Infeasible Paths in the Context of Data Flow Based Testing Criteria: Identification, Classification and Prediction

Silvia Regina Vergilio[1], **José Carlos Maldonado**[2], **Mario Jino**[3]

[1]Federal University of Paraná
DInf-UFPR - Curitiba, PR
CP: 19081, CEP: 81531-970, Brazil
+55 41 33613411 fax: +55 41 33613502
silvia@inf.ufpr.br

[2]University of São Paulo - USP
ICMC-USP - São Carlos, SP
CP:668, CEP: 13560-970, Brazil
jcmaldon@icmc.sc.usp.br

[3]State University of Campinas
DCA-FEEC-UNICAMP - Campinas, SP
CP: 6101, CEP: 13083-970, Brazil
jino@dca.fee.unicamp.br

Abstract

Infeasible paths constitute a bottleneck for the complete automation of software testing, one of the most expensive activities of software quality assurance. Research efforts have been spent on infeasible paths, basically on three main approaches: prediction, classification and identification of infeasibility. This work reports the results of experiments on data flow based criteria and of studies aimed at the three approaches above. Identification, classification, and prediction of infeasible paths are revisited in the context of data flow based criteria (Potential Uses Criteria-PU). Additionally, these aspects are also addressed in the scope of integration and object-oriented testing. Implementation aspects of mechanisms and facilities to deal with infeasibility are presented taking into consideration Poketool - a tool that supports the application of the Potential Uses Criteria Family. The results and ideas presented contribute to reduce the efforts spent during the testing activity concerning infeasible paths.

Keywords: software testing, data flow based criteria, infeasible paths.

1. Introduction

Testing is one of the most traditional activity for software quality assurance. Considering that software testing aims at revealing faults, a good test case is one that has a high probability of finding a not-yet revealed fault. In other words, a good test case is one that reveals a fault if it is present.

To test a program with all possible input values is, in general, impossible. To select a finite set containing the best inputs, different testing criteria were proposed. Testing criteria have the goal of revealing as many faults as possible with minimal effort and cost. A testing criterion establishes a predicate that has to be satisfied by a given test case set T. It requires that elements of a program, such as statements or paths are exercised by the test cases. It constitutes a means to assess the effectiveness of the testing activity and to provide reliability measures. Usually, they are classified as functional, structural or fault-based testing techniques. The structural criteria consider a particular implementation to derive the testing requirements and are classified as: Control Flow [28], Data Flow [12, 19, 23, 27, 28, 30] and Complexity Based Criteria [26].

Data flow based criteria are among the most investigated criteria for unit testing in the past few years [12, 19, 23, 27, 28, 30]. It would not be unrealistic to say that these criteria have reached their maturity, in the sense that they have moved from the state of the art to the state of the practice. Prototype tools have been developed [4, 6, 14] and a commercial tool is available to support the application of some of these criteria [1]. Some authors have been investigated strategies to allow the application of structural criteria in real and large systems [18].

In spite of this popularization, a common problem associated with the use and evaluation of structural criteria is that amongst the required elements may occur infeasible ones (paths and/or associations). A path through a program is infeasible if there is no set of values for the input and global variables and parameters of the program that cause the path to be executed. In general, there is no algorithm to determine the feasibility of a path; it is an undecidable question [6].

Infeasible elements required by structural testing criteria constitute a bottleneck to the complete automation of software testing - either to generate a test set or to evaluate its adequacy. Data flow based criteria often require exercising infeasible elements since most programs contain infeasible paths, even well-formulated correct programs. Some experiments applying the all-definition-uses(du)-paths and all-potential-du-paths criteria, have reported up to 50% of the total required elements as being infeasible [22, 32, 35, 37].

Infeasibility impacts significantly the effort, cost and time to apply structural testing criteria, in testing, debugging and maintenance activities. Studies on infeasible paths, described in the literature, address three basic approaches: prediction, classification and identification of infeasibility. Malevris et al [24] use the number of predicates in a path to predict infeasibility, considering the all-LCSAJs (Linear Code Sequence and Jumps) criterion. They concluded that the greater the number of predicates in a path, the greater the probability of the path being infeasible. Hedley and Hennel [11] show the main causes of infeasible paths in a program and present a classification for those causes. Frankl [6] introduces a heuristic to determine infeasible associations, based on data flow analysis and symbolic execution techniques.

Among the above mentioned works, only Frankl focuses infeasibility in the context of data-flow testing. Additionally, all of them address only unit testing and nowadays, data flow based testing criteria and tools are being applied within a broader context. For example, data-flow based criteria have been extended to integration testing [2, 8, 10, 20], specification testing [3, 7], object-oriented software [9, 16, 25], concurrent and parallel programs [39, 33] and, most recently, to web applications [21, 29]. In all these new contexts the problem of infeasible paths remains. More recent studies with object-oriented software [25], web applications [29], concurrent programs [5] and test of specifications [13] point out the negative effects of the presence of infeasible paths.

The goal of this paper is to address these basic approaches - classification, prediction and identification - in the context of data flow based testing, more specifically in the context of the Potential Uses Criteria Family [22], by carrying out two experiments [22, 31]. These experiments were conducted using Poketool [4], a tool that supports the application of the Potential Uses Criteria Family and the control flow based criteria: all-nodes and all-edges. During this experiment, we evaluate some existent dependencies between units and the results obtained with both experiments can be used in the broader context of data-flow based testing. We illustrate this fact by addressing the problem of infeasible paths in the integration phase and in the testing of object-oriented software.

Section 2 presents basic terminology and the Potential Uses Criteria. Section 3 describes the experiments. The following sections describe the main results obtained according to the three approaches in the literature. The main causes that generate infeasible paths are described and a classification based on the program contexts is proposed in Section 4. Prediction models validating Malevris' work within the context of data flow based criteria are presented in Section 5. To help in the task of determining infeasibility, some facilities, infeasible patterns and Frankl's heuristic are discussed in Section 6. This section also presents an extension to Poketool that incorporates these facilities. Section 7 addresses the usage of knowledge and information obtained from unit testing to reduce the effort necessary to deal with infeasibility during the integration phase and in the test of object-oriented software. In Section 8 we present the final remarks on our work and future steps on our research.

2. Concepts and Terminology

The basic abstraction used to establish the elements required by structural testing criteria is the control flow graph G, also called program graph. Basically, they require the execution of components of the program under test in terms of corresponding elements of the program graph: nodes, edges and paths. Each block of statements is associated to a node in the graph G. Edges are associated to possible transfers of control between nodes and a path is given by a sequence of nodes, $(n_1, n_2, ..., n_m)$ where $m \geq 2$ and (n_i, n_{i+1}) is an edge in the graph for $i = 1..m-1$. In Figure 1, the program *getlist* and its control flow graph are shown. Examples of control flow based criteria are: all-nodes, all-edges and all-paths.

A common characteristic of data flow based criteria is that they require associations: the interactions between variable definitions and subsequent uses of these definitions [19, 27, 28, 30]. A variable is defined when a value is saved in its corresponding memory position. Typical definition statements includes function side effects (parameter by reference, global variables and member variables) and involve assignments, input commands and initialization of variables in declarations. A use of x occurs when a reference is made to the value associated to x.

The motivation for data flow based adequacy criteria comes from the observation that a test set, which is adequate with respect to control flow based adequacy criteria, such as all-nodes or all-edges, is unable to identify the presence of some simple faults, more specifically, computation faults [38]. Data flow based adequacy criteria aim at reducing these limitations.

To satisfy a structural criterion C it is necessary to exercise all the required elements according to the criterion; in this case, we say that a 100% coverage is obtained and that the corresponding test case set T is C-adequate.

The Potential Uses criteria do not require an explicit use to occur, as all of data flow based criteria do; just a definition clear path from nodes where a definition occurs to nodes in the graph where the definition could be possibly used is enough to characterize an association, in this case, a potential-association. Two of the Potential Uses criteria are defined below.

- All-potential-uses criterion (PU): requires the execution of paths in G that cover all potential-associations in the program. A potential-association (i,j,x) or (i,(j,k),x) is established if a variable x is defined in a node i and there is a path from node i to j (or edge (j,k)) that does not redefine x. The associations <3,9,line1> and <3,(9,11),line1> are examples of required potential-associations for the program *getlist* (Figure 1) because *line1* is defined in node 3 and there is a path in the graph from Node 3 to Node 9 and to arc (9,11) that does not redefine *line1*. However, <1,4,line2> and <1,(3,4),line2> are not potential-associations; all paths from node 1 to node 4 redefine *line2* in node 3.
- All-potential-du-paths criterion (PDU): requires the execution of paths in G that cover all potential-du-paths in the program. A potential-du-path with respect to a variable x is given by a sequence of nodes $(i, n_1, n_2 ..., n_n, j)$ of G such that, a variable x is defined in node i, all the nodes in the sequence $(n_1, n_2 ..., n_n)$ are distinct and contain no redefinition of x. For *getlist* the paths (3, 4, 5, 7, 8, 2, 9, 10) and (3, 4, 5, 7, 8, 2, 3) are potential du-paths with respect to variable *line1*.

To derive the elements required by the Potential Uses criteria we need only to associate variable definitions to each node in the control flow graph. For example the variables *line1, line2* and *nlines* are defined in statements of the program *getlist* associated to Node 3. Determining every potential-association for Node 3 is determining every node to which there is a definition clear path with respect to each of the variables defined in Node 3. Poketool [4] is a testing tool that supports the Potential Uses criteria family [23] and the all-nodes and all-edges criteria for testing of C programs. It provides facilities to instrument the source code, to execute code and to do coverage analysis of a given set of test cases. It also provides other information from static and dynamic analysis of the program.

Given a path, the number of predicates is the number of components of the boolean expressions of all conditions associated to the nodes of that path. For instance, the path (1, 2, 3, 5, 7, 8, 2) has five predicates. The condition in Node 3 has one component, in Node 5 has two components and in Node 2 the condition is considered twice.

As stated before, a program path is infeasible if there is no set of values for the input variables, global variables and parameters of the program that cause the path to be executed. For example, the path (1, 2, 9, 11, 13, 15, 16) in the program *getlist* is infeasible. An association can be covered by a set of distinct paths of the program. If all paths in this set are infeasible the association is consid-

```
stcode getlist(char *lin, int *i, stcode *status)
/*  1 */ {
/*  1 */  int num, done;
/*  1 */  line2 = 0;
/*  1 */  nlines = 0;
/*  1 */  done = (getone(lin,i,&num,status)!=OK);
/*  2 */  while (!done)
/*  3 */  {
/*  3 */    line1 = line2;
/*  3 */    line2 = num;
/*  3 */    nlines++;
/*  3 */    if (lin[*i]==SEMICOL)
/*  4 */      curln = num;
/*  5 */    if ((lin[*i]==COMMA)||(lin[*i]==SEMICOL))
/*  6 */    {
/*  6 */      *i = *i + 1;
/*  6 */      done = (getone(lin,i,&num,status)!=OK);
/*  6 */    }
/*  7 */     else
/*  7 */       done = 1;
/*  8 */  }
/*  9 */  nlines = min(nlines,2);
/*  9 */  if (nlines == 0)
/* 10 */    line2 = curln;
/* 11 */  if (nlines <= 1)
/* 12 */    line1 = line2;
/* 13 */  if (*status != ERR)
/* 14 */    *status = OK;
/* 15 */  return(*status);
/* 16 */ }
```

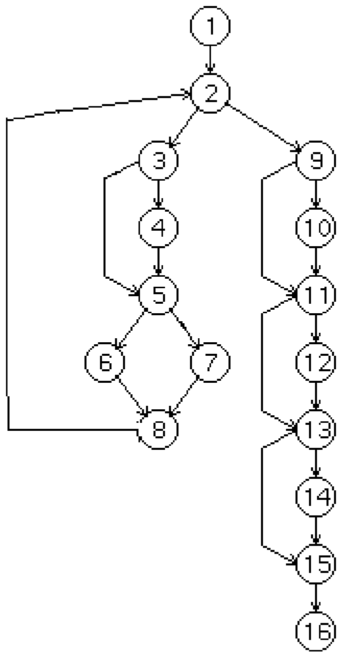

Figure 1. Program Getlist

ered infeasible and it is not possible to obtain a 100% coverage for the criteria. The tester is responsible for dealing with this situation since determining feasibility of a path is, in general, undecidable.

Data-flow based criteria were first used in the context of unit testing, which has the goal of testing a module separately. Extensions to these criteria have been applied at the integration testing level [2, 8, 10, 15, 20, 34]. These criteria aim at revealing interface faults. They are based on call graphs and inter-procedural flows. Inter-procedural associations and paths are the required elements in integration testing.

When we consider object-oriented software, there are three levels of testing [9]: a) Intra-method testing, that tests methods individually; b) Inter-method testing, that tests public methods together, with other methods in its class; and c) Intra-class testing, that tests interactions of public methods, as they are called in various sequences. The first level is equivalent to unit testing and the second and third ones correspond to the integration testing of procedural programs.

3. Description of the Experiments

This section describes two experiments with data flow based criteria that provided data and results pointing out the relevance of infeasible paths for the application of these criteria.

Both experiments were conducted using Poketool, considering the all-potential-uses criterion (PU), and the all-potential-du-paths criterion (PDU). The programs were selected from [17], also used in an experiment by Weyuker [36]. These programs, originally written in Pascal, were translated to C, as Poketool does not support testing of Pascal programs.

Drivers were built to be as general as possible to avoid making feasible paths into infeasible ones, since restrictions in the input provided by the driver to a program may inhibit the execution of feasible paths. The experiment used complete programs instead of stubs, since all the programs were available. Using stubs instead of complete programs may generate a higher degree of infeasibility, due to the usually limited functionality of stubs compared to that of complete programs. The steps carried out in both experiments were basically the same:

1. Initial ”ad-hoc” test set generation considering only functional aspects of the programs;

2. Test set adequacy analysis with respect to the data flow criteria PU and PDU;

3. Manual identification of infeasible elements;

4. Generation of test cases to cover the remaining required elements; and

5. Execution of new test cases until adequacy is achieved.

The main differences between the two experiments are discussed next.

3.1. Experiment I [22, 32]

Experiment I is referred to as "Unit testing". The set of 27 programs listed in Table 1 was used. A driver and an initial test set for each unit have been used. Table 1 presents the final coverage for each criterion, number of executed/required elements. The infeasible elements have been manually determined. For instance, for the program *append*, the PU criterion requires 49 associations, but only 38 were executed because 11 are infeasible; the PDU criterion requires 43 potential-du-paths, but only 23 are feasible. Notice that only three of the 27 programs do not have infeasible elements: *archive*, *dodash* and *getcmd*.

With the goal of replicating Malevris' work for data flow based criteria, the number of predicates of the potential-du-paths required for all programs in Table 1 were determined. Table 2 shows the number of infeasible and feasible potential-du-paths with q predicates, with $0 \leq q \leq 18$. For example, there are 47 potential-du-paths with 13 predicates, 31 of them infeasible and 16 feasible.

3.2. Experiment II [32]

This experiment is referred to as "Cluster Testing". A cluster includes two or more units from Experiment I. Such clusters are listed in Table 3. Notice that some units from Experiment I were not used in Experiment II. An initial "ad-hoc" test case set was generated for a cluster of units.

For the first unit of each cluster, Experiment II used the driver used in Experiment I. For the other units inside a cluster, we used complete programs, that is, we did not use stubs nor drivers. Some programs used in Experiment I were selected to study the influence of other contexts on unit testing. Notice that the program *dodash* is called in two different contexts and belongs to two different clusters (see Table 3).

During Step 4, additional test cases were generated to cover the remaining elements of every unit in each cluster. Table 3 presents the final coverage obtained. Observe that the number of required elements of *makepat* and *getfns* inside a cluster is greater due some potential associations with respect to global variables that are not required when the unit is tested alone.

Table 1. Experiment I - Main Results

	Coverage: Executed Elem. / Total of Required .Elem.	
Unit	PU	PDU
archive	17/17	17/17
append	38/49	20/43
change	37/38	24/27
ckglob	34/35	20/27
cmp	10/13	9/12
command	44/47	42/61
compare	38/38	34/64
compress	36/39	24/34
dodash	33/33	21/21
edit	110/130	86/346
entab	57/80	29/70
expand	25/39	15/25
getcmd	119/119	119/119
getdef	56/59	40/49
getfn	16/18	13/20
getfns	37/44	25/34
getlist	58/82	38/102
getnum	8/8	7/12
getone	59/62	30/177
gtext	27/33	12/22
makepat	134/207	95/253
omatch	21/33	18/43
optpat	12/13	8/21
spread	54/61	33/47
subst	176/219	87/322
translit	133/139	126/333
unrotate	80/94	41/70
Total	1469/1749	1033/2371

The task of generating test cases to satisfy a criterion is more difficult when the unit is inside a cluster. Semantic aspects of different programs make difficult to determine test cases that cause the execution of a particular path in a unit inside a cluster. Moreover, notice that the number of infeasible paths is very high for both experiments: 1338 in a total of 2371 (56.4%) in Experiment I and 411 in a total of 880 (46,7%) in Experiment II. If we consider, in Experiment I only the programs used in Experiment II, the percentage of infeasible elements found in Experiment II is greater than that of Experiment I, as some elements could not be exercised due to the restrictions imposed by the calling programs. The final coverages obtained for both experiments are different. In the next section, we discuss these results and present a classification for infeasibility: intrinsic and extrinsic.

4. Classification

The infeasibility of a path is related to semantic aspects of the program. In the experiments described in the last section, the infeasible elements were manually determined – a costly, fault-prone and tedious task! This

Table 2. Number of Predicates in Potential-du-paths of Experiment I

nprd.	0	1	2	3	4	5	6	7	8	9	10	11
infeas.	0	19	40	37	83	109	190	228	105	109	130	111
feas.	19	104	104	101	115	106	117	96	50	44	61	33
Total	19	123	144	138	198	215	307	324	155	153	191	144

npred	12	13	14	15	16	17	18	Total
infeas.	57	31	24	40	14	5	6	1338
feas.	25	16	10	17	8	5	2	1033
Total	82	47	34	57	22	10	8	2371

Table 3. Experiment II - Main Results

	Coverage: Executed Elem. / Total of Required Elem.	
Cluster	PU	PDU
archive	17/17	17/17
getfns	37/45	25/34
command	44/47	42/61
getcmd	119/119	119/119
optpat	12/13	8/21
makepat	133/209	95/253
dodash	33/33	19/21
translit	133/139	126/333
dodash	33/33	18/21
Total	562/655	469/880

section reports the main causes of infeasibility and their dependences. Based on Hedley and Hennel, the causes found are classified into three categories:

a. Dependence among predicates in a path: A path can include two or more predicates which are the same or are opposite; the evaluation of one implies the evaluation of the other. For example, in the program *getlist* (Figure 1), the predicates of Nodes 9 and 11 are dependent. The sub-path given by the sequence (9, 10, 11, 13) is infeasible. If edge (9, 10) is executed, the sequence (9, 10, 11, 12, 13) is also executed.

b. Inconsistent values of variables in a predicate: A variable in a predicate is defined with a value different from that which causes the path to be executed. These paths are classified into two groups:

b1. The predicate is in a *while* or *repeat* statement. In this case the variable controls a loop. The variable *done* in the program *makepat* (Figure 2) controls the loop headed by Node 2. Sub-path (13, 15, 17, 18, 19, 20, 21, 23, 2, 3) is infeasible. At Node 13 the value of *done* is 1. There are no definitions of *done* through (15, ..., 2); hence, the loop is exited.

b2. The predicate is in an *if* or *case* statement. The parameter *j* in the program *dodash* (Figure 3) is defined in *makeset* with value 0. Sub-path (1, 2, 3, 5, 7, 9) is infeasible, as there is a redefinition of *j* through the sequence (1, .. ,7) and the predicate $(j > 0)$, associated to the edge (7,9), can not be satisfied.

c. Combinations of the causes from categories above: The following example illustrates this category. The sub-path given by the sequence (3, 5, 7, 8, 9, 10, 11) in the program *optpat* (Figure 4) is infeasible due to causes from categories *a* and *b2*. The string *pat* is a global variable initially defined with ENDSTR. If *makepat*, called in Node 7, defines *pat*, the edge (10,11) in *optpat* is not executed. The loop headed by Node 2 in *makepat* is executed at least once, when (1,3,5,7) is executed in *optpat*. This is because the predicate of that loop is dependent on the predicates of edges (3,5) and (5,7) in *optpat*. When these edges are executed the loop is also executed, since there is a correspondence between $lin[*i+1]$ and $arg[i]$, and between $lin[*i]$ and $delim$. The program *addstr* adds to *pat* a character different from ENDSTR when $j < MAXPAT$. As $j = 0$ and $j < MAXPAT$ when *makepat* begins, in the first path through the loop headed by Node 2, Node 13 must be executed to avoid calling *addstr*. But in this first time $i = start$ and the condition of Node 11 is false. Thus, there is no way to avoid defining *pat* in *makepat* and, consequently, to execute edge (10,11) in *optpat*.

An infeasible sequence such as (9, 10, 11, 13), found in *getlist*, is called "infeasible pattern" [32]. If a path includes an infeasible pattern it is infeasible. Determining infeasible patterns helps the task of determining infeasible elements; a large number of infeasible elements can be identified using a pattern. Infeasible patterns can be classified into the same categories used to classify infeasible paths.

Another classification for infeasibility concerns the context of programs. For example, the number of infeasible paths found in *dodash* is different in both experiments.

In Experiment I, all the required elements in *dodash* are exercised when using the driver. The driver is the main program in Figure 3. The same does not happen in Experiment II. The number of infeasible elements depends on the cluster to which *dodash* belongs and on its calling context.

Differently of the classification proposed by Hedley and Hennel, we are also interested in the interdependencies among the units being tested. The results of Experiment II leads to the classification of infeasibility of a path or pattern as intrinsic or extrinsic. **Intrinsic Infeasibility** exists independently of the context. It is caused by internal features. The example used in Category *a* (Dependence among predicates in a path) is an intrinsic infeasibility of *getlist*. **Extrinsic Infeasibility** is imposed by the testing context. Infeasibility may be dependent on the calling program context. The example used in Category *b2* is an extrinsic infeasibility of *dodash* with respect to the calling program *makeset*. Infeasibility can also be dependent on the called program context. The example used to illustrate Category *c* is an extrinsic infeasibility of *optpat* imposed by the called program *makepat*.

Table 4 presents a summary of the number of infeasible paths and infeasible patterns found in Experiment II. For example, in the first cluster, there are 9 infeasible paths and 4 patterns, all in *getfns*. They belong to Category *a* and are infeasible due to intrinsic characteristics. It is possible for a path or pattern to be classified into more than one category. The result presented concerns the first cause identified. The most common category of infeasible paths is Category *a* (231 of 411 (56%)) and most of them were generated by intrinsic features of the unit (348 of 411 (85%)).

5. Prediction

As expected, in the experiments described previously, it was more difficult to generate test cases to execute a path with a large number of predicates and to determine its infeasibility. The greater the number of predicates in a path the greater the difficulty of satisfying the combination of these predicates and executing the path. This fact motivated a study, based on Malevris' work, to check the influence of the number of predicates on the infeasibility of a path. In spite of being intuitive, the influence must be validated statistically.

To study this influence, polynomial and exponential models were explored considering the interval $1 \leq q < 12$. (Recall that q is the number of predicates in a path). Given that only three programs have potential-du-paths with $q \geq 12$, all the potential-du-paths of these programs were not considered. The study was accomplished using Table 5, which is different from Table 2, to avoid the influence of particular characteristics of those three programs. The first two columns (0 and 1 predicate) were merged. The values of Table 5 are depicted in Figure 5. Figure 5 shows the influence of the number of predicates on the number of infeasible paths. The greater the number of predicates the greater the percentage of infeasible paths found in the experiment for $1 \leq q < 12$. These results, however, should be further investigated in a broader selection of programs.

Malevris et al used paths generated to cover LCSAJ testing [24]. Our study used potential-du-paths, required by the all-potential-du-paths criterion. The first step, similarly to Malevris', is to explore the hypothesis, H_0, of the existence of equal proportions of feasible paths for all q, where q is the number of predicates. H_0 true means that there is no relation between number of predicates and feasibility. A χ^2 test on data from Table 2 resulted in $\chi^2_\upsilon(0.005) = 35.718$ and $\upsilon = 17$, where υ is the number of degrees of freedom. There is a standard χ^2 table and a value $\chi^2_\upsilon(\alpha)$ such that if $\chi^2 > \chi^2_\upsilon(\alpha)$ the hypothesis H_0 can be rejected with probability α. From the standard table $\chi^2_{11}(0.005) = 35.718$ and H_0 can be confidently rejected meaning that there is a relation between the number of predicates of a path and its feasibility.

In the second step, a least squares fitting process of the function $f = ke^{\lambda q}$ gives the model $p = 0.965e^{-0.160q}$, with $r^2 = 94.4$, where p is the probability that a path with q predicates is feasible. For further details see [31].

This model validates Malevris' results within the context of data flow based criteria and shows that the number of predicates in a path can be used as a metric for predicting its infeasibility. The greater the number of predicates in a path, the greater the probability of the path be infeasible.

The relation between the number of infeasible potential-du-paths and other characteristics of the program being tested has also been studied. These characteristics are: number of nodes, number of variables and number of variable definitions. The obtained models show that all these characteristics influence the number of infeasible potential-du-paths. These models are described by Maldonado [22].

This kind of information is relevant for establishing strategies to generate data-flow adequate test case sets, specifically for selecting complete paths associated to a data-flow required testing element. For instance, a given strategy may assign a higher priority to paths with a lower number of predicates, as they have a higher probability of being feasible, given by the model. To illustrate this

```
 int getccl(char *arg, int *i, char *pat, int *j)}
 { int k,jstart;
   *i = *i + 1;
   if (arg[*i]==NEGATE)
   {
     addstr(NCCL,pat,j,MAXPAT);
     *i = *i + 1;
   }
   else
     addstr(CCL,pat,j,MAXPAT);
   jstart = *j;
   addstr('0',pat,j,MAXPAT);
   dodash(CCLEND,arg,i,pat,j,MAXPAT);
   k = *j-jstart-1;
   pat[jstart] = k;
   return(arg[*i] == CCLEND);
 }

int makepat(char *arg, int start, int delim, char *pat)
/*  1 */ { int i,j,lastj,lj,done;
/*  1 */   j = 0; i = start; lastj = 0; done = 0;
/*  2 */   while((!done)&&(arg[i]!=delim)&&(arg[i]!=ENDSTR))
/*  3 */   { lj = j;
/*  3 */     if (arg[i]==ANY)
/*  4 */       addstr(ANY,pat,&j,MAXPAT);
/*  5 */     else if ((arg[i]==BOL)&&(i==start))
/*  6 */           addstr(BOL,pat,&j,MAXPAT);
/*  7 */          else if ((arg[i]==EOL)&&(arg[i+1]==delim))
/*  8 */                addstr(EOL,pat,&j,MAXPAT);
/*  9 */               else if (arg[i]==CCL)
/* 10 */                      done = getccl(arg,&i,pat,&j);
/* 11 */                    else if((arg[i]==CLOSURE)&&(i>start))
/* 12 */                         {lj = lastj;
/* 12 */                          if((pat[lj]==BOL)||(pat[lj]==EOL)
                                      || (pat[lj] == CLOSURE))
/* 13 */                            done = 1;
/* 14 */                          else
/* 14 */                            stclose(pat,&j,lastj);
/* 15 */                         }
/* 16 */                         else
/* 16 */                         { addstr(LITCHAR,pat,&j,MAXPAT);
/* 16 */                          addstr(esc(arg,&i),pat,&j,MAXPAT);
/* 16 */                         }
/* 17                            end if 11 */
/* 18                       end if 9 */
/* 19                  end if 7 */
/* 20             end if 5 */
/* 21        end if 3 */
/* 21 */     lastj = lj;
/* 21 */     if (!done)
/* 22 */       i++;
/* 23 */   }
/* 24 */   if ((done)||(arg[i]!=delim))
/* 25 */     return -1;
/* 26 */   else if (!addstr(ENDSTR,pat,&j,MAXPAT))
/* 27 */          return -1;
/* 28 */        else
/* 28 */          return (i);
/* 29 */ }
```

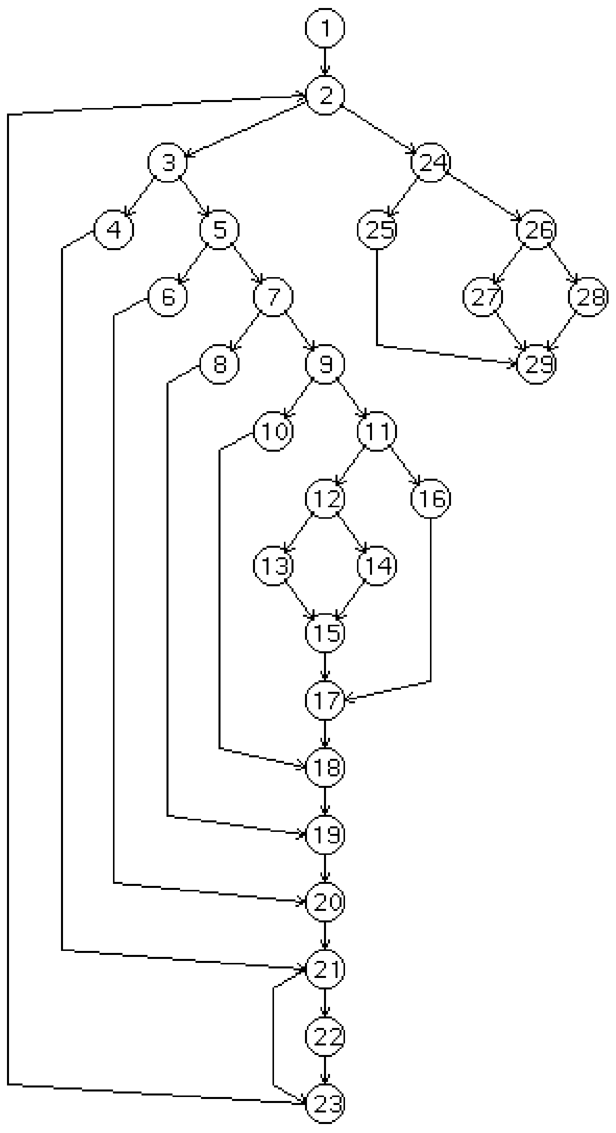

Figure 2. Program Makepat

```
void dodash(int delim, char * src, int * i, char * dest, int * j, int maxset)}
/*  1 */ { int k;
/*  2 */   while((src[*i]!=delim)&&(src[*i]!=ENDSTR))
/*  3 */   { if (src[*i]==ESCAPE)
/*  4 */       addstr(esc(src,i),dest,j,maxset);
/*  5 */     else if (src[*i]!=DASH)
/*  6 */            addstr(src[*i],dest,j,maxset);
/*  7 */          else if ((*j<=0)||(src[*i+1]==ENDSTR))
/*  8 */                 addstr(src[*i],dest,j,maxset);
/*  9 */               else if ((isalnum(src[*i-1]))&&(isalnum(src[*i+1]))
                                 && (src[*i-1]<=src[*i+1]))
/* 10 11 12 */              { for (k = src[*i-1]+1;k<=src[*i+1]; k++)
/* 12 */                       addstr(k,dest,j,maxset);
/* 13 */                     *i = *i + 1;
/* 13 */                    }
/* 14 */                    else
/* 14 */                      addstr(DASH,dest,j,maxset);
/* 15                       end if 9 */
/* 16                  end if  7 */
/* 17             end if 5 */
/* 18        end if 3 */
/* 18 */     *i= *i + 1;
/* 18 */   }
/* 19 */ }

void main(int argc, char *argv[])
{ int i,j;
  char str[MAXSTR], dest[MAXSTR];
  if (argc<=3)
  { printf("Erro numero de argumentos");
    exit(1);
  }
  strcpy(str,argv[1]);
  i = atoi(argv[2]);
  j = atoi(argv[3]);
  dodash(ENDSTR,str,&i,dest,&j,MAXSTR);
  dest[j] = ENDSTR;
  puts(dest);
}
```

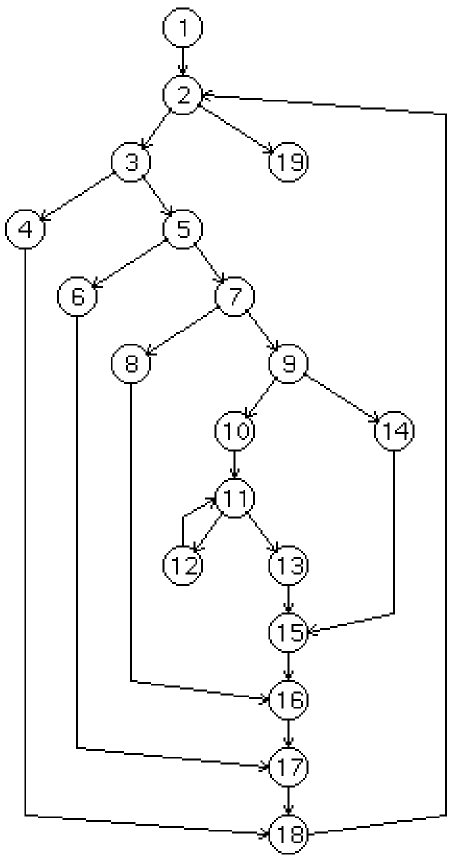

Figure 3. Program Dodash

Table 4. Infeasible Paths Found in Experiment II

Cluster	# Infeas./ Req.Elem. (%)	Infeasibility: # Path/ #Patterns						
		Category				Dependence		
		a	b1	b2	c	intrsc.	extr.- called	extr.- calling
archive	0/17 0%							
getfns	9/34 26.47%	9/4				9/4		
command	19/61 31.1%	19/4				16/3	3/1	
getcmd	0/119 0%							
optpat	13/21 56%			9/6	4/1	9/6	4/1	
makepat	158/253 62.45%	114/12	14/1	18/5	12/4	156/21		2/1
dodash	2/21 10%			2/1				2/1
translit	207/333 62.76%	89/2		63/2	55/5	156/6	51/3	
dodash	03/21 14.29%			3/1				3/1
Total	411/880 46.7%	231/22	14/1	95/22	71/10	348/40	56/5	7/3

aspect, consider again program *optpat* in Figure 4. The association $< 1, (10, 11), lin >$ is required by the all-potential uses criterion. The paths candidate to cover that association are given in Table 6, which also presents the number of predicates in each path and its probability of being feasible, given by the exponential model. Using the strategy "fewer number of predicates", one of the paths with $p = 0.60$, that is, with higher probability of being feasible, would be selected. Notice that this strategy avoids selecting the infeasible paths: (1, 3, 5, 7, 8, 9, 10, 11, 12, 13, 15) and (1, 3, 5, 7, 8, 9, 10, 11, 12, 14, 15).

6. Identification

Determining infeasibility of an element required by a structural criterion is an undecidable question. Based on Frankl's work and on the experience conducting experiments with testing criteria an extension to Poketool was proposed. This extension helps the identification and automatic elimination of infeasible elements.

Figure 6 presents a simplified module diagram of Poketool. *Poketool* module translates the source code into an intermediate language which allows abstract information on the control flow of the program; the program graph is generated from this intermediate language representation. *Poketool* module also analyses the code and generates the list of required elements and the instrumented program.

Pokeexec module executes the executable program with test cases provided by the tester and produces the executed paths. *Pokeadeq* module does the adequacy analysis with respect to a chosen criterion and generates the list of unexercised elements and a coverage measure.

To deal with infeasibility two new modules were proposed: *Pokeheuris* and *Pokepattern*. *Pokeheuris* helps the task of determining infeasible potential associations using Frankl's heuristics. *Pokepattern* manipulates infeasibility patterns and eliminates infeasible elements from the list of required elements.

Frankl's heuristics are applied to check the infeasibility of a given association. To apply Frankl's heuristics, first we determine all the paths candidate to cover an association. Next, we try eliminating the infeasible candidate paths by analysing the loops in the program. If a loop is executed at least once and all the paths through this loop that redefine the predicate variables also redefine the variables of the association, there will be no paths covering the association; all the paths can be eliminated and the association is infeasible. For other cases, nothing about the association infeasibility can be concluded.

For example, consider the control flow graph of program *getlist* in Figure 1 and the association $(1, 10, line2)$. The variables defined in each node are presented in the graph. The loop headed by Node 2 is executed at least once because $(!done)$ is $true$ in Node 1; hence, (1, 2, 9, 10) is infeasible. All the remaining paths candidate to cover this association include paths through the loop

```
stcode optpat(char *lin, int *i)
/*  1 */ { if (lin[*i]==ENDSTR)
/*  2 */     *i = -1;
/*  3 */   else if (lin[*i+1] == ENDSTR)
/*  4 */          *i = -1;
/*  5 */        else if (lin[*i+1]==lin[*i])
/*  6 */               *i = *i + 1;
/*  7 */             else
/*  7 */                *i =  makepat(lin,*i+1,lin[*i],pat) ;
/*  8                    end if 5 */
/*  9                end if 3 */
/* 10 */   if (pat[0]==ENDSTR)
/* 11 */     *i = -1;
/* 12 */   if (*i==-1)
/* 13 */   { pat[0] = ENDSTR;
/* 13 */     return ERR;
/* 13 */   }
/* 14 */   else
/* 14 */     return OK;
/* 15 */ }
```

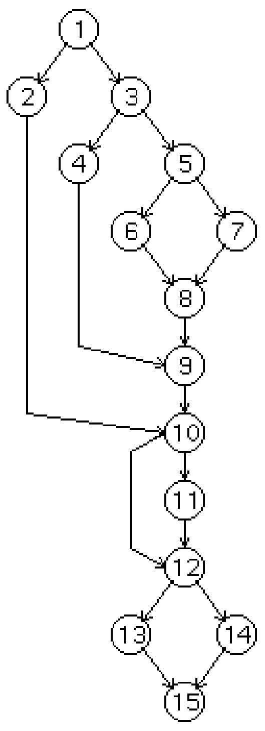

Figure 4. Program Optpat

Table 5. Number of Predicates in Potential-du-paths with $1 \leq q < 12$

nprd	1	2	3	4	5	6	7	8	9	10	11	12	Total
infeas.	19	40	37	78	91	140	156	57	49	22	36	0	725
feas.	78	78	74	87	76	78	52	25	15	16	0	0	598
Total	97	118	111	165	167	218	208	82	64	38	36	0	1323

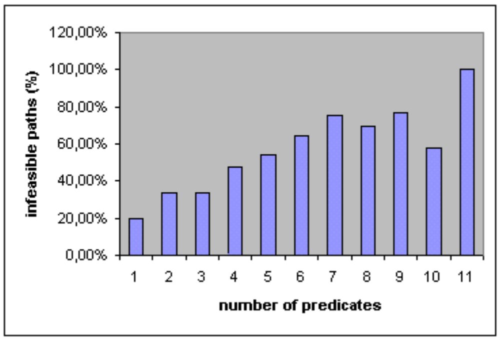

Figure 5. Number of Predicates and Percentage of Infeasible Paths

Table 6. Paths to Cover the Association <1,(10,11),lin>

Path	number of predicates	Probability p
(1, 2, 10, 11, 12, 14, 15)	3	0.60
(1, 2, 10, 11, 12, 13, 15)	3	0.60
(1, 3, 4, 9, 10, 11, 12, 14, 15)	4	0.50
(1, 3, 4, 9, 10, 11, 12, 13, 15)	4	0.50
(1, 3, 5, 6, 8, 9, 10, 11, 12, 14, 15)	5	0.43
(1, 3, 5, 6, 8, 9, 10, 11, 12, 13, 15)	5	0.43
(1, 3, 5, 7, 8, 9, 10, 11, 12, 14, 15)	5	0.43
(1, 3, 5, 7, 8, 9, 10, 11, 12, 13, 15)	5	0.43

headed by Node 2. All of them can be eliminated as all paths inside that loop redefining the variable *done* in the predicate in Node 2 also redefine the variable $line2$. Thus, the association is infeasible.

Pokeheuris uses data generated by *Poke-tool* module such as symbolic information about the program, mainly about the predicates, and paths that are candidates to cover an association. If module *Pokeheuris* can conclude something about such infeasibility the association is an infeasibility pattern and the module *Pokepattern* is called. If the tester identifies an infeasible pattern, he can provide this information to the *Pokepattern* module. *Pokepattern*, for an infeasible pattern and a criterion, identifies the infeasible required elements and updates the coverage result.

7. Infeasibility in a Broader Data-Flow-Based Context

The same problems concerning infeasibility in the unit and cluster testing appear when applying data-flow based testing in a broader context such as integration testing, test of web applications, parallel and object-oriented software testing, and specification testing.

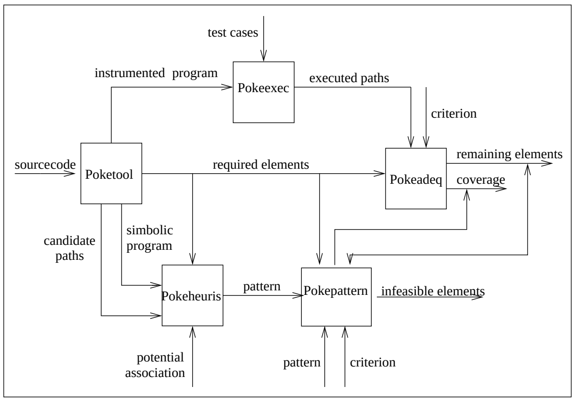

Figure 6. Poketool Modules

Based on a preliminary study, we claim that all the background, concepts and information available at the unit testing phase and all the results obtained with the experiments described herein are relevant and can be taken into consideration to reduce the effort and overcome the limitations imposed by infeasibility in a broader data-flow-based testing context. This section illustrates our claim for integration and object-oriented testing.

7.1. Inter-procedural Infeasibility

An inter-procedural association or path is infeasible if there is no complete feasible inter-procedural path that covers it. Any inter-procedural path that contains an intrinsic or extrinsic infeasible pattern is also infeasible.

Consider again programs *makeset* and *dodash* (Figure 3). Variable *j* is defined in Node 1 ($1m$ in *makeset*) and used in Node 12 ($12d$ in *dodash*). An inter-procedural association $< 1m, 12d, j >$ is required but it is infeasible. All paths that would cover it are infeasible because they necessarily include the infeasible sequence (1, 2, 3, 5, 7, 9) intrinsic to program *dodash*.

The same classification of infeasibility proposed for unit testing can be used for integration testing. Infeasibility is generated by the same basic causes identified at the unit level, presented previously, or by a composition of them.

Concerning prediction, it is expected that if Malevris' hypothesis holds at the unit level it also holds at the integration level. New studies are being carried out in this direction.

Concerning identification of infeasibility, the heuristics proposed by Frankl can also be applied to determine infeasible inter-procedural associations. An example of their application at the integration level can be extracted from Figure 1. In Node 1, the global variable *nlines* is defined and can be used in the program that calls *getlist*; an inter-procedural association is established. All the inter-procedural candidate paths that include pattern (1, 2, 9) can be eliminated; thus, the set of candidate paths is empty and the infeasibility of the association is established.

7.2. Infeasibility in Object Oriented Software

To test object-oriented software according to Harold and Rothermel [9], we have to cover associations in the following levels: intra-method, inter-method and intra-class.

The intra-method testing is equivalent to unit testing and the correspondence is direct. Intra-methods and intra-class testing occur during the integration of classes. In the same way, the causes of infeasibility and the categories presented in Section 4 are valid. Consider the example of Figure 7, extracted from [9]. The example contains a part of the class description of *SymbolTable* and its class call graph. In the intra-method testing each unit is tested separately. To perform inter-method testing on the *AddtoTable* method we integrate the methods *AddSymbol, Lookup, AddInfo, GetSymbol, Hash*, and test several calls to *AddTable*. For intra-class testing, we may select test sequences such as $<$*SymbolTable, AddTable, GetfromTable*$>$ and $<$*SymbolTable, AddTable, AddTable*$>$.

Consider these sequences to illustrate infeasibility and the intra-class association definition-use $<$3,(8,9),numentries$>$. The association needs to be covered in the first call of *AddTable*, as *numentries* is always redefined in a second call to *AddTable*. Thus, the feasibility of this association depends on the context of the method (or main program) that uses the sequences. If $tablemax >= 0$ the association is infeasible (cause *a*, extrinsic).

```
          public class SymbolTable {
/*  1 */    private TableEntry table[];
/*  2 */    private int numentries, tablemax;
/*  3 */    public SymbolTable (int m)
/*  4 */    { tablemax = m;
/*  5 */      numentries = 0;
/*  6 */      table = new TableEntry[tablemax];
/*  7 */    }
           ...
            public int AddTable(char symbol,char[] syminfo)
            {  int index;
/*  8 */       if (numentries<tablemax)
/*  9 */       { if (Lookup(symbol,index)==FOUND)
/* 10 */             return NOTOK;
/* 11 */           AddSymbol(symbol,index);
/* 12 */           AddInfo(syminfo,index);
/* 13 */           numentries++;
/* 14 */  return OK;
/* 15 */       }
/* 16 */       return NOTOK;
            }
           ...
          }
```

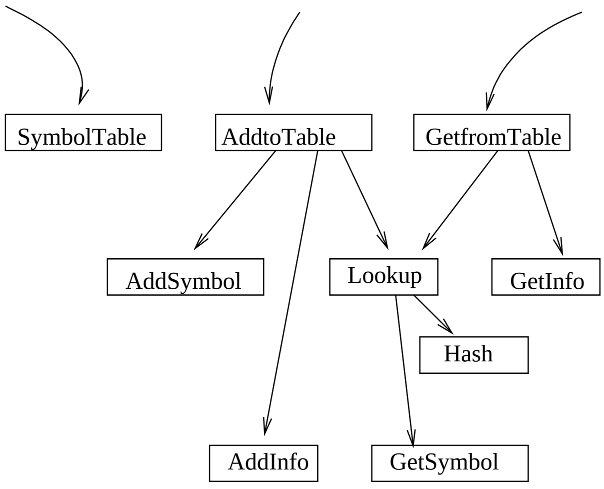

Figure 7. Class SymbolTable

8. Conclusions

This work concerns the issue of infeasible paths and structural testing, mainly data flow based testing. The described experiments show that most studied programs have infeasible paths.

Thus, facilities to deal with these aspects and to reduce the effect of infeasible paths on the testing activity are necessary.

Given these facts, infeasibility issues in the context of data flow based testing have been deeply investigated considering three basic research topics: classification, prediction and identification. These points have been addressed in the scope of unit testing as well as of integration and object-oriented software testing. Moreover, an extension to Poketool to provide facilities to deal with infeasible paths has been briefly presented.

The knowledge and information summarized in this paper can provide a good feedback to software development. For instance, the classification of infeasibility causes provides information for determining heuristics to eliminate infeasible paths as well as guidelines for program writing. Based on the main causes of infeasibility some guidelines can be established: 1) "Be careful about using a particular predicate more than once"; 2) "Be careful with definitions and consequent testing of control-loop variables"; and so on. Other contributions are intrinsic and extrinsic infeasibility and infeasible patterns proposed in this paper, since they are useful information in the context of software production and reuse and ease the automatic identification of infeasible paths.

Infeasible patterns permit to determine the infeasibility of a great number of elements. They capture semantic aspects of the program that can be used in regression and integration testing. Intrinsic patterns can be used in the integration of a program (method or process) with other programs. They are always valid, even if changes occur in the called or calling programs; they will be changed only if the unit (method or process) changes. It is a fact, a knowledge associated to the unit, that eases the task of automatically eliminating the infeasible elements, once the tester discovers an infeasible pattern.

The obtained results statistically validate the influence of the number of predicates of a path on its feasibility for $1 \leq q < 12$ (where q is the number of predicates in a path). Further studies are needed to investigate Malevris' hypothesis for programs containing paths with $q \geq 12$. A module named *Pokepaths*, which generates paths candidate to cover an element required by a given criterion supported by Poketool was developed. *Pokepaths* consider the number of predicates to select paths. We intend to implement Frankl's heuristics, as well as other facilities, to determine infeasibility in the integration level and in other data-flow based contexts.

In summary, the results presented in this paper contribute to the planning of the testing activity and to the establishment of testing strategies. For instance, the application of a context-independent unit testing strategy, similar to Experiment I, is very important: 1) you may exercise a greater number of required elements; and 2) you acquire more knowledge about the unit. The knowledge about infeasible elements and infeasible patterns can be used for unit cluster testing, where determining infeasibility is more difficult. It may also contribute to reduce costs in the debugging and maintenance phases and in regression testing.

References

[1] H. Agrawal and et al. Mining systems tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[2] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN'88 Conference on Programming Languages, Design and Implementation*, pages 47–56. Atlanta - Georgia, 22-24, June 1986.

[3] A. Carniello. *Teste Baseado na estrutura de casos de uso*. Master Thesis, DCA/FEEC/Unicamp, February 2003. (in Portuguese).

[4] M.L. Chaim. *POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados*. Master Thesis, DCA/FEEC/Unicamp, Campinas - SP, Brazil, April 1991. (in Portuguese).

[5] M. B. Dwyer, L.A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. On Software Engineering and Methodology*, Vol 13(4), October 2004.

[6] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data*. PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., October 1987.

[7] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. on Soft. Engin.*, Vol 17(6), June 1991.

[8] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *The Journal of Systems and Software*, 4:309–315, 1984.

[9] M.J. Harrold and G. Rothermel. Performing data flow on classes. In *ACM-SIGSOFT*, pages 154–163. New Orleans-USA, December 1994.

[10] M.J. Harrold and M.L. Soffa. Selecting and using data for integration testing. *IEEE Software*, Vol. 8(2):58–65, March 1980.

[11] D. Hedley and M.A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proceedings of VIII International Conference on Software Engineering*, pages 259–266. UK, 1985.

[12] W.E. Herman. Flow analysis approach to program testing. *The Australian Computer Journal*, Vol. 8(3):259–266, November 1976.

[13] R.M. Hierons, T.-H. Kim, and H. Ural. On the testability of SDL specifications. *Computer Networks*, 44(5):681–700, October 2004.

[14] J.R. Horgan and S. London. *ATAC- Automatic Test Coverage Analysis for C Programs*. Bellcore Internal Memorandum, June 1990.

[15] Z. Jin and A. J. Offut. Integration testing based on software couplings. In *Proceedings of the X Annual Conference on Computer Assurance (COMPASS 95)*, pages 13–23, Gaithersburg, Maryland, January 1995.

[16] P. Jorgesen and C. Erickson. Object oriented integration. *Communications of the ACM*, 9, September 1994.

[17] B.W. Kernighan and P.J. Plauger. *Software Tools in Pascal*. Addison-Wesley Publishing Company Reading, Massachusetts - USA, 1981.

[18] Y.W. Kim. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 145–155. IBM Press, Toronto-Canada, 2003.

[19] J.W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, Vol. SE-9(3):347–354, May 1983.

[20] U. Linnenkugerl and M. Mullerburg. Test data selection criteria for (software) integration testing. In *Proceedings of the First International Conference on Systems Integration*, pages 709–717. IEEE Press, Morristown, New Jersey, April 1990.

[21] C.H. Liu, D.C. Kung, P. Hsia, and C.T. Hsu. Structural testing of web applications. In *11th International Symposium on Software Reliability Engineering*, pages 84–96. IEEE Press, 2000.

[22] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, July 1991. (in Portuguese).

[23] J.C. Maldonado, M.L. Chaim, and M. Jino. Bridging the gap in the presence of infeasible paths: Potential uses testing criteria. In *XII International Conference of the Chilean Computer Science Society*, pages 323–340. Santiago, Chile, October 1992.

[24] N. Malevris, D.F. Yates, and A. Veevers. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, Vol. 32(2):115–118, March 1990.

[25] V. Martena, A. Orso, and M. Pezze. Interclass testing of object oriented software. In *International Conference on Engineering of Complex Computer Systems (ICECCS'01)*, pages 135–144. IEEE Press, Maryland- USA, December 2002.

[26] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, Vol. SE-2(4):308–320, December 1976.

[27] S.C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[28] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[29] F. Ricca and P. Tonella. Analysis and testing of web applications. In *23rd International Conference on Software Engineering (ICSE'01)*, pages 25–34. IEEE Press, Toronto-Canada, May 2001.

[30] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28(3):157–163, July 1988.

[31] S.R. Vergilio. *Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas*. Master Thesis - DCA/FEEC/Unicamp, Campinas - SP, Brazil, January 1992. (in Portuguese).

[32] S.R. Vergilio, J.C. Maldonado, and M. Jino. Infeasible paths within the context of data flow based criteria. In *VI International Conference on Software Quality*, pages 310–321. Ottawa-Canada, October 1996.

[33] S.R. Vergilio, S.R.S. Souza, and P.S.L. Souza. Coverage testing criteria for message passing parallel programs. In *IEEE Latin American Test Workshop (LATW'05)*, pages 161–166. Salvador -Bahia, Brazil, March 2005.

[34] P. Vilela, J.C. Maldonado, and M. Jino. Data flow based integration testing. In *XIII Brazilian Symposium on Software Engineering*, pages 393–409. Florianópolis, SC, Brazil, October 1999.

[35] E.J. Weyuker. An empirical study of the complexity of data flow testing. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 188–195. Computer Science Press, Banff - Canada, July 1988.

[36] E.J. Weyuker. The evaluation of program-based software test data adequacy criteria. *IEEE Transactions on Software Engineering*, Vol. SE-16(2):121–128, February 1988.

[37] E.J. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, Vol. SE-19(3):914–919, September 1993.

[38] L.J. White and E.I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, Vol. SE-6(3):247–257, May 1980.

[39] R-D. Yang and C-G. Chung. Path analysis testing of concurrent programs. *Information and Software Technology*, 34(1):101–130, January 1992.