An assessment of the number of paths needed for control flow testing

N. Malevris
Department of Informatics
Athens University of Economics and Business
76 Patission Street, Athens 104 34 Greece, Tel. +30-1-8203126
Fax: +30 -1-8226204, e-mail: nicos.malevris@aueb.gr

Abstract

Generating test data does require a great amount of effort. The estimation of this effort beforehand is difficult and unclear when generating test data for exercising certain program structural characteristics. Methods that try to generate test data based on a minimum number of paths fail to estimate a lower bound in advance. When some of the paths they generate are infeasible, this situation is worsened. Thus, the generation of the test paths is important due to the presence of infeasible paths. Knowing the effort involved in deriving the appropriate test paths is a very useful exercise. This effort is linked directly to the estimation of the effort involved in achieving a high level of reliability. In this paper methods that do suggest a lower bound and an experimental upper bound are presented. Their ability of circumventing the incidence of infeasible paths is discussed and the lower and upper bounds of the cost and effort for test data generation analysed.

Keywords

Software testing effort, white box testing, infeasible paths, test path selection

1 INTRODUCTION

The presence of errors in the software contributes to the high level of its unreliability. The removal of errors does reduce this level by establishing the confidence of the software tester, hence increasing its reliability.

In Veevers (1991) the direct analogy of coverage metrics to the contribution of software reliability is exploited. He suggests that the increase in covering program statements, branches, LCSAJs, does contribute to the increase of reliability. If this argument holds, and there is no reason why it should not, as Hennell (1991) also claims in his paper, it should be very useful to estimate the effort required in achieving a high level of reliability.

The coverage of the program structural characteristics as demonstrated in the present paper can be achieved by generating test paths and exercising them with test data. The problem of test data generation, however, during the testing phase of the software life cycle is two fold. Firstly, it is required to generate test data that will exercise certain features of the program under test and decide upon its situation by recording the level of satisfaction of the criterion, set during the test. Secondly, since test data generation is a very difficult and costly process, it is required that the cost overheads are kept to a minimum. In software testing it is very important to know what the bounds for test data generation are. Knowing the bounds, one can decide on the level of effort during the testing phase and on the cost involved for the whole testing exercise. In this paper, the lower and upper bounds for test data generation are examined in an attempt to formulate the testing exercise aiming at high quality of software. The question of minimising over the number of test cases needed to be generated can be translated into the number of paths required to be traversed along the program flow graph. This approach requires that a graph, the control flow graph, Yates and Hennell (1985) is deduced and that the sought set of paths is generated by applying graph theoretic techniques.

Surprisingly, although the number of automated tools has increased in the literature, there is still little, if not none at all, mention on the methodical way which they employ for the generation of the appropriate paths for inclusion in their tools and test data generation. Of these methods, some are based on the intuition of the tester, others on the generation of a minimum possible number of paths.

Those based on the intuition of the tester require that during program execution, the tester may supply the system with a randomly selected data set from the program input space, or with a specific data set that will force the execution of a particular path in order to cover certain program features such as statements, branches, LCSAJs (An LCSAJ is a sequence of consecutive statements in the program text, starting at an entry point or after a jump and terminating with a jump or at an exit point), etc. This latter execution of a path is usually based upon the derivation of a data set with the use of a symbolic evaluator or executor.

Software testers have been attracted by minimum path generation methods. The reason being that the testing exercise has always been regarded as a very expensive process. The generation of test data is a very costly experience in the software life cycle and consequently, if the cost overheads of generating test data could be reduced to a possible minimum, this would of course save effort and expenses.

There exists a number of methods that generate such a minimum set of paths. The most widely referenced of them is that due to Ntafos and Hakimi (1979). In their approach they use the representation of a program by its control flow graph and deduce the required set of paths by applying a minimum flow method. This method is a generalisation of Dilworth's theorem that deals with the generation of a minimum number of paths needed to cover all the vertices of a graph. Other methods of interest have tried to formulate the problem as a linear programming issue and suggested on this basis the derivation of such a path set by using an integer linear programming method, Wang et. al. (1989).

All the above mentioned methods are theoretically sound and have appeared in the literature as a panacea for the test data generation problem. However, their completeness is under further discussion in that all of the before mentioned algorithmic methods are mostly useful if the piece of software under test is examined statically and not dynamically. If the static approach is followed, then these approaches do generate a minimum path set which will cover all statements, branches, LCSAJs, etc., in the program. However, this approach is unrealistic since the validity of a path must be examined with test data. If the dynamic approach on the contrary is adopted, then any of these methods fails to meet the requirements of a path cover with a minimum number of paths. The reason is that when these paths are exercised with real sets of data, they have positive chances of being infeasible (a path is called infeasible if there are no data that can force its execution). In fact it has been shown by Yates and Malevris (1989) and Hedley and Hennell (1985) that there is a great number of infeasible paths in a program. Therefore, any number of paths, minimum or not, generated by a particular method, may contain paths that are infeasible. As a result of this situation, the achievable cover these path sets can provide will be far from the desirable 100% in most cases. Full coverage (100%) of the structural characteristics, as proclaimed by Hennell (1991), reflects the level of reliability achieved for the program under test. Woodward (1984) on the other hand suggests that the full coverage of certain program features increases one's confidence about the software.

Unfortunately, there appears hardly any mention in the literature on the notion of infeasible paths as if these are non existent. There exists no method that will avoid the generation of infeasible paths purely for the reason that a path in order to be validated for feasibility or not, it must be firstly generated and secondly exercised with test data. It is during the exercise with test data that one can decide upon the infeasibility of a path. In fact, as it has been proved by Weyuker (1979), the problem of determining the infeasibility of a path is undecidable. Consequently, there is not known technique that can generate feasible paths only.

Yates and Malevris (1989) proposed a path selection strategy for performing effective branch testing by attempting to reduce the incidence of infeasible paths upon the generation of the path set that will provide full branch coverage. An extension to this strategy has been reported by Malevris (1995) in providing a method for performing LCSAJ testing. Both strategies stem from the basic principle that the number of predicates along a path can determine the status of infeasibility of the path. The basic characteristics of the strategies developed are:

1. Generation of a set of paths P, each involving a minimum number of predicates.

2. Derivation of a data set that corresponds to the paths in P, execution of the program with it and determination of the coverage achieved.

While cover < required coverage (100% or other) repetition of step 3

3. Selection of an uncovered element of the program, and successive generation of the second, third, fourth etc. path through that element that contains the next least number of predicates correspondingly. Terminate the generation of paths when a feasible one has been found. Evaluation of the new value of the cover achieved.

2 GENERATION OF THE PATH SET P

In making use of the observation that the number of predicates determines the status of the feasibility of a path, the path set can be generated in turn as containing shortest paths through the program's statements, shortest paths through the program's branches and finally shortest paths through the program's LCSAJs.

This will fulfil the requirements set at step 1 of the strategies. The derivation of data sets at step 2 can be performed either by using a test data generator tool in the form of a symbolic evaluator or in the worst case manually.

Step 3 of the strategies can be performed, if of course the coverage achieved at step 2 is less than the required value. The generation of the paths at step 3 can be performed if the paths are generated as the *K*th shortest paths through the element (statement, branch or LCSAJ) that remains uncovered, for all the uncovered elements after performing step 2.
It must be explained here that the value of the required coverage at step 2 can be either 100% (usually in the cases of statement and branch testing) or a value < 100% if there are infeasible LCSAJs in the program. This can of course happen due to the nature of an LCSAJ, which may contain more than one predicate. If the predicates it contains happen to be infeasible then the housing LCSAJ can never be covered by test data. Hence, the coverage of LCSAJs, is only valid if values of coverage of < 100% are considered which of course exclude the possible infeasible LCSAJs.

3 DETERMINATION OF THE NUMBER OF PATHS REQUIRED FOR CONTROL FLOW TESTING

As already stated, the majority of the methods in the literature determine the number of paths in the path set as a minimum number of paths. Unfortunately, it is not known how many paths such a path set may contain as a function of the elements of the control flow graph, making difficult, if not impossible, the estimation of the cost involved for test data generation before the calculation of the actual path set. On the contrary, the number of paths generated at step 1 of the strategies can be calculated precisely as: $C = E-N+2$, where E is the number of arcs and N the number of nodes of the graph used in turn for each structural characteristic of the program (Decision to Decision graph, LCSAJ graph). The calculation of C can be found in Yates and Hennell (1985), who suggested a method for branch testing whose philosophy has been embodied here too in generating the paths at step 1 of the different strategies, modified each time to reflect the particularities of the structural characteristics.

The number of paths C calculated at step 1, is of course not the minimum possible one, as it has not been tried to be generated as such. The effort behind the path generation method is to minimise over the number of predicates each path contains (thus making the path more likely to be feasible) and not over the total number of paths the resulting path set will contain. In fact, although C may not minimum, it is not far from that. This argument is based on two independently carried out experimental results, by Yates and Hennell (1985) and by Bertolino and Marre (1994). The values calculated are shown in Table 1. E is the

As can be seen, the discrepancies between the values of the two columns C and M, are not very significant. the average difference between C and M for the first 26 programs is 3.15 paths/program (the 27th program is treated as an outlier). Of course, although these 3.15 paths/program are by first look in favour of minimum path set generation methods, a closer look into their nature, suggests that this may not be as straightforward as it seems. It must be taken into account that usually the paths contained in minimum path sets are longer hence they contain more predicates. Therefore, they have not only more chances of being

Table 1 Number of paths per program with the different approaches

Method	Program No	E	C	M
Yates & Hennell	1	4	3	2
"	2	8	5	2
"	3	11	6	5
"	4	7	4	1
"	5	10	6	3
"	6	9	5	1
"	7	14	8	2
"	8	12	7	2
"	9	9	5	1
"	10	8	5	2
"	11	5	3	1
"	12	9	5	1
"	13	12	6	2
"	14	9	5	2
"	15	13	7	1
"	16	12	7	5
"	17	9	5	1
"	18	8	5	5
Bertolino & Marre	19	3	2	1
"	20	4	2	2
"	21	16	11	10
"	22	8	4	3
"	23	19	9	4
"	24	10	5	2
"	25	19	10	2
"	26	25	11	6
"	27	100	49	8

infeasible, but also the cost involved for test data generation per path is much higher than the corresponding cost for paths that are generated as shortest ones. Hence, although the cost involved for test data generation per path for each approach is not known, it is the intuition of the author that the two overall values of cost per path and per method are rather similar. The overall advantage however is with the shortest paths generation approach.

It must also be noted that the minimum path set methods do suggest that the number of paths they contain is both lower and upper bound of the number of paths that are needed for test data generation in order to cover the structural characteristics of the program under test. However, it has already been stated that paths tend to be infeasible. If the path set that contains a minimum number of paths contains infeasible ones then there is no way that these infeasible paths can be replaced by other feasible ones. Therefore it is meaningless to adhere to bounds the values of which are firstly unknown beforehand, secondly not standard.

On the contrary, with the proposed methods, in case any of the paths generated at step 1 is infeasible, a replacement can be generated according to step 3 of the strategies. Therefore, they provide the basis upon which one can develop additional paths to increase the cover and possibly fully cover all program elements. The application of the above strategies to a set of 35 subprograms taken from the NAG Fortran Library, in order to assess their effectiveness in covering branches and LCSAJs, yielded a set of values that can be used to assess the effort required when infeasible paths are present. In fact, the number

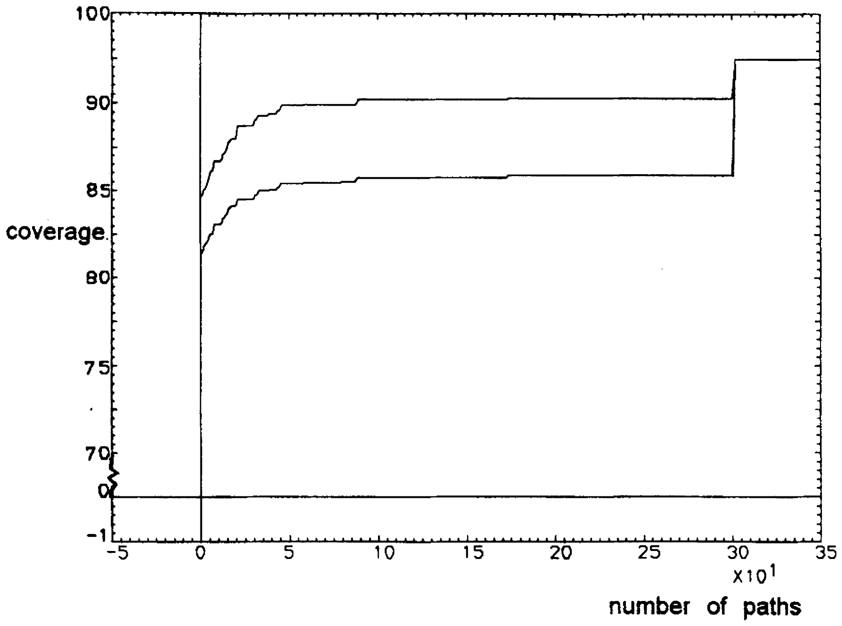

Figure 1 Coverages achieved versus number of paths

of paths required to cover each of the branches for all 35 subprograms was 1.39 paths/branch and 2.1 paths/LCSAJ. Both results were calculated considering a maximum number of 40 paths generated per uncovered element as it is not cost effective to generate more in order to gain higher coverage. This can be seen in Figure 1, where the coverages of branches (top line) and LCSAJs (bottom line) are calculated for the 35 subprograms. These two results are quite important as they allow the tester to estimate beforehand the cost involved in generating test data when attempting to cover program branches or LCSAJs.

4 CONCLUSIONS

The generation of test data in order to reveal errors is a very difficult process. Still, more difficult may be the inability to calculate the cost involved for testing certain program characteristics, in an attempt to attain a higher level of reliability from the software tester's point of view Hennell (1991). Attempts have been made, in the literature, to relate the generation of test data to that of generating program paths. Most methods that exist attempt to generate the required paths as a minimum number to reduce the cost of test data generation involved. By doing so, they fall into the trap of infeasible paths mostly because some of the paths they generate are long paths, hence more likely to contain conflicting predicates. Additionally, the infeasibility of paths is not levied resulting in the non coverage of the elements under test. Moreover, there is no mention on the effort needed to generate test data even if all paths included in the minimum path set were feasible. On the contrary the methods presented in the present paper, do go further beyond the achievements of all other methods in the literature in assessing the effort required for testing the most important characteristics of control flow testing (statements, branches, LCSAJs). They provide a lower bound $C = E-N+2$ of the number of paths needed for covering all the program elements under test, and an upper bound of 40 paths beyond which number it is not cost effective to generate paths as the increase in the coverage achieved is minimal. The coverages achieved with the maximum of 40 paths is nearly 90% for the branches and 85% for the LCSAJs. Although the upper bound proposed is on experimental grounds, it does provide the basis for estimating the effort required as on average 1.39 and 2.1 paths per branch and LCSAJ respectively need to be generated in order to achieve a fairly high coverage (in some cases 100%). These values are within the limits set by Ntafos (1988) who suggests that the number of test paths for branches and LCSAJs can be O(n) and O(nxn) respectively. Of course these values may not be representative of the whole of the population of the programs, they however indicate that the methods described here can be used to generate additional paths if some of them are found infeasible and this can be done with relatively little extra effort. Under these findings, lower and upper bounds of the effort required for achieving software reliability could also be calculated by using the relationship between coverage metrics and reliability. This together with the effort involved in the derivation of the appropriate path sets is the contribution of the present paper to the software engineering community mainly due to the very costly exersice of the testing phase of the software life cycle and the production of qualitative and reliable software.

5 REFERENCES

Bertolino, A. and Marre, M. (1994) Automatic generation of path covers based on the control flow analysis of computer programs, *IEEE Trans. Soft. Engin.*, **20**, 885-99.

Hedley, D. and Hennell, M.A.(1985) The causes and effects of infeasible paths in computer programs, in *Proceedings of the 8th International Conference on Software Engineering,* London, UK, 259-66.

Hennell M.A.(1991) Testing for the achievement of software reliability, *Reliab. Engin. and System Safety, Elsevier Scien. Pub. Ltd Eng.*, **32**, 119-34.

Malevris, N.(1995) A path generation method for testing LCSAJs that restrains infeasible paths, *Information and Software Technology,* **37**, 435-41.

Ntafos, S.C. (1988) A comparison of some structural testing strategies, *IEEE Trans. Soft. Eng.*, **14**, 868-74.

Ntafos, S.C. and Hakimi, S.L. (1979) On path cover problems in digraphs and applications to program testing, *IEEE Trans. Soft. Eng.*, **5**, 520-9.

Veevers, A. (1991) Some issues in software reliability assessment, *Journal of Soft. Testing Verif. and Reliab.*, **1**, 17-22.

Wang H.S., Hsu S.R. and Lin J.C. (1989) A generalized optimal path selection model for structural program testing, *J. Syst. and Software,* **10**, 55-63.

Weyuker, E.J. (1979) The applicability of program schema results to programs, *International Journal of Computing and Information Science,* **8**, 387-403.

Woodward M.R.(1984) An investigation into program paths and their representation, *Technique et Science Informatiques,* **3**, 273-9.

Yates, D.F. and Hennell, M.A. (1985) An approach to branch testing, *Proc. 11th Workshop on Graph Theoretic Techniques in Comp. Science,* Wurtzburg, West Germany, 421-33.

Yates, D.F. and Malevris, N.(1989) Reducing the effects of infeasible paths in branch testing, in *Proceedings of the 3rd Symposium on Software Testing, Analysis and Verification (TAV3),* Key West, Florida, USA, 48-56.

6 BIOGRAPHY

Nicos Malevris received a PhD in Computer Science from the University of Liverpool, an MSc in operational research from the University of Southampton and a BSc in mathematics from the University of Athens. He is currently an Assistant Professor in the Department of Informatics at the Athens University of Economics and Business. His interests include software engineering and in particular software testing, software reliability and software quality issues.