Information and Software Technology 1995 **37** (8) 435–441

A path generation method for testing LCSAJs that restrains infeasible paths

N Malevris

Department of Informatics, Athens University of Economics and Business, 76 Patission Street, Athens 104 34, Greece

The testing of all LCSAJs (Linear Code Sequences and Jumps) in a given code unit has been regarded as a very useful and important method for program testing that goes beyond testing program branches. It involves the generation of a set of paths and execution of these paths with sets of data. Thus it can be regarded as a path testing method too; see Woodward *et al.*[1] for example. Criteria for selecting such paths have, to date, received scant attention in the literature. In this paper, a suitable path selection strategy which aims at reducing the number of infeasible paths generated is proposed. The methods embodying this strategy are discussed and their application to a set of program units is reported and analysed. The ability of the methods in providing a high coverage for the LCSAJs in a program is also exposed.

Keywords: software testing, infeasible paths, program paths, LCSAJs

Software testing does require almost half the time devoted in developing a piece of software throughout its life cycle. It is therefore essential to establish a high level of confidence in the software produced before its delivery. This can be achieved by examining the piece of software with the aim of locating and removing all possible sources of errors. Such an exercise may be difficult and expensive, in that there is no single method that can guarantee to reveal all errors. Instead, a combination of testing methods can provide the basis of locating all flaws in a program.

Of the techniques usually adopted, the white box methods are the most widely used. These methods examine various characteristics and program elements with test data. The coverage of such elements provides the grounds for building the confidence in the software's veracity. Since generating all combinations of test data to test a program is a rather unrealistic approach due to the presence of possibly an infinite number of paths, instead, the coverage of specific elements in a program is attempted by generating sets of entrance-to-exit paths. The execution of the program with appropriate sets of data through the selected paths will provide a coverage for the program elements. The elements that are usually considered in software testing when adopting such a testing approach are statements and branches.

Ntafos[2] proves that the effort required to test all statements and branches in a program P by generating path sets to cover them is 0(n), n being the number of basic blocks in P. However, a number of errors may still be undetected after fully exercising all program branches and statements with test data. The types of errors that may go undetected are usually errors that depend on the special conditions of program structures such as loops that control the repetition of segments of code a number of times.

The strategy that fulfils this gap is the one proposed by Howden[3] and is known as boundary-interior path testing. With this strategy, paths that iterate loops a number of times are selected and tested, thus exercising the boundary conditions (entering the loop without causing it to be iterated) and the interior conditions (iterating the loop at least once). However, such a path testing strategy as proved by Ntafos[2] may require an exponential number of paths to be generated, thus making this method impractical to use.

An alternative to the boundary-interior path testing strategy is the strategy that requires all LCSAJs in a program to be tested successfully with test data.

An LCSAJ is defined as a linear sequence of executable code commencing either at the start of a program or at a point to which control flow may jump. It is terminated either by a specific control flow jump or by the end of the program. It may contain predicates which must be satisfied in order to execute the linear code sequence and terminating jump[4]. Following the definition of an LCSAJ, an LCSAJ graph can be constructed from the program LCSAJs and their connection in it.

LCSAJ testing

Testing all LCSAJs in a program entails exercising them with test data: the technique usually adopted is similar to the one for testing statements and branches. A set of paths is generated and the program is executed with test data that will be driven along the selected paths, forcing all LCSAJs in the program to be exercised. The effectiveness of the test data selected, therefore of the path set too, can be evaluated by the TER_3 (Test Effectiveness Ratio) measure, established by Woodward *et al.*[1]. This is the ratio of the number of LCSAJs exercised at least once over the total number of LCSAJs. TER_3 attains its maximum value of 1 when all program LCSAJs are covered.

This ratio is similar to the ratios for statements and branches, TER_1 and TER_2 respectively, suggested by Brown[5]. It has been proved[1], that $TER_3 = 1 \Rightarrow TER_2 = 1 \Rightarrow TER_1 = 1$. This means that testing all LCSAJs in a program with test data implies that all branches and statements are exercised too. Hence, LCSAJ testing is a stronger criterion than testing statements and branches in a program. Additionally, the generation of paths that cover either all statements or all branches or all LCSAJs, suggests that all three can be viewed as path testing strategies[1], with the first two regarded as the simplest path testing strategies and LCSAJ testing higher in this hierarchy. Ntafos[2] proves that the number of paths required to fully test all LCSAJs in a program can be $0(n^2)$. The fulfilment of $TER_3 = 1$ is in accordance with the boundary-interior method as Woodward *et al.*[1] argue. This is in disagreement with what Ntafos[2] suggests, who claims that boundary-interior path testing is incomparable to $TER_3 = 1$. We believe that the boundary-interior method does include the testing of all LCSAJs. The assertion made follows directly from the definition of an LCSAJ, which exercises the loop once or three times, thus fulfilling both the boundary and the interior requirements.

The polynomial number of paths required to test all LCSAJs in a program makes this testing technique a more realistic one compared to the boundary-interior method which requires an exponential number of paths in the worst case.

Hedley and Hennell[4] suggest that a great number of LCSAJs may be infeasible. An infeasible path (or subpath) is a path (or subpath) for which no data can be generated that can force its execution. This is due to the presence of conflicting predicates along the path (or subpath). In their study carried out on a number of 88 Fortran routines derived from the NAG Fortran Library, 619 of the 4790 LCSAJs generated were found to be infeasible, i.e. 12.5% of the LCSAJs they examined.

In a similar study by Woodward *et al.*[1], seven routines (six Algol 68 and one Fortran routine) again from the NAG Library, showed that a great deal of the paths generated were infeasible. In fact, in this study it was shown that the possibility of a path being infeasible depends on the number of LCSAJs contained by the path. Hence, paths that contain very few LCSAJs (up to five LCSAJs per path) are almost all feasible. This result is not directly connected with the previous finding that 12.5% of the LCSAJs are infeasible. Paths that contain few (up to five) LCSAJs, also contain a very small number of predicates. As has been shown by Yates and Malevris[6], paths that contain a small number of predicates have a very high probability of being feasible. However, the value of 12.5% of LCSAJs being infeasible is the mean of a sample of LCSAJs irrespective of the number of predicates they contain.

The reasons for the presence of infeasible LCSAJs, either when considered individually or as a combination with other LCSAJs in a program, are explained by Hedley and Hennell[4] in an attempt to justify their existence in a program. Consequently, when a set of paths that will cover all LCSAJs in a program needs to be generated, it has a high probability of containing infeasible paths. If infeasible paths are contained in the path set generated, then no full coverage of the LCSAJs can be achieved resulting in $TER_3 < 1$. The problem in such a case is resolved by generating additional feasible paths which will provide the sought full coverage for the LCSAJs.

The problem of generating the initial set of paths has been addressed by many authors. However, there appears no systematic way of generating the required paths except the method proposed by Ntafos and Hakimi[7]. Additionally, the problem of encountering infeasible paths in the path set generated is mentioned by surprisingly few authors, as Hedley and Hennell[4] report. Yet no one suggests any systematic approach for not including infeasible paths in the path set. In fact Gabow *et al.*[8] have proved that the avoidance of infeasible paths by allowing the user to specify his own statement sequences is an NP-complete problem.

The reasons for the existence of infeasible paths in a program can be concentrated in the presence of the conflicting predicates along each of these paths. Yates and Hennell[9] observed that if Π is a path through a code unit, and Π involves $q > 0$ predicates, then:

> for Π to be feasible, all q predicates must be consistent, whereas infeasibility of Π requires as few as two predicates to be inconsistent.

Yates and Malevris[6] showed that this observation can be generalized and used as a measure of path feasibility. In fact, they showed that if Π_1 and Π_2 are paths through a code unit, involving $r > 0$ and $s > r$ predicates respectively, then Π_1 is more likely to be feasible than Π_2. This can be applied to include LCSAJs too.

Lemma

If Π_1 and Π_2 are two LCSAJ paths with $r > 0$ and $s > r$ predicates respectively, then Π_1 is more likely to be feasible than Π_2.

Proof

Since an LCSAJ is a structure of code it follows that a path containing LCSAJs is simply a program path that can be obtained by replacing the LCSAJs by their corresponding code. Hence, as the claim by Yates and Malevris[6] has been derived for program paths the result follows straightforwardly for the LSCAJs too.

Path generation strategy

Following the above argument, a path generation method is required which will generate the paths to cover all LCSAJs in the program and will take into account the above Lemma. This leads to the conclusion that from the path sets that could validly be used to test a code unit's LCSAJs, the path set Π^* whose constituent paths each involves a minimum number of predicates is most likely to contain the least number of infeasible paths.

However, as proved by Weyuker[10] the problem of determining the feasibility of a program path is undecidable. Consequently, no guarantee can be given that any path set generated will contain only feasible paths. The question then posed is the following: If some/all of the paths in Π^* are found to be infeasible, and as a result full LCSAJ coverage cannot be achieved, which path(s) should next be generated in an attempt to increase the cover?

The answer to this question is given by Yates and Malevris[6] where a method for branch testing is presented. If branch rw, say, remains uncovered because the path through it is infeasible, then this method considers the path with the next smallest number of predicates that contains rw, and so on until a path that is feasible is found. This method of generating the paths in terms of the number of predicates they contain maintains the claim described earlier of the paths having good chances of being feasible ones.

The findings of the above method by Yates and Malevris can be applied here in the case of the LCSAJs with a number of modifications. These modifications are based on the principle that a branch is represented by a directed arc whereas an LCSAJ by a node in the control flow graph and LCSAJ graph respectively.

This difference can be easily remedied by replacing each LCSAJ by a couple of nodes connected by a dummy arc with zero weight associated with it. Such an approach would have increased the number of elements in the LCSAJ graph, affecting the performance of the algorithm developed for generating the paths in Π^*. A more sophisticated approach was taken towards resolving this matter and can be found in Malevris[11].

The path generation strategy for testing the LCSAJs in a code unit c can now be specified by the following steps:

(1) Generate a path set Π^* for the code unit c.
(2) Derive the value of TER_3 for c in respect of Π^*.

While $TER_3 < 1$, repeat step (3).

(3) Select an uncovered LCSAJ w of c, generate Π_r (w), r = 1,2,..,N, and recalculate TER_3.

Where Π_r (w) is the path through w which involves the *rth* smallest number of predicates.

There are two questions that arise in connection with step (3) of this strategy. First, if there are several uncovered LCSAJs, in what order should they be selected? The answer to this question is given in the section where the results are presented. Second, what method(s) are employed to implement the above strategy? The methods used to implement steps (1) and (3) of the above strategy, hereafter referred to as LSPM (LCSAJ Shortest Path Method) and ELSPM (Extended LSPM), can be found in Malevris[11].

Selection mechanisms

In order to assess the effectiveness of the proposed strategy, LSPM and ELSPM were coded in Fortran 77, interfaced, and the resulting implementation applied to the LCSAJ graphs of a set of N = 35 subroutines taken from the NAG Fortran Library. The 22 of the 35 subroutines are the same subroutines used in Yates and Malevris[6] where a method for branch testing was presented. The reason why the same subroutines are used here, too, is for comparing the effectiveness of the method for testing LCSAJs against that for testing branches. Yates and Malevris[6] showed that in order to cover all branches in a subroutine the average number of paths required for the 22 subroutines is $n^* = 20.59$, hence requiring only a mean of 2.52 paths per arc or 1.39 paths per branch. These results were obtained with the application of the methods developed for branch testing. These values were derived after considering all the orderings of the arcs left uncovered when $TER_2 < 1$ for each of the 22 subroutines.

In order to remove any sign of bias in the way the remaining uncovered branches were selected to be covered, all the orderings of the arcs need to be calculated. However, this may be prohibitive as the number of arcs that are left uncovered may be considerable. Additionally, the situation is worsened when considering the number of LCSAJs that need to be tested. Usually, an LCSAJ graph contains more nodes than its corresponding control flow graph; consequently there are more LCSAJs in a program than there are branches. Hence, considering all the orderings of the LCSAJs that are left uncovered is prohibitive. In order to proceed in evaluating the effectiveness of the method for LCSAJ testing it is, therefore, necessary to consider a selection mechanism that will establish the order in which the uncovered LCSAJs are selected in turn, for path generation so as to achieve their coverage. For these reasons a selection mechanism is compulsory.

The matter of investigating what the best mechanism is for establishing the order in which the uncovered components (arcs, LCSAJs, etc...) will be considered is a subject of ongoing research. The mechanisms adopted in this paper are presented and statistically justified below:

(a) Arc selection mechanism

The selection mechanism adopted here is based on the observation that the greater the number of paths through an arc in the graph, the greater are the chances that one of them will be feasible. According to this observation, the arc to be selected first will be the arc with the greatest number of paths through it. The number of paths through an arc, say wz, is calculated as follows: If P_{wz} is the number of paths through arc wz, then $P_{wz} = f \cdot d_i(w) \cdot d_o(z)$, where $d_i(w)$ is the number of incoming arcs to node w, $d_o(z)$ the number of arcs outgoing from node z and f is the number of paths entering w for each incoming arc of w (assumed equal for each incoming arc of w).

(b) LCSAJ selection mechanism

The strategy chosen here incorporates the basic findings of Yates and Malevris[6]. Each LCSAJ involves a certain number of predicates and this number, as the authors statistically proved, can determine its feasibility. Thus, the LCSAJ with the smallest number of predicates among the LCSAJs that are left uncovered is chosen to be considered for path generation first. However, if there are ties then these are broken by using the selection mechanism (a) for the arcs with the modification that instead of an arc a node is to be considered.

Statistical evaluation of the selection mechanisms

In this section the efficacy of the selection mechanisms described above is validated. In Yates and Malevris[6] a number of 22 subroutines were tested for covering their branches with a specific method. For 13 of these subroutines, some arcs still remained uncovered. To continue with further covering them, Yates and Malevris considered all orderings of these arcs and calculated the average number of paths needed for all cases. To assess the effectiveness of the selection mechanism (a), this was applied to the same 13 subroutines and then a statistical test was performed. The statistical test involved a hypothesis testing. In this testing the mean μ_0 (the average number of paths needed to be generated to cover the remaining uncovered branches per subroutine) of the sample of the 13 subroutines where all orderings were considered, was tested against the mean μ_1 of the same sample of subroutines having applied the selection mechanism (a). The testing of hypothesis was performed and the result showed that the two means μ_0 and μ_1 are the same with 95% confidence.

Similarly, for the LCSAJs an identical statistical test was performed incorporating the selection mechanism (b). Here all the orderings for the LCSAJs were derived for nine of the subroutines, as calculating the orderings for the rest of the subroutines was prohibitive and the hypothesis testing also proved that μ_0 and μ_1 are the same with 95% confidence. Consequently, it has been statistically proved that the selection mechanisms in both cases give results very close to the average cases obtained when considering all orderings of the arcs or LCSAJs. It was therefore decided to apply the selection mechanism (b) to all the remaining uncovered LCSAJs of the 35 subroutines.

Results

LCSAJ coverage achieved with LSPM

The 35 subroutines taken from the NAG Fortran Library to which LSPM was applied are presented in Table 1, together with the number of LCSAJs each one possesses and the number of branches in the corresponding control flow graph. Also given in Table 1 for each subroutine are: its

Table 1. Coverages achieved with the LSPM method

Subroutine	No of LCSAJs	No of LCSAJs not covered	No of infeasible LCSAJs	TER_3 for LSPM	Relative TER_3	TER_2 for LSPM
A02ABF	8	0	0	1.00	1.00	1.00
C02ADZ	12	3	0	0.75	0.75	0.83
C02AEZ	5	0	0	1.00	1.00	1.00
C06AAZ	10	2	2	0.80	1.00	1.00
C06ABZ	36	30	1	0.17	0.17	0.24
C06DBF	14	2	0	0.86	0.86	0.81
C06EBT	19	13	1	0.32	0.33	0.38
C06GCF	9	1	1	0.89	1.00	1.00
D02XHF	16	0	0	1.00	1.00	1.00
E01AAF	12	7	1	0.42	0.45	0.73
E01ABF	25	15	0	0.40	0.40	0.52
E02AFF	35	27	3	0.23	0.25	0.26
E02AKZ	23	3	3	0.87	1.00	1.00
E02BBF	19	4	2	0.79	0.88	0.91
E02GBN	26	14	2	0.46	0.50	0.65
E02RBF	24	4	3	0.83	0.95	0.90
F01AFF	3	0	0	1.00	1.00	1.00
F01AHF	19	11	2	0.42	0.47	0.50
F01AZF	17	9	2	0.47	0.53	0.52
F01BEF	12	6	1	0.50	0.55	0.67
F01CLF	13	6	5	0.54	0.88	0.83
F01CMF	7	0	0	1.00	1.00	1.00
F01CSF	18	11	4	0.39	0.50	0.63
F03AGF	32	10	2	0.69	0.73	0.83
F03AMF	16	0	0	1.00	1.00	1.00
F04AGY	9	2	1	0.78	0.88	1.00
F04AQF	21	18	5	0.14	0.19	0.19
F04MAY	19	19	2	0.00	0.00	0.00
F04MCV	16	0	0	1.00	1.00	1.00
G01BCF	46	17	5	0.63	0.71	0.88
G04ADF	18	18	5	0.00	0.00	0.00
G05EAF	41	34	7	0.17	0.21	0.22
S15AEF	18	16	2	0.11	0.13	0.16
S17ACF	14	0	0	1.00	1.00	1.00
S18CCF	20	0	0	1.00	1.00	1.00

number of LCSAJs, the number of LCSAJs not covered, the number of infeasible LCSAJs, the value of TER_3, the relative value of TER_3 and also the value of TER_2 achieved.

The application of LSPM to the 35 subroutines realized a mean value of 0.62 for TER_3. Full TER_3 coverage was achieved for only nine of the 35 subroutines, leaving a total of 302 LCSAJs uncovered. Of these LCSAJs, 62 are infeasible LCSAJs, which is 9.51% of the total number of 652 LCSAJs contained in the 35 subroutines. This result conforms with the claim by Hedley and Hennell, where approximately 12.5% of the LCSAJs tested were found to be infeasible. Consequently, if the maximum achievable TER_3 coverage is considered (by excluding all infeasible LCSAJs for all 35 subroutines), its mean value is calculated as 0.66 by fully covering an additional three subroutines of the 35, whereas for the rest of the subroutines the relative TER_3 (ratio of LCSAJs covered over total number of feasible LCSAJs) varied between 0.0 and 0.95.

LCSAJ coverage achieved with ELSPM

ELSPM was then applied to the remaining 240 coverable LCSAJs by using the selection mechanism (b) described in the section where the selection mechanisms were presented. The application of ELSPM to a selected LCSAJ w, involved the generation of the *2nd*, *3rd*, ..., k*th* shortest path through w where $k = \min(r, 300)$ and the shortest feasible path through w is the r*th* shortest path through that LCSAJ. The results obtained are presented in the form of the graph of Figure 1.

The graph is a plot of C_k against k for k in the range of (1, 300) where:

$$C_k = \frac{\sum_{m=1}^{n} TER_3(m, k)}{q \cdot \sum_{m=1}^{q} A_m}$$

q is the number of subroutines considered, q = 1, ..., 35
TER_3 (m,k) is the cover of subroutine m achieved in respect of the k total paths covered and is defined as:

$$TER_3(m, k) = \begin{cases} TER_3(m, k_i) & k_i \le k < k_i + k_{i+1} \\ TER_3(m, k_{i+1}) & k = k_i + k_{i+1} \end{cases}$$

where k_i is the k_i*th* shortest feasible path through LCSAJ i.

$$\sum_{m=1}^{q} A_m$$

is the mean achievable coverage for the 35 subroutines. Hence, C_k shows the mean relative coverage for each value of k.

As can be seen from the graph, the increase in C_k is considerable for smaller values of k. C_k attains a value of 0.85 at k = 40. For k in excess of this value, the rate of increase of C_k begins to slow, and to slow markedly after k = 100. The values of C_k achieved as can be seen by the graph for k = 10, 20, 40, 200 and 300 are 0.76, 0.81, 0.85,

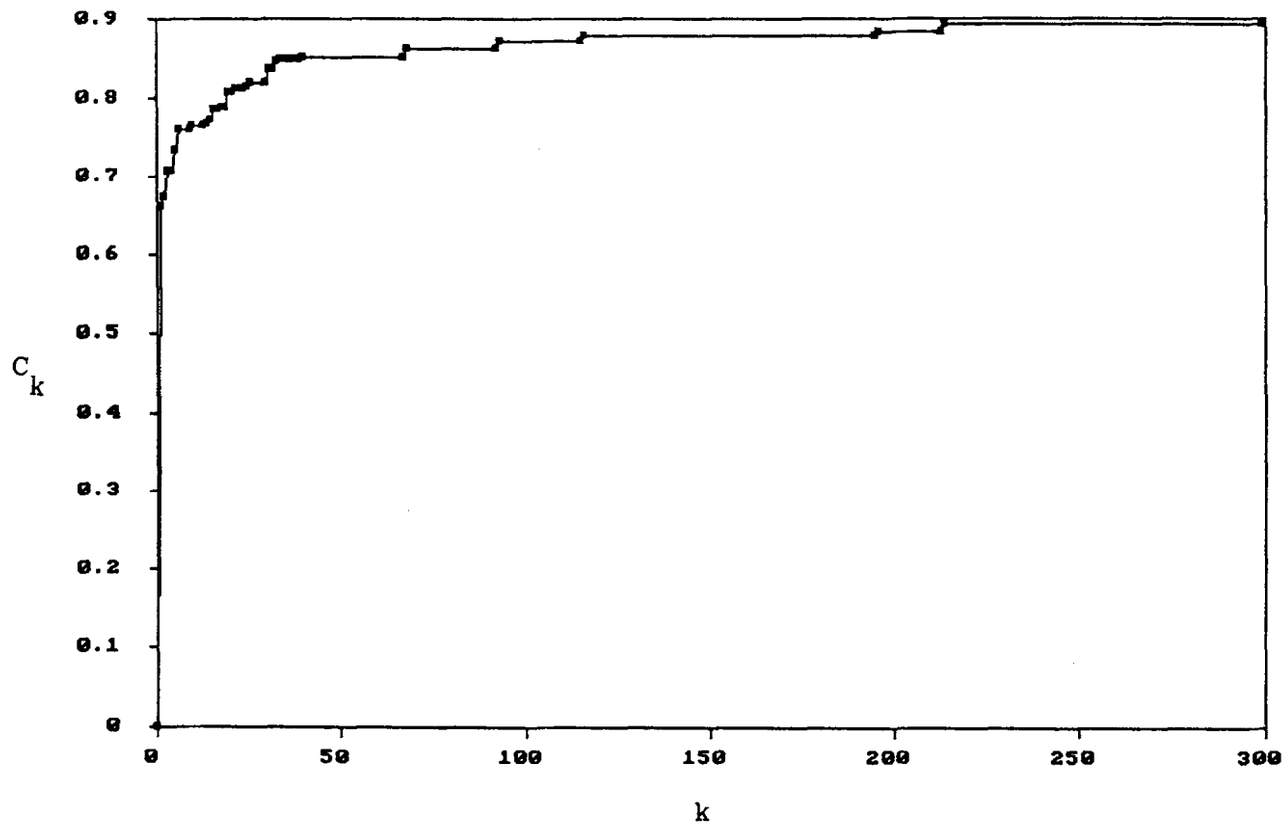

Figure 1 Plot of each mean achieved coverage against number of paths

0.88 and 0.89 respectively. The results show the effectiveness of ELSPM in providing a high coverage of the LCSAJs. It can also be concluded that the ideal number of paths that need to be generated and tested per subroutine is 40 paths, as it is not cost-effective to generate and test paths in excess of this value in an attempt to increase the LCSAJ cover for each of the 35 subroutines.

In order to estimate the ability of LSPM and ELSPM in covering an LCSAJ, the mean number of paths generated per LCSAJ in achieving a maximum cover, V, needs to be calculated. In calculating V, the suggestion of generating no more than 40 paths in order to achieve a maximum cover was adopted. Hence, V can be calculated as:

$$V = \frac{\sum_{m=1}^{35} T_m}{\sum_{m=1}^{35} |U_m|}$$

where

$$T_m = \sum_{j \in U_m} \min(40, k_m(j))$$

and U_m is the number of LCSAJs of subroutine m that are left uncovered after the application of LSPM, $k_m(j)$ is the number of paths through LCSAJ $j \in U_m$ that are generated by both LSPM and ELSPM in order to cover j when ELSPM is applied to the LCSAJs of U_m by using the selection mechanism described earlier.

According to the assumptions discussed above, the calculation of V renders a value of V = 3.71. V can now be used to provide an estimate of the total number of paths l^* that both LSPM and ELSPM will need to generate to achieve a maximum value of TER_3 for a given code unit. If l_1 is the mean number of LCSAJs per subroutine covered by LSPM using only one path and l_2 is the mean number of LCSAJs per subroutine to which ELSPM was applied, then, $l_2 = 240/35 = 6.86$. Since the mean number of LCSAJs per subroutine is 16.86, then, l_1 can be calculated as $l_1 = 10.00$. Using l_1 and l_2, l^* can be derived as $l^* = l_1 + Vl_2$, hence, $l^* = 35.45$, therefore requiring a mean of 2.1 paths per LCSAJ.

Branch coverage achieved

Covering all LCSAJs in a program also provides a full coverage for the branches of the program, hence $TER_3 = 1 \Rightarrow TER_2 = 1$. The ability of both LSPM and ELSPM in providing a coverage for the branches too is evaluated here. As can be seen in Table 1, the path sets generated by LSPM to cover all LCSAJs in the 35 subroutines provided a mean maximum achievable value for $TER_3 = 0.66$. These path sets also provided a coverage for some of the branches and the mean value for TER_2 was calculated as $TER_2 = 0.71$. It is evident that full branch coverage could not be achieved due to the presence of infeasible paths in the path sets generated by LSPM. Therefore, ELSPM was applied to the remaining uncovered LCSAJs and the corresponding values of TER_2 were calculated each time there was an increase in the mean achievable coverage for the LCSAJs. Hence, the values calculated for TER_2 for k = 10, 20, 40, 200 and 300 were 0.81, 0.86, 0.90, 0.93, 0.94 respectively. It can therefore be argued that both LSPM and ELSPM do provide a substantial coverage for the branches upon achieving a maximum cover for the LCSAJs. The application of LSPM to the 35 subroutines discovered 'bugs' for the subroutines F01CLF, E02RBF, E02GBN, G05EAF and consequently the values for TER_2, calculated above represent the maximum achievable coverages excluding the infeasible branches that cannot be covered. This can be regarded as the ability of the LSPM method in discovering 'bugs'.

The results obtained here after having applied LSPM and ELSPM to a number of 35 Fortran subroutines indicate that the presence of infeasible paths is a factor that cannot be ignored. However, their presence is usually ignored by other researchers and the results provided in the literature are based on their absence rather than on their presence. In fact, there is almost no chance that infeasible paths will not be encountered when generating a path set, in order to provide a cover for branches or LCSAJs in a program.

Woodward[12] performed a profound analysis of the NAG Fortran Library. He analysed 116 subroutines and calculated the minimum path sets to cover all basic blocks, branches, LCSAJs and other basic program elements. He observed that, on average, the minimum path sets to cover all branches and LCSAJs in a program contain no more than 4.1 and 9.7 paths respectively. The medians of the samples considered give 3.5 and 7 paths respectively. He also calculated the number of extra paths needed to cover the LCSAJs once all branches have been covered, thus having achieved $TER_2 = 1$. The average number of extra paths needed for the 116 subroutines was 5.6 and the median of the sample three paths. In fact, it is concluded by Woodward[12] that 16% of the routines require no extra paths, 51% require fewer than five extra paths and 16% require more than 10 extra paths. However, and as it is argued by Woodward in his paper, the results he derived are of theoretical rather than of practical importance, the reason being in the presence of infeasible paths which Woodward as well as most of the other researchers exclude from their results claiming that their methods and findings are purely static and not dynamic.

In order to assess the effectiveness of LSPM in providing a small number of extra paths to achieve $TER_3 = 1$ a similar examination was performed with the 35 subroutines considered in this paper. For 8.6% of the subroutines no extra paths were needed, 17.14% of the subroutines required one extra path, 68.6% of the subroutines required fewer than five extra paths, and 5.7% needed 10 or more extra paths. These results are along the same lines as the results presented by Woodward since the average number of extra paths needed per subroutine is three paths which agrees with the median value calculated by Woodward[12]. However, as already stated, this type of result is of minimal importance due to the presence of infeasible paths in the path sets generated. If the presence of the infeasible paths is considered, then full coverage of the LCSAJs cannot be achieved, resulting in $TER_3 < 1$. As already shown in this paper, the employment of ELSPM can increase the coverage and the number of paths needed in total to achieve

a maximum cover is 35.45 paths per subroutine, therefore requiring an extra 25.45 paths per subroutine (the average number of paths generated by LSPM per subroutine is 10 paths; this agrees with the mean value calculated by Woodward). The discrepancy of 25.45 paths between the static and dynamic values calculated above, is due to the presence of infeasible paths. This matter is consequently too important to be ignored by path generation methods in the literature.

Conclusions

In this paper the issue of path testing has been addressed. In particular, the problem encountered when attempting to generate all program paths has been discussed. The need to generate a set of paths that will cover certain program elements in an attempt to approach the all paths criterion has also been considered.

Of these program elements, the LCSAJs do provide the basis for building the programmer's confidence, if a set of paths can be generated that will, upon successful execution of the program with appropriate sets of data, cover all program's LCSAJs. However, LCSAJs can contain infeasible parts of code. Moreover, the paths that may be generated are likely to contain infeasible LCSAJs or combinations of LCSAJs that might cause infeasibility, thus not fulfilling the covering of all LCSAJs criterion. These approaches are of static importance rather than of dynamic. To date, the problem of infeasible paths has received scant attention and the path generation methods in the literature do not consider their presence in the path sets they generate, although they accept their existence. In this paper, a path generation strategy which aims at minimizing the number of infeasible paths generated while testing LCSAJs has been proposed. The strategy is founded on the assertion that the fewer the predicates a program path contains, the more likely it is that the path is feasible.

To investigate the efficacy of the proposed strategy, LSPM and ELSPM, the path generation methods embodying the strategy were applied to a set of 35 Fortran subroutines. The effectiveness of the strategy proposed is exhibited by the LCSAJ cover obtained. LSPM achieved full coverage for nine subroutines, maximum achievable for 12 subroutines (having excluded the infeasible LCSAJs the subroutines contain) and a mean cover of 0.62 and 0.66 respectively. ELSPM increased this to a maximum achievable cover of 0.85 (excluding the infeasible LCSAJs), and with generating no more than 40 shortest paths through an LCSAJ, as the contrary would not prove to be cost effective. This maximum cover was achieved by generating no more than 2.1 paths per LCSAJ and 35.45 paths per subroutine. The ability of LSPM and ELSPM in providing a high cover for the branches has also been exposed. In fact, the application of LSPM realized a value for $TER_2 = 0.90$ with generating a maximum of 40 paths. Finally, it was shown that LSPM and ELSPM can provide real coverages for the LCSAJs in a program. To substantiate this claim, LSPM was compared with a method proposed in the literature by Woodward[12] and it was found that if a set of paths that covers all branches is available, then, on average, an extra three paths are theoretically needed per subroutine in order to cover all of its LCSAJs too. Both LSPM and the method by Woodward lead to the same result. However, and as has been shown in this paper, the infeasibility of paths makes this result static and of little practical value. If feasibility of the paths is at stake, then the application of ELSPM will increase the cover, requiring an additional 25.45 paths per subroutine. On the contrary, no other method in the literature provides a methodical way of circumventing the problem of encountering infeasible paths in the path sets it generates. This must be regarded as the strength of the strategy discussed in this paper, as it tends to minimize the number of paths (feasible and infeasible) generated during LCSAJ testing by also reducing the associated time and cost overheads.

The above make the methods easily applicable to the software testing industry where the testing of a piece of software is regarded as a very important task. Their significance lies in their ability to achieve a very high coverage of LCSAJs at very little extra cost in a methodical way. This allows the methods to be incorporated either to existing tools or to new ones with little modifications.

Acknowledgements

The author would like to thank NAG Ltd for permission to access the source code of their Fortran Library, and the anonymous referees for their useful comments and suggestions.

References

1 Woodward, M R, Hedley, D and Hennell, M A 'Experience with path analysis and testing of programs' *IEEE Trans. Soft. Eng.* Vol 6 No 3 (1980) pp 278–286
2 Ntafos, S C 'A comparison of some structural testing strategies' *IEEE Trans. Soft. Eng.* Vol 14 No 6 (1988) pp 868–874
3 Howden, W E 'Methodology for the generation of program test data' *IEEE Trans. Comput.* Vol 24 No 5 (1975) pp 554–559
4 Hedley, D and Hennell, M A 'The causes and effects of infeasible paths in computer programs' *Proc. 8th Int. Conf. on Soft. Eng.*, London, UK (1985) pp 259–266
5 Brown, J R 'Practical applications of automated software tools' *TRW Report TRW-SS-72-05* TRW Systems, Redondo Beach, CA, USA (1972)
6 Yates, D F and Malevris, N 'Reducing the effects of infeasible paths in branch testing' *Proc. ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)* Key West, FL, USA (1989) pp 48–54
7 Ntafos, S C and Hakimi, S L 'On path cover problems in digraphs and applications to program testing' *IEEE Trans. Soft. Eng.* Vol 5 No 5 (1979) pp 520–529
8 Gabow, H N, Maheshwari, S N and Osterweil, L J 'On two problems in the generation of program test paths' *IEEE Trans. Soft. Eng.* Vol 2 No 3 (1976) pp 227–231
9 Yates, D F and Hennell, M A 'An approach to branch testing' *Proc. 11th Int. Workshop on Graph Theoretic Techniques in Computer Science*, Wurtzburg, Germany (1985) pp 421–433
10 Weyuker, E J 'The applicability of program schema results to programs' *Int. J. Comput. Inf. Sci.* Vol 8 (1979) pp 70–96
11 Malevris, N *An effective approach for testing program branches and linear code sequences and jumps* PhD Thesis, University of Liverpool, UK (1988)
12 Woodward, M R 'An investigation into program paths and their representation' *Technique et Science Informatiques* Vol 3 No 4 (1984) pp 273–279