Path Selection in the Structural Testing: Proposition, Implementation and Application of Strategies

Letícia M. Peres
Dinf/UFPR
mara@inf.ufpr.br

Silvia R. Vergilio
DInf/UFPR
silvia@inf.ufpr.br

Mario Jino
DCA/FEEC/UNICAMP
jino@dca.fee.unicamp.br

José C. Maldonado
ICMC-USP
jcmaldon@icmsc.sc.usp.br

Abstract

Structural testing criteria help the tester in the generation and evaluation of a test case set T. They are predicates to be satisfied to consider the testing activity ended and generally require the execution of a set P of paths, capable of exercising certain elements in the program under testing. Determining P is an important and hard task, and its automation is strongly desirable for easing the criteria application. This task can influence on the efficacy and on the testing effort and costs. This work explores the use of diverse programs characteristics to propose strategies for selection of testing paths. The work also describes a module that implements a framework for representation and automation of those strategies. Using this module, a testing procedure is presented and a strategy, that uses the number of predicates to select paths, is evaluated. The obtained results give some information about the main advantage of this strategy: to easy the automatic test data generation by reducing the number of selected infeasible paths.

Key Words: *Software Testing, Software Metrics, Structural Criteria, Selection of Test Paths,*

1. Introduction

Testing is an essential Software Engineering activity for software quality assurance. It is usually defined as the process of executing a program with the goal of revealing faults. Hence, a good test case is that one with high probability of finding an unrevealed fault. To guide the testing activity and to select "good test cases", some testing criteria have been proposed during the last decades.

Structural testing criteria are based on the knowledge of the internal structure of the program implementation. The most well known structural criteria are classified as control-flow and data-flow based criteria [10, 11, 15, 18]. Control-flow based criteria establish testing requirements (required elements) based on the control flow graph associated to a program. For instance, the all-edges criterion, also known as branch testing, requires that all edges of the graph be exercised at least once during program testing [18]. Data-flow based criteria, in addition to the control flow graph, use data flow information to establish the testing requirements [18,11]. For instance, the Potential Uses Criteria require the execution of paths from the graph that exercise the definition and the subsequent potential uses of program variables during testing.

A structural criterion is defined as a predicate to be satisfied to consider the testing activity ended. To satisfy a criterion, the execution of complete paths that cover the required elements is necessary. However, satisfying a criterion is not always possible, in due to infeasible paths. A path is infeasible if there is no set of values to the input variables that causes its execution. A required element is infeasible if all the paths that cover it are infeasible. Determining infeasible paths is an undecidable question [7]. The existence of infeasible paths is a problem for automatic satisfaction of the structural criteria, a task that spends a lot of effort and time. According to Bertolino and Marré [1,2], test path generation tools should be concerned with minimizing the number of infeasible paths generated. However, most tools do not offer mechanisms to reduce the effects of infeasibility.

Other important question with respect to satisfaction of a structural criterion is the selection of a path to cover a given required element. The set of cover paths can be infinite and the best paths should be selected. Selecting the best paths, which cover the elements required by a structural testing criterion, is the focus of this work. The term "best" is related to some underlying objectives, such as: 1) to decrease the effort spent in the application of structural criteria and in program testing; 2) to increase the efficacy of test data generated; and 3) to make easier the automatic generation of test data, reducing the number of infeasible paths.

With these objectives, we proposed in [16,17] the use of certain program characteristics to establish path selection strategies for satisfying a structural testing criterion. A framework for representing those strategies was also presented. The use of some characteristics as complexity, number of variables and testability, in strategies can allow an increase in the efficacy. Other characteristics as number of predicates and nodes can ease the automatic test data generation. The number of predicates of a path can be used in a strategy to reduce the number of infeasible paths [12, 19, 22].

A module, supporting the proposed framework was implemented. This module, called Poke-Paths, extends a useful testing tool, Poke-Tool, which supports distinct structural testing criteria. Poke-Paths offers a considerable economy of time and effort by generating cover test paths automatically.

This work describes the context of Poke-Paths and presents results from an empirical study. This study has two goals 1) to validate the framework implementation: as a result, a procedure of testing using Poke-Paths is proposed; and 2) to observe the "lower number of predicates" strategy: results are presented, with respect to the number of feasible paths selected by three strategies: lower number of predicates, greater number of predicates and random selection.

This paper is organized as follows. Section 2 presents the framework used for representing the path selection strategies considering the mentioned characteristics. Some strategies proposed in [16,17] are described using the framework. A complete example illustrates the "lower number of predicates" strategy, which is the focus of Section 4. Section 3 shows the module Poke-Paths and the testing procedure. Section 4 presents results from the empirical study using Poke-Paths. The conclusions and future works are presented in Section 5.

2. Representing diverse path selection strategies: the framework

This section presents the framework proposed in [16,17]. It permits the representation of diverse strategies that allow different program characteristics to be applied for selection of paths, depending on the testing objectives. The framework makes easier the presentation and automation of these strategies.

For the directed graph $G=(N,A,s,f)$, representing the program control flow, where: N is a set of vertices representing command blocks, from now on called nodes; A is the edge set representing the control flow program; s is the entry node of graph, i.e., the first command block of the program; and f is the final node of the graph.

A complete path is defined as a finite sequence of nodes from G, denoted by $(n_1, n_2, ..., n_{j-1}, n_j)$, where $n_1=s$, $n_j=f$, $j \geq 1$, and j is the size of the path. The weight of a path π is:

$$P_\pi = \sum_{i=1}^{j} \left(p(n_i) \right)$$

where n_i is the i^{th} node of the path and $p(n_i)$ is the *weight* of the node n_i, according to a path selection strategy S.

Let $\Pi_{R,C} = \{\pi_1, \pi_2, ..., \pi_{n-1}, \pi_n\}$ be a finite set of complete paths that covers the set R of elements required by a criterion C. A path selection strategy S is given by the pair $S=(p(n), e(\pi))$, where: $p(n)$ is a function which assigns a *weight* to the graph nodes and $e(\pi)$ is a function which selects a path π of $\Pi_{R,C}$. Thus, the set of paths selected by a strategy S is defined as: $S\Pi_{R,C} = \{x \in \Pi_{R,C} \mid e(x) = 1\}$, where e is a function that has value 1 if the path is chosen by the strategy and 0 otherwise.

In summary, a weight assignment is individually done to each graph node. This assignment is based on the analysis of the source code and on the control flow and data flow of the program. In the sequel, the sum of weights of each path is determined, allowing the selection of paths according to their summed weights; e.g., paths with the greatest or the least sums, i.e., paths having the characteristic associated to them in the greatest or the least degree.

Several programs characteristics can influence the testing activities. We propose in [16,17] the use of these characteristics to establish test path selections strategies. Next we represent some strategies using the framework; more details are found in [15,16]. Section3 presents implementation aspects of the framework, that is, of the pair S.

2.1.Complexity

It is given by software characteristics that affect the developer productivity in software composing, comprehension and maintenance [6], as well as software testing. The study of software complexity is based on well-known software metrics, widely applied for cost estimation; they can only be used after programs are coded, for instance, Cyclomatic Number [13], LOC (Lines of Code) [6], Npath [14], Software Science [8], Data Quantity [6], and Nesting Level [9].

Concerning path selection, complex paths according these metrics, affect developer's productivity, especially in code comprehension, which is crucial to test data generation. On the other hand, a path or program, which is more complex than others, maybe has a greater probability of containing faults among its commands and structures; this is due to a greater difficulty of code composition, comprehension and maintenance. Hence, the selection of both simple and complex paths can be interesting to the tester, either to ease the generation of test data or to enhance the efficacy of software testing. The framework represents all strategies that use the complexity metrics above mentioned. The next paragraph presents an example using LOC.

A strategy using LOC is represented by associating to each node the number of code lines. Thus, for each instance of the ordered pair S=(p(n), e(π)), p(ni) is the number of lines of code found in the block corresponding to node ni. The tester establishes the function e(π) that can select the path with the greatest weight or with the least weight, as desired.

2.2 Testability

According to Voas et al. [20], testability is determined by code structure, semantics and program input distribution. In functional testing with random input, a program has a high testability when faults are readily detected through this testing. Voas et al introduce Sensitivity Analysis, where "sensitivity" means a prediction of the probability that a fault will cause a failure in the software on a particular location with a specific input distribution.

Sensitivity analysis can be applied in the context of path selection by evaluating the frequency of execution of commands from the input of a random set of test data; the selected paths are those with less executed command blocks.

This approach must be applied in this strategy to find the values of weights. Test data are generated randomly and the program is executed on them. Each reached node and the number of times it is reached are recorded. The purpose of the strategy is to select paths that have the least average range. Thus, we define the ordered pair S=(p(n), e(π)) with $p(n_i)$ = average range of node n_i, and e(π) = function that selects the path with the least total weight, i.e., least P_Π. Two disadvantages of this strategy can be pointed out. Bertolino and Strigini [3] question some definitions given by Voas et al. and there is no empirical work applying Sensitivity Analysis. However, this strategy is useful as it selects paths containing nodes with low probability of being reached, given a testing criterion to be satisfied. This can increase the efficacy of testing.

2.3. Feasibility

A path is feasible if there is an input set of values that exercises it. Feasibility of paths is an un-decidable question and a limitation that makes difficult the automation of testing activities [18,12]. Malevris et al. [12,22] propose to minimize time and cost of branch testing by affirming that: the lower the number of predicates in the path the greater the probability of the path be feasible. A predicate is a simple boolean form in a condition. These authors propose a strategy of path selection with few predicates to reduce the number of selected infeasible paths. Vergilio et al. [19] validated the Malevris' hypothesis in the context of data flow testing criteria. The strategy "lower number of predicates" proposes the selection of paths with lower number of predicates because those paths have higher probability of being feasible [12,22]. According to the proposed framework, the pair S=(p(n), e(π)) is defined as $p(n_i)$ = number of predicates in the command block corresponding to node n_i, and e(π) = function that selects the path with least total weight, i.e., least P_Π.

The use of this strategy can reduces costs and effort spent in the criteria application by reducing the effects of infeasible paths. Due this, the "lower number of predicates" strategy was chosen to validate the module Poke-Paths that implements the pair S. An empirical study was accomplished and described in Section 4. Below, we present a more complete example applying this strategy.

The example uses the program *entab.c* (Figures 1 and 2). The ordered pair S represents the "lower number of predicates" (or lnp) strategy. Table 1 shows the weight assigned to each node, using the function p(n) defined for this strategy.

Consider R=(14,15), the element required by the criterion C = all-edges. Table 2 presents the set $\Pi_{R,C}$, four possible complete paths that cover R, and the weight P_π of each path. The path selected by the strategy S, lower number of predicates is π_3=(1, 2, 3, 7, 9, 14, 15) with lower P_π.

Table 1. Weights assigned to nodes according to the lower predicate strategy

p(1) = 0	p(6) = 0	p(11) = 0
p(2) = 0	P(7) = 1	p(12) = 0
p(3) = 1	P(8) = 0	p(13) = 0
p(4) = 1	P(9) = 1	p(14) = 1
p(5) = 0	p(10) = 1	p(15) = 0

Table 2. Elements of set $\Pi_{R,C}$ for the example and respective weights

	Path	Pπ
π_1	(1,2,3,4,6,3,7,9,14,15)	5
π_2	(1,2,3,4,5,6,3,7,8,7,9,10,11,13,14,15)	7
π_3	(1,2,3,7,9,14,15)	4
π_4	(1,2,3,7,8,7,8,7,9,14,15)	6

```
    void entab ()
1   {
1      int c, col, newcol;
1      int tabstops[MAXLINE];
1      settabs (tabstops);
1      col = 0;
2      do
2      {
3         newcol = col;
3         while ((c = getchar ()) == BLANK)
4         {
4            newcol++;
4            if (tabpos (newcol, tabstops))
5            {
5               putchar (TAB);
5               col = newcol;
5            }
6         }
7         while (col < newcol)
8         {
8            putchar (BLANK);
8            col++;
8         }
9         if (c != ENDFILE)
9         {
10           putchar (c);
10           if (c == NEWLINE)
11              col = 0;
12           else
12              col++;
13        }
14     }
14     while (c != ENDFILE);
15  }
```

Figure 1. The code of *entab*

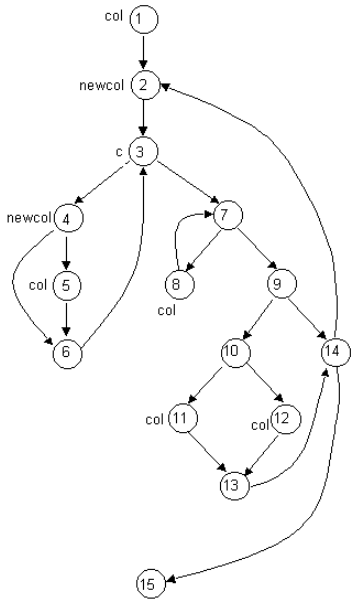

Figure 2. The control flow graph of *entab*

3. Automating the framework: module Poke-Paths

A module for automatic generation of test paths was developed to help the testing activity. This module, called Poke-Paths, implements the framework using the ordered pair S. This means that all strategies proposed in last section are implemented in this module. Poke-Paths extends a testing tool, named Poke-Tool [5]. Figure 3 illustrates the context of Poke-Paths inside Poke-Tool. Poke-Tool implements the Potential Uses Criteria Family [11] and the control-flow based criteria: all-edges and all-nodes. Among other functionalities, determines the elements required by these criteria.

Pokernel is responsible for the generation of elements required by the criteria, of the instrumented program and of the program control flow graph (CFG). Pokeexec supports the execution of the instrumented program, directing to files: input data, parameters, outputs, and the paths executed by the test cases. Pokeaval allows the evaluation of the coverage achieved by the applied test cases, for a given criterion. This module also produces the list of covered (executed) elements.

Routines that implement the heuristics proposed by Frankl and Weyuker [19] were incorporated to Poke-Tool to determine infeasible paths and associations. Pokepattern supports treatment of patterns of unfeasibility [19]. This module removes infeasible elements from the required element set. The user

identifies those elements and certain infeasible patterns that can help this task. The purpose is to detect and eliminate infeasible elements as soon as possible in the testing process to decrease the costs of test data generation.

Poke-Paths allows the generation of complete paths for elements required by different structural criteria. It also allows extension to other criteria, such as the Data Flow Based Criteria Family (DFCF)[18] or any criteria whose required elements are nodes, edges, definition-use association or paths.

Pokedata is a "test data generator" module that uses genetic algorithms and dynamic techniques to generate testing data for complete paths [4].

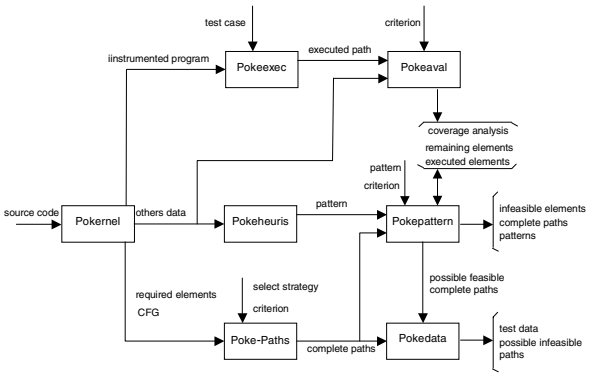

Figure 3. Poke-Tool environment

A Testing procedure using module Poke-Paths

1. Pokernel generates the required elements according to a given criterion, modifies the program under testing, and creates the instrumented program.

2. The tester generates an initial test case set Ti

3. Pokeexec executes the instrumented program with Ti, and saves the inputs, obtained outputs and executed paths.

4. Pokeaval gives the coverage for the chosen criterion and generates the list of uncovered required elements.

5. The tester adds new test cases to satisfy the criterion and makes the following steps until get the desired coverage or satisfy the criterion.

a) Application of heuristics for determination of infeasible elements: this step uses Pokeheuris. The required elements that are analyzed are those not executed until this moment.

b) Analysis of the remaining required elements and manual identification of infeasible elements. The patterns associated to this step are inserted in the file <function>.pattern, by Pokepattern. In this way, it is possible to maintain systematically the unfeasibility patterns identified to the program, to eliminate elements that contain a pattern of this file and to reevaluate the coverage.

c) Generation of complete paths to required elements not executed and not detected as infeasible. This is made by Poke-Paths. It is possible to generate paths with greater probability to be feasible with the lower predicate strategy, or other strategy defined in this paper and represented by the framework implemented by Poke-Paths.

d) Unfeasibility identification, using Pokepattern and <function>.pattern file. Some infeasible complete paths generated in last step can be eliminated.

e) Generation, using Pokedata, of test data to execute the complete paths.

f) Execution of test data found in the last step.

4. Applying strategies: an evaluation using Poke-Paths

Using Poke-Paths, a study was carried out with the goal of comparing three strategies: "lower number of predicates (or l.n.p.), the opposite strategy "greater number of predicates" (or g.n.p.), and the random selection (r.s.) of paths. In order to do this, two aspects were considered: number of feasible paths and efficacy, given in terms of the revealed errors.

We defined the gnp strategy using the ordered path S= (p(n), e (π)), where the function p(n) is the number of predicates of the command block corresponding to the node ni and e(π) is a function that selects the path with the greatest total weight (Pπ).

This application used three programs in C language, extracted from broadly known benchmarks, which have already been used in other software testing works [21,11,19]. Complete paths were generated to satisfy the all-edges and all-potential-uses criteria supported by Poke-Tool. All paths had their feasibility determined.

For each program the same number of paths were selected by all strategies. The column total of Tables 3 and 4 gives that number respectively for all-potential uses and all-edges criteria. Those tables also show the percentage of feasible paths selected by each strategy.

For example, during the application of all-potential-uses criterion (Table 3), the lnp strategy selected 546 paths for program cal, 58,42% of them are feasible. For three programs (*cal, entab, and expand*) the lnp strategy selected the greater percentage of feasible paths. Only for the program *comm*, this does not happen. In this case, the strategy gnp selected a greater percentage. The worst result for gnp strategy is the program expand. The random selection does not present the best percentage for any program.

Analyzing the code of the program *comm*, for which lnp presented a different behavior, we observe control flow structures that induces the lnp strategy not to select feasible paths. These structures, that can be detected and treated statically, are: "*for*" command, with a great number of entries determined in the code; and the infinite loop ("*while* (1)") with its output ("*exit(*0)") nested under several predicates.

Table 3: Percentile value of feasible paths, for all-potential-uses criterion.

Program	lnp %	gnp %	rs %	Total
cal	58.42	12.82	28.75	546
comm	19.88	57.61	22.51	2279
entab	46.24	24.20	29.57	186
expand	59.56	1.64	38.80	183
total	30.28	44.80	24.92	3192
total*	51.96	16.67	31.37	1122

Using a static determination and eliminating paths with these structures from the total number of paths, we obtained the percentages in the last rows of Tables 3 and 4 (total*). The lnp strategy, presents now better results for both criteria. It selects 51.96% of feasible paths among 1122 selected paths for satisfying all-potential uses criterion. The gnp strategy only 16.67% and the rs, 31.37% of feasible paths. Similarly, for all-edges criterion, the lnp selects 41.42% of feasible paths among 408 selected paths. The gnp strategy 25.74% and the rs, 32.84% of feasible paths.

Table 4: Percentile value of feasible paths, for all-edges criterion.

Program	lnp %	gnp %	rs %	Total
cal	43.79	25.44	30.77	169
comm	26.54	46.38	27.08	373
entab	37.29	28.81	33.90	59
expand	58.82	3.92	37.25	51
total	34.51	36.02	29.45	652
total*	41.42	25.74	32.84	408

One aspect that could be considered is whether there is a reduction of efficacy when selecting, between two paths, the one with lower number of predicates. Apparently, the greater the number of predicates the greater the probability of finding out faults in a program. A second application was conducted in order to observe those lnp aspects and details are in [16]. The application used the program cal and twenty incorrect versions derived by Wong [21]]. It does not point out any decrease of efficacy using the lnp strategy, even when compared to the random strategy.

5. Conclusion

The choice of a path selection strategy to be used with a given structural criterion may have some implications: reduction of effects caused by infeasible paths, easier automation of data generation, reduction of costs and time dedicated to this activity, reduction of the number of generated tests, and greater efficacy concerning revealed faults.

The strategies should always select paths from an initial path set, capable of covering the elements required by a given structural testing criterion. That happens because no specific analysis of testing coverage is proposed by the selection strategies. Such strategies were created to ease the application of the structural testing criteria without compromising the quality of software testing.

One of the most interesting ideas explored here is to apply characteristics of software and of testing in test path selection. Characteristics used in path selection strategies were complexity, testability and feasibility; any other characteristic that can be mapped to commands in the program source code can also be used and represented by the framework of Section 2. All strategies proposed in this paper are implemented in the module Poke-Paths. Yet, these strategies can also be associated to the lower number of predicate strategy through a combined proportional sum of weights. Another function p(n) of the ordered pair S could also be created to provide a tiebreak criterion for path selection, as a way to increase the probability of selecting a feasible path.

A testing procedure, using the module Poke-Paths was presented. This procedure provides automatic generation of test paths for satisfying the structural testing criteria implemented by the tool Poke-Tool.

The module Poke-Paths was validated and used to apply the lower number of predicates strategy. The work presented some evidences of the practical application of

this strategy and its capacity of selecting feasible paths and revealing faults. In addition to that, it was observed that it is possible to detect and treat some structures that hinder the selection of feasible paths when applying the lower number of predicate strategy, to increase the probability of selecting feasible paths. Furthermore, some evidences are given that, the lower number of predicate path selection strategy does not imply a reduction of its capacity of revealing faults, according to the application carried out.

More details of these empirical works can be found in [16]. The module Poke-Paths with the environment Poke-Tool constitutes a useful test bed for, in the future, identifying the correlation between metrics and success of the testing strategies here proposed. In order to do this, new experiments should be accomplished.

References

[1] A. Bertolino and M. Marré. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. IEEE Transactions on Software Engineering, 20(12):885-899, December 1994.

[2] A. Bertolino and M. Marré. Unconstrained Duas and Their Use in Achieving All-Uses Coverage. In ISSTA '96- ACM, San Diego, CA, 1996.

[3] A. Bertolino and L. Strigini. On the Use of Testability Measures for Dependability Assessment. IEEE Transactions on Software Engineering, 22(2):97-108, February 1996.

[4] P.M.S. Bueno. Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software. Master's thesis, DCA/FEEC/Unicamp, Brazil, 1999 (in Portuguese).

[5] M.L. Chaim. POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados. Master's thesis, DCA/FEE/Unicamp, Campinas/SP, Brazil, 1991.(in Portuguese)

[6] S.D. Conte, H.E Dunsmore, and V.Y. Shen. Software Engineering Metrics and Models. Benjamin/Cummings Publishing Company, Inc., 1986.

[7] P.G. Frankl and E.J. Weyuker. An Applicable Family of Data Flow Testing Criteria. IEEE Transactions on Software Engineering, 14(10):1483-1498, October 1988.

[8] M. Halstead. Elements of Software Science. North-Holland, 1977.

[9] W. Harrison and K.I. Magel. A Complexity Measure Based on Nesting Level. SIGPLAN Notices, 16(3), March 1981.

[10] J.W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. IEEE Transactions on Software Engineering, 9(3), May 1983.

[11] J.C. Maldonado. Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. Doctorate thesis, DCA/FEE/Unicamp, Brazil 1991.(in Portuguese)

[12] N. Malevris, D.F. Yates, and A. Veevers. Predictive Metrics for Likely Feasibility of Program Paths. Information and Software Technology, 32(2):115-119, March 1990.

[13] T.A. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 2(4):308-320, December 1976.

[14] B.A. Nejmeh. NPATH: A Measure of Execution Path Complexity and Its Applications. Comm. of the ACM, 31(2):188-210, February 1988.

[15] S.C. Ntafos. A Comparison of Some Structural Testing Strategies. IEEE Transactions on Software Engineering, 14(6): 868-873, June 1988.

[16] L.M. Peres. Estratégias de Seleção de Caminhos no Contexto de Critérios Estruturais de Teste. Master's thesis, DCA/FEEC/Unicamp, Brazil, 1999 (in Portuguese).

[17] L.M. Peres et alli. Path Selection Strategies in the Context of Software Testing Structural Criteria. 1st IEEE Latin American Test Workshop, Rio de Janeiro, Brazil, 2000.

[18] S. Rapps and E.J. Weyuker. Data Flow Analysis Techniques for Test Data Selection. Proc. Int. Conf. Software Engineering, Tokyo, Japan, September 1982.

[19] S.R. Vergilio, J.C. Maldonado, and M. Jino. Infeasible Paths within the Context of Data Flow Testing Criteria. Proc of the Sixth International Conference on Software Quality, Ottawa, Canada, October 1996.

[20] J. Voas, L. Morell, and K. Liller. Predicting where faults can hide from testing. IEEE Software, pages 41-47, March 1991.

[21] W.E. Wong. On Mutation and Data Flow. PhD Thesis, Department of Computer Science, Purdue University, West Lafayette-IN, USA, December 1993

[22] D.F. Yates and N. Malevris. Reducing The Effects Of Infeasible Paths In Branch Testing. ACM SIGSOFT Software Engineering Notes, 14(8):48-54, December 1989.