain is free from any non-input variables.

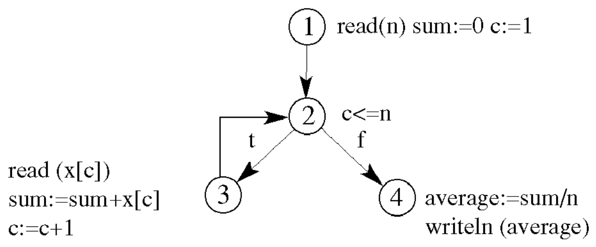
```
sd := sub-domain with non-input variables;
Find the table of definition-use association for non-input variables in sd;
NEW-SD := empty new-sub-domains;
For each triple (d, u, v) in the table Do
    Begin
        If the assignment statement of v at node d is not a progress statement Then
            Begin
                exp := right-hand side of the assignment statement;
                new-sd := sd after replacing v by exp;
            End
        Else If the assignment statement of v depends on input variables Then
            Begin
                p1:= path from the begin node to the node d;
                p2:= path from node d to the node u;
                n-sd := sub-domain-of (p1) and sub-domain-of (p2);
                new-sd := n-sd after replacing v by exp;
            End;
        Add new-sd to NEW-SD;
    End
```

Figure 5. Handling sub-domains with non-input variables

Example 2. The following program calculates the average of an array of numbers.

```
Program simple;
Var n , c :integer;
    sum , average :real;
    x :array[1..100] of real;
Begin
    read(n);
    sum := 0;
    for c :=1 to n do
    begin
        read( x[c]);
        sum := sum + x[c]
    end;
    average := sum/n;
    write(average)
End.
```

The program flow graph is:



Definition-use association	*Subdomain*
(1, (2,3), n)	c<=n
(1,(2,4),n)	c>n
(1,4,n)	c>n

The sub-domain (c<=n) depends on c (a non-input variable), hence requires applying all-uses criterion on c itself.
Definition-use association of c
(1,(2,3),c)
(1,(2,4),c)

At node 1, the assignment c:=1 is not progress statement of the loop. Hence, the new sub-domain is (1<=n), also the sub-domain (c>n) becomes (1>n). The next subsection will complete this example.

4.5 Handling Arrays

The existing data flow analysis treats arrays as scalar variables, that is, a reference to any element is treated as a reference to all elements. In order to overcome this problem, our new procedure analyses the index of the array to check whether it is an input variable or a non-input variable. Then, for each sub-domain, our procedure determines the initial value of the index and how the index changes its value in the program. The strategy for handling arrays is shown in Figure 6.

```
For each test case Do
   Begin
        Let X be a variable of type array;
        If the index of X is an input variable
           Then index := the value of the index variable in the test case
           Else  index := initial value of index;
        Replace X by X[index]
        Step := the value used to modify the index value;
        If Step is defined by an input variable
            Then index := the value of the Step variable in the test case
             Else  index := index + Step;
  End
```

Figure 6. Handling Arrays

In example 2, assume that the domain of n is [-3‥10], the domain of x is [-50‥50], and the criterion is all-uses. Applying our procedure we get:

Definition-use association	*Test cases*
(1,(2,3),n)	n = 5, x[1] = 30.34, x[2] = 4.56 x[3] = -34.78, x[4] = 7.89, x[5]=9
(1,(2,4),n)	n = -1
(1,4,n)	n = -2

In the first case, 30.34 is an element from the domain of x, the index c is a non-input variable whose initial value is 1, then the case x = 30.34 is replaced by x[1] = 30.34. Since Step = 1, then index value becomes 2 using the 'handling loop'procedure, and x = 4.56 is replaced by x[2] = 4.56. Similarly, x[3] = -34.78, x[4] = 7.89 and x[5]=9 replace the corresponding cases. The other test cases don't need to apply the strategy in Figure 6.

5. Conclusions

In this paper, we proposed a new method for automatically generating test data for Pascal programs. Our method addresses the weak points of the most recent test data generator, namely the Dynamic Domain Reduction (DDR) procedure. This allows us to generate tests for larger class of problems than those previously reported in the literature.

REFERENCES

[1] ZHU, H., HALL, P. and MAY J. (1999), "Software Unit Test Coverage and Adequacy", ACM Computing Surveys, Vol. 29, No. 4.

[2] WEYUKER, E. J. (1986), "Axiomatizing Software Test Data Adequacy". IEEE Trans. On Software Eng., Vol. 12, NO. 12, pp. 1128-1138.

[3] DEMILLO, R. and OFFUTT, A. J. (1991), "Constraint-Based Automatic Test Data Generation", IEEE Trans. On Software Eng., Vol. 17, NO. 9, pp. 900-910.

[4] FRANKL, P. G. and WEYUKER, E. J. (1993), "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing ". IEEE Trans. On Software Eng., Vol. 19, NO. 8, pp. 774-787.

[5] FRANKL, P. G. and WEYUKER, E. J. (1993), "Provable Improvements on Branch Testing". IEEE Trans. On Software Eng., Vol. 19, NO. 10, pp. 962-975.

[6] OFFUTT, A. J., PAN, J., TEWARY K. and ZHANG, T. (1996), "An Experimental Evaluation of Data Flow and Mutation Testing". Software-practice and Experience, Vol. 26, No. 2, pp. 165-176.

[7] RAPPS, S. and WEYUKER, E. J. (1985), "Selecting Software Test Data using Data Flow Information". IEEE Trans. On Software Eng., Vol. 14, NO.4, pp. 367-375.

[8] FRANKL, P.G. and WEYUKER, E. J. (1988), "An Applicable Family of Data Flow Testing Criteria". IEEE Trans. On Software Eng., Vol. 14, NO.10, pp. 1483-1498.

[9] WEYUKER, E. J. (1993), "More Experience with Data Flow Testing". IEEE Trans. On Software Eng., Vol. 19, NO. 9, pp. 912-919.

[10] FRANKL, P. G. and WEYUKER, E. J. (1993), "AFormal Analysis of the Fault-Detecting Ability of Testing Methods". IEEE Trans. On Software Eng., Vol. 19, NO. 3, pp. 202-213.

[11] INCE, D. C. (1987), "The Automatic Generation of Test Data". The Computer Journal, Vol. 30, NO. 1, pp 63-69.

[12] KOREL, B. (1990), "Automated Software Test Data Generation". IEEE Trans. Software Eng. Vol. 16, NO. 8, pp. 870-879.

[13] FERGUSON, R. and KOREL, B. (1996), "The Chaining Approach for Software Test Data Generation", ACM Trans. On Software Eng. And Methodology, Vol. 5, NO. 1, pp. 63-86.

[14] OFFUTT, A. J. and PAN, J. "The Dynamic Domain Reduction Procedure for Test Data Generation", http://www.isse.gmu.edu/faculty/ofut/index.html.