Information and Software Technology 54 (2012) 915–932

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Mutation based test case generation via a path selection strategy

Mike Papadakis [*,1], Nicos Malevris

Department of Informatics, Athens University of Economics and Business, Athens, Greece

ARTICLE INFO

Article history:
Received 26 September 2011
Received in revised form 13 February 2012
Accepted 26 February 2012
Available online 5 March 2012

Keywords:
Software testing
Unit testing
Mutation testing
Path testing
Adequacy criterion
Infeasible paths

ABSTRACT

Context: Generally, mutation analysis has been identified as a powerful testing method. Researchers have shown that its use as a testing criterion exercises quite thoroughly the system under test while it achieves to reveal more faults than standard structural testing criteria. Despite its potential, mutation fails to be adopted in a widespread practical use and its popularity falls significantly short when compared with other structural methods. This can be attributed to the lack of thorough studies dealing with the practical problems introduced by mutation and the assessment of the effort needed when applying it. Such an incident, masks the real cost involved preventing the development of easy and effective to use strategies to circumvent this problem.
Objective: In this paper, a path selection strategy for selecting test cases able to effectively kill mutants when performing weak mutation testing is presented and analysed.
Method: The testing effort is highly correlated with the number of attempts the tester makes in order to generate adequate test cases. Therefore, a significant influence on the efficiency associated with a test case generation strategy greatly depends on the number of candidate paths selected in order to achieve a predefined coverage goal. The effort can thus be related to the number of infeasible paths encountered during the test case generation process.
Results: An experiment, investigating well over 55 million of program paths is conducted based on a strategy that alleviates the effects of infeasible paths. Strategy details, along with a prototype implementation are reported and analysed through the experimental results obtained by its employment to a set of program units.
Conclusion: The results obtained suggest that the strategy used can play an important role in making the mutation testing method more appealing and practical.

1. Introduction

Software testing establishes the main method for identifying errors while suggesting an acceptable level of confidence for the software under investigation. Assessing the quality of a given data set a coverage measure must be utilised. Such a measure, tries to determine the extent to which the occurrences of some software features have been successfully exercised. Different program features lead to different criteria and hence different testing requirements. Criteria usually require the execution with actual data, to exercise their characteristics such as statements, branches, decisions, paths etc. Thus, the use of coverage criteria can be also seen as a guide for selecting appropriate test cases to increase the level of the testing thoroughness.

Mutation testing is a fault-based testing technique originally introduced by Hamlet [1] and DeMillo et al. [2]. Researchers have provided evidence that mutation testing forms a rather powerful testing criterion able to effectively detect more faults than most of its structural testing rivals [3,4]. Mutation testing induces syntactical alterations of the code under test with the aim of producing semantically different versions of the considered code. Each such mutated program version contains one simple syntactic change from the original source code. To assess the quality of the test cases, mutation requires the execution of the altered program versions with the goal of distinguishing them from the original one. A mutant is said to be killed if there exists such a test whereas, it is said to be equivalent if there is not. Assessing the testing quality is usually measured by a ratio of the killed over the totally introduced mutants. This constitutes a criterion usually referred to as the mutation testing criterion [5]. Generating test cases successfully, with respect to a predefined-targeted killing ratio of the introduced mutants forms the focus of this paper.

The generation of test cases when performing mutation can be a very costly incident. To generate mutation adequate test cases, a tester must design test cases able to infect, execute and reveal all the introduced mutants. The term infect is used to express the case where mutant execution caused a discrepancy in the program

* Corresponding author.
E-mail addresses: michail.papadakis@uni.lu (M. Papadakis), ngm@aueb.gr (N. Malevris).
[1] New affiliation: Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, L-1359 Luxembourg, Luxembourg.

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2012.02.004

state, at the altered program point by the introduced mutant. This can be achieved by iteratively producing and executing test cases with both the original and the mutated program versions, in order to determine the mutants killed. The determination is done by comparing the outcomes of the original and the mutated program versions. This process can be very expensive if performed in a non systematic way. Researchers usually believe that mutation expenses are considerably higher than what the other white box criteria require. This belief relies on the huge number of testing requirements that it introduces and on the observation that mutation usually needs more test cases than the other structural testing criteria [3]. To date, the effort involved when applying mutation has received scant attention in the literature. This lack of effort quantification in relation to the absence of practical solutions addressing the peculiarities of the mutation testing process have resulted in its reduced usage compared to the other structural criteria such as branches, data flow, etc.

In the present paper, a path based approach for generating mutation based test cases is proposed and empirically evaluated. The experimental evaluation is performed with the aim of assessing the effort entailed by mutation testing during the test generation process. There are various ways in producing test cases. The one employed in the present study is based on selecting paths that the mutants lie on, and subsequently executing them with test data, thus utilising each one of the mutants in order to kill them. The drawbacks of such a strategy, as it has been reported in the literature, are attributed to the presence of infeasible paths [6,7]. The strategy adopted in this exercise, seeks to reduce the undesirable effects of infeasible paths during the testing process and hence reciprocally, also reduce the overall testing effort. Therefore, it can be argued that there is a direct relationship between the number of generated test paths and the effort involved for performing mutation. In fact, and without any loss of generality, the experiment assesses the relative cost-effective application of weak mutation [8,5] testing based on a proposed path selection method able to eliminate the undesirable effects of infeasible paths.

The paper is organised as follows: In Section 2 the relevant to the conducted experiment notation and terminology is given along with some related to the area work. In Section 3 a path selection strategy which seeks to reduce the incidence of infeasible paths is proposed. In order to evaluate the strategy and to interpret its achievements, certain features of the prototype implementation are analysed and presented in Section 4. Additionally, in Section 5 the results of the application of the proposed strategy to derive mutation test cases to a set of program units, are presented. The important findings of the experimental study along with the nature of infeasible paths, together with certain characteristics of the proposed strategy, can justify pertinent comments about the effort associated with weak mutation testing in general. These are presented in Section 6. Finally, the conclusions are discussed in Section 7.

2. Background and related work

This section introduces the relevant notation, terminology and a brief description of the techniques used in the conducted experiment. Selected relevant related work on generating mutation adequate test cases is also described.

2.1. Notation and terminology

The control flow of a unit under test is usually modelled with a graph known as the *Control Flow Graph* (CFG) denoted by $G_c(N, E)$, where C is a program unit whose connected directed graph is composed of a set of nodes N and a set of edges E. Each node n represents a basic block and each edge e represents a possible transfer of flow between two basic blocks say n_i and n_j with $i \neq j$. A *basic block* (*node n*) is a simple sequence of one or more successive, executable statements, such that the sequence has only one entry and only one exit point and once execution of the entry point statement takes place, it will always cause a consecutive execution of all statements up to the exit one in a row. Based on the CFG a *path* is a finite sequence of edge-connected nodes $\langle n_1, n_2, \ldots, n_k \rangle$ such that for every j in $1 \leqslant j \leqslant k$ the nodes n_j, n_{j+1} form an edge of the set E of G_c. An S-to-F path is usually referred to as a *complete path*, S and F being the entry and exit nodes of the graph respectively. A path may not always be a complete one and this is usually referred to as a *subpath*.

Modelling a unit's flow corresponds to one-to-one mapping between the "actual" program paths of code unit C and respective complete paths of G_c. Thus, actually executing one of C's paths is equivalent to traversing, "covering" its corresponding path in G_c. Further, since the actual execution of a program path implies execution of its code elements covering a complete path (based on the model G_c) implies covering its constituent modelled elements.

It must be noted here that the peculiarities of the different programming languages must be reflected in the CFG. In particular, there are issues that need specific and more detailed analysis that can affect the size of N and E thus also affecting any test data generation that attempts to use them as a basis. In particular, the problems that can arise deal with the compound decisions (containing short-circuit operators). In modern programming languages compound decisions must be converted to a series of simple ones, thus enforcing the introduction of additional nodes and edges for their representation. Details of the effects of such incidents can be found in [9].

2.2. Testing based on selected paths

Generally, path based approaches work by selecting an appropriate set of paths and trying to derive test cases for these selected paths. Generating test cases according to a selected path is traditionally performed based on symbolic execution [10,11]. Recently, search based testing approaches [12,13] have also been proposed in order to tackle this problem effectively. In either case (using symbolic execution or search-based testing) the testing process works by iteratively selecting and producing tests for paths that cover the sought test elements. This activity entails the following steps:

1. Select a set P of paths through G_c which covers the target code elements.
2. Generate a set of data D that will drive the execution of C according to the selected set P.
3. Execute the unit C with the set of data D.

However, not all paths of G_c, selected in step 1, may represent valid computations with respect to the code unit C. These invalid paths are called *infeasible paths*, as there exist no data sets that will force their execution (see Section 2.4.2 for details). The existence and influence of infeasible paths in practice, plays a major role as it compels performing steps 1 and 2 iteratively in order to construct a set of data D that covers the target code elements. This difficulty can be extremely problematic [7,14] and results in unbearable overheads as the conducted experiment shows, see Section 6.3 for a discussion of this issue.

Also, if mutation is to be considered, the control flow graph of C is rather inadequate to base the sought test path generation activity. Such being the case, a more appropriate model is needed as this is discussed in [15], where an enhanced CFG is suggested and is presented in Section 3.2.

2.3. Mutation testing criterion

Mutation testing (as initially introduced), is a fault-based technique for the quality assessment of the various test cases as proposed by DeMillo et al. [2]. In practice, it is employed by creating a slightly different to the original program set of programs. The minor alterations to the original program are called mutants. The mutants are introduced by modifying the original program's source code based on simple syntactic rules called mutation operators.

Due to its notion, a mutant is nothing else but a deliberately inserted mistake.

The tester's goal when employing mutation is to create tests able to reveal these deliberately introduced mutants. In order to assess a test set, the tester should execute it both with the original and the mutated programs. In case the outputs of the original and a mutated program differ, the particular mutant is considered as covered (exercised) and referred to as killed. In the opposite case the mutant is considered as uncovered and referred to as alive. A live mutant indicates one of the following two situations: either a) the mutant is equivalent, i.e. the mutated program is functionally equivalent to the original and thus, revealing it test cases do not exist, or b) the test set is incapable of unrevealing and killing it.

The test set D adequacy of a program C is measured by the mutation score ($MS(C, D)$) and the relative mutation score ($MS^*(C, D)$) computed as follows:

$$MS(C,D) = \frac{\#\ \text{Dead Mutants}}{\#\ \text{Total Mutants}} \quad \text{and}$$

$$MS^*(C,D) = \frac{\#\text{Dead Mutants}}{\#\ \text{Total Mutants} - \#\ \text{Equivalent Mutants}}$$

There have appeared in the literature, many interpretations of how to implement the mutation activity. The usual one, called strong mutation deals with the comparison of the original and mutated program outputs at the end of the execution process. Strong mutation is usually performed by introducing one alteration per program variation. This type of mutant is called first order mutant [16]. By simultaneously embedding many alterations to the program, the produced mutants are called higher order mutants [16–18]. Another mutation variant interpretation is weak mutation [8,5], which tries to deal with the comparison of the program outputs immediately after the execution of the altered fraction of the program. In the present paper, the second approach is considered for two main reasons: First, as suggested in the literature [5,19,20] weak mutation constitutes a good alternative to strong mutation, and second, it is more straightforward to generate the relevant data as they are better embodied in the proposed path generation method as identified in existing attempts [21,15].

2.4. Generating test cases

This section introduces the underlying concepts and issues, for the generation of test cases, as they are employed in the present paper.

2.4.1. Symbolic execution

The symbolic execution [10,11] technique, analyses the source code of a program by replacing its actual input parameters with symbolic ones and simulating its execution based on a set of program paths. Executing a path symbolically, forms an effective way of describing its computations. Symbolically executing all possible program paths can result in the verification of the entire program. The method represents possible input values of program variables along a selected path as algebraic expressions by interpreting the operations performed along that path on the symbolic inputs. Thus, a symbolic state for a given point according to a selected path forms a mapping from input variables to symbolic values and a set of constraints called path condition over those symbolic values. Path conditions represent a set of constraints and symbolic expressions represent the mapping between the input variables to symbolic values and all together form the computations performed over the selected path. A symbolic expression is either a symbolic value or an expression composed over symbolic values, thus composed of variables, parentheses and programming language operators. Constraints are treated-evaluated as Boolean values (True or False) and composed of pairs of symbolic expressions related by one of the conditional operators (==, !=, <, $\leqslant$, >, $\geqslant$).

A path condition thus, forms a conjunctive constraint set, obtained from the decisions of the selected path. For each program decision encountered, in a path, a new condition is built by replacing all variable references by its previously computed symbolic expressions and based on the decision's outcome (true or false value) as formed by the path. Solving the path conditions results in input values which when assigned to the program, execution is driven along the selected path. If the path condition has no solution then the path is infeasible.

2.4.2. The feasible path and equivalent mutant problem

Gabow et al. [22] proved that the problem of obviating infeasible paths can be reduced to the "Halting Problem" thus forming an undecidable problem. In another study conducted by Budd and Angluin [23] it is proved that, in general, there is no computable procedure able to identify the functional equivalence between two given programs, argument which can be used to imply the undecidability of the equivalent mutant problem. Offutt and Pan [24] realised that the problem of equivalent mutants is an instance of the feasible path problem. The feasible path problem was formed as the problem of finding appropriate test data that satisfy a given testing requirement such as covering a node, branch, path or mutant. In view of this, they defined a set of constraints that represent conditions under which a mutant will die (in the weak mutation approach). The constraints system that forms these conditions is formed by the logical conjunction of a specific to each mutant constraint(s) with the disjunction of all paths conditions that reach the target mutant. A similar approach, based on program slicing, was presented by Hierons et al. [25]. Additionally, this approach seeks potentially affected locations by a mutant, thus providing guidance to the tester.

2.4.3. Selecting tests that kill mutants

Practically, applying the testing criteria serves as a rule of thumb for selecting a subset of all possible inputs. The main objective of these rules is to guide the selection of test cases to those able to expose the majority of the program's faults. Testing to fulfil a criterion establishes an associative, to the criterion used, confidence level.

Generally, in order to expose a fault, a test case should necessarily execute the program's source code part, containing it. Needed is also the execution of the fault so as to incorrectly affect the program's state and this incorrect state to propagate to the programs output. The formulation of these three situations serves as the foundation guide to generate tests according to the mutation criterion and in the literature these are referred to as reachability, necessity and sufficiency conditions [21]. Based on these three conditions DeMillo and Offutt developed a test data generation technique called Constraint-Based Test data generation (CBT) [21].

Current test data generation approaches [21,26–28] try to utilise directly the reachability and necessity conditions. The sufficiency condition due to its high complexity is indirectly satisfied either via the satisfaction of the reachability and necessity conditions only [26,21] either with the use of some additional heuristics such as the mutants' impact on the program execution [27], the search of the path space [29] and the fitness guided search of the path space [28]. As presented in [21,19,28], tests that meet the reachability and necessity conditions have a higher chance of meeting the sufficiency condition too than those meeting the reachability condition only. These approaches describe the reachability conditions as a disjunction of the path conditions derived based on the symbolic execution of all paths reaching a target mutant. The necessity conditions are described based on the following formation:

Let the original expression be e and the mutated one e'. Then, the condition is formed by the necessity constraint " $e \,!= e'$ ".

2.5. Practical problems posed by mutation

Practically applying mutation requires the completion of four main activities. Each one of the performed activities introduces various difficulties which make the use of mutation quite expensive. These activities are: (a) the mutant programs generation, (b) the test case generation, (c) the test case execution and (d) the evaluation of the achieved mutation score. All these activities are mainly influenced by the number of the introduced mutants which in practice turns out to be enormous. Specifically, the number of the introduced mutants has been quantified as $O(\text{Vals} \times \text{Refs})$ [30], where Vals is the number of data objects and Refs is the number of data references. Reducing the number of the introduced mutants forms one of the key research issues of mutation testing research. To date, two main approaches have appeared in the literature. The first one considers only a small sample of the whole mutant set which is selected at random [31,16]. The second one, named selective or constraint mutation [32,16], applies only a subset of the mutant operators. Both these approaches achieve to considerably reduce the introduced number of mutants, nevertheless, the problem of the huge number of the introduced mutants still remains.

The generation of mutated programs naturally requires the production and compilation of many program versions, one per introduced mutant. In practice, as this number can be enormous, the mutant generation results in a huge compilation effort. To alleviate this problem, various approaches have been proposed in the literature. The most important ones are the mutant schemata [33,30] the compiler-integrated [34] and interpreted approaches [31]. The mutant schemata method "enables to encode all mutations into one source-level program" [33,30] and thus avoiding performing a huge number of compilations. The compiler-integrated method utilises a special compiler that produces code patches that act as the mutants when applied to the code under test. Thus, instead of compiling the various mutant programs, only the patch application is required. Finally, interpreted execution approaches [31] can also be used in order to avoid the huge compilation overheads. It is noted that the prototype developed for the present research, utilises this method.

Mutation score evaluation poses the need for executing the introduced mutant programs with the produced test cases in order to determine the killed ones. In practice mutant execution requires a huge amount of resources [8,5] as the number of mutants may be enormous. To tackle this problem, various mutation testing alternatives have been proposed. The one utilised in the present paper is named weak or firm mutation [8,5,20]. Additionally, performing mutation requires the identification of the equivalent mutants in order to evaluate the adequacy of the produced test cases. This is a well known undecidable problem as discussed in Section 2.4.2 and thus only heuristic approaches can be utilised.

All the abovementioned issues are open for future research and are not directly dealt with in the present paper. However, successfully resolving them will make mutation to be widely adopted in practice. The main point addressed by the present paper is the automation of the test case generation according to mutation. This forms a difficult and open research problem of mutation [18]. Additionally, addressing the test case generation problem paves the way for the complete automation of the mutation testing process. Generating test cases for killing mutants, introduces some special challenges not faced in the case of structural testing. It is these challenges that are tackled by the approach described in the present paper and details are given in the following sections.

2.6. Related work

Although mutation testing has been suggested and studied for over three decades now, strategies that guide the generation of test cases and its practical employment are very limited. A small number of research articles dealing with the generation of test cases or the mutation testing application effort have appeared in the literature as also pointed out in [18].

DeMillo and Offutt proposed the CBT method [21] which briefly discussed in Section 2.4.3. This method tries to describe in algebraic expressions, the conditions under which a mutant is killed. They introduced the reachability condition which states that the mutated code must be exercised by the execution path of the test cases. If a test case fails to execute the mutated code, it is guaranteed that the test case has no chance of revealing the seeded mutant [21]. This is a direct consequence of the mutation application as it introduces one syntactic change to every mutated program and thus both execution paths, the one of the original and the other of the mutated program, form the same execution computations. The necessity condition states that the execution of the mutated statement must cause a discrepancy compared to the original program state [21]. The sufficiency condition states that the infected program state must propagate up to the last program statement. The execution path and its computations must use the internal different value calculated at the mutated statement (necessity condition) and must form a different observable computation from there onwards to the program's output.

The CBT approach [21] uses control flow analysis, symbolic evaluation, mutant related constraints, a constraint satisfaction technique and test case executions in order to automatically generate the required test data. The method targets on the reachability and necessity conditions while assuming that fulfilling them will also indirectly meet the sufficiency condition too. These two conditions are described as mathematical systems of constraints which are conjoint and solved by a constraint satisfaction technique called domain reduction. The reachability condition is described by a path expression of all program paths from input to the mutated statement node, in fact all loopless program paths. The necessity conditions are described by a specific, to each mutant, expression(s) in order to infect the program's state immediately after the mutated statement. The above approach has been implemented in a tool which integrates with the Mothra [31] mutation testing environment and automates the mutation based test case generation for programs written in the Fortran programming language. Although empirical evaluation of the CBT technique reveals its effectiveness [21], yet it suffers from many shortcomings associated with symbolic evaluation. Specifically, problems can occur in the presence of arrays, loops, nested expressions and non linear expressions as recognised in [35,5]. Moreover, issues such as the exhaustive generation of program paths affect its cost-effectiveness also appear.

In an attempt to address some of the drawbacks of the CBT method, the Dynamic Domain Reduction (DDR) [35] method was proposed. The DDR method was developed based on the CBT approach by embedding some dynamic features in order to effectively handle program constructs and develop a more efficient constraint satisfaction technique. The DDR method tries to generate, in a similar fashion to the CBT method, a set of values for each input variable (its domain), that makes the path expression true, for all the set values. Its difference comes from the reduction procedure that is performed based on a search method over the input variables domains. Specifically, the method traverses a selected path and at each encountered branch predicate it reduces the input variables' domains. This is done so as to evaluate these branch predicates to "true" for any assignment of values from the reduced domain set. When all selected predicates have been examined, the reduced domains form sets of the required test cases that traverse the selected path. In the work of Baars et al. [13] an attempt to generate test cases based on search based testing is suggested. This work introduces a symbolic execution based fitness function, based on the symbolic information of selected paths, in order to efficiently guide the test input search. This approach shows that path selection strategies when integrated with search based testing are able to provide practical solutions to the symbolic execution difficulties.

Other dynamic approaches based on searching input domain sets have been proposed for various testing criteria. Ayari et al. [36] proposed an evolutionary approach based on the Ant Colony Optimisation (ACO) to automate the generation of input variables. This work is based on the foundations of the CBT approach by measuring how close a test case is to reach and kill a mutant. A generic approach on using state of art techniques such as the dynamic test generation approaches has been proposed in [26] by utilising mutant schemata. The mutant schemata technique [26] achieve to help automated tools to perform mutation, by reducing the weakly killing mutant problem to the covering branches one. Thus, existing structural automated tools and techniques can be directly utilised for performing mutation. More recently, Harman et al. [28] proposed a hybrid approach based on Dynamic Symbolic Execution and Search Based Software Testing in order to effectively kill either first or higher order mutants. This approach aims at fulfilling the mutants' sufficiency condition via search based testing. Their results suggest that by doing so an average improvement of approximately 15% and 16% on the mutation scores of first and second order mutation can be gained.

The mutation testing effort depends on the number of mutants involved. Strategies involving mutation should therefore attempt to limit the number of the mutants introduced, while avoiding the introduction of equivalent ones. Such an approach is proposed in [17] where the construction of higher order mutants with the use of search based optimisation approaches is utilised. In this work it is suggested that the number of mutants and equivalent ones can be dramatically limited by introducing more than one mutant at a time. Sampling higher order mutants can also reduce various cost factors of mutation such as the number of introduce mutants, equivalent ones and the number of required test cases [37]. However, both of these approaches rely on mutation analysis rather than on generating test data as in the present paper.

3. The method used

The aim of the present paper is to provide a practical insight on generating test cases for killing mutants. To this end, a path based strategy is proposed along with its assessment on the required effort with respect to the generated candidate sets of test paths. Additionally, guidelines for the application of mutation testing are also given, by measuring the ability of killing mutants utilising test paths. The reason behind the choice of candidate sets of paths as a measurement of the effort stems from the observation that infeasible paths greatly influence the effort associated with the testing process [6,7]. This is evident from the following experiments where a huge number of paths are analysed in order to find a small number of feasible paths, see Section 6.3 for a discussion on the influence of infeasible paths on the testing effort. Additionally, following the propositions made by Weyuker [38], the hidden cost of infeasible – unexecutable testing requirements of the various testing criteria must be taken into account when evaluating their effort. A method embodying these observations is described and analysed in this section.

3.1. The extended shortest path method

The a priori prediction of the infeasibility of a program path is an undecidable problem and thus heuristic techniques that automatically select likely to be feasible paths can be employed only. In view of this, Yates and Hennell [39] advanced, and argued the proposition that:

A program path that involves $q >= 0$ predicates is more likely to be feasible than one involving $p > q$.

Formal statistical investigation of this proposition was undertaken in Yates and Malevris [14], wherein it was concluded, with great statistical significance, that the feasibility of a path decays exponentially with the increasing number of the predicates it involves. As a result, Yates and Malevris [14] proposed a path selection method, to reduce, a priori, the incidence of infeasible paths amongst those that are generated for the purpose of branch testing. Although introduced to support branch testing, the method was founded only upon a consideration of the number of predicates in a program path. Thus, the method does not seek to optimise, in any way, on any one testing criterion. Hence, it maybe used with equal validity, and without bias, in an attempt to fulfil any other testing criterion. It is this path generation method's spirit that was extended to allow its use to effectively produce the sought test cases for performing mutation testing.

In general the proposed approach selects candidate paths from the enhanced control flow graph model which is detailed in the next subsection. Specifically, the selection of the path sets is performed relying solely on the adopted graph representation. Thus, the only factor that affects the consideration order of the selected paths is the paths' length, which is represented by the sum of the contained nodes weights.

The strategy for illustration purposes herein can be outlined as:

```
While ($MS(C, D) < 1$ AND $i < k$),
        Repeat steps 1, 2 and 3 once for each mutant element e in set M.
    1.  Generate the next candidate path $\Pi_e^i$ (aiming at mutant element e)
    2.  If path found to be feasible, then recalculate the value of Coverage based on actual execution.
    3.  Eliminate all covered elements from the mutant set M.
```

Here, Π_e^i denotes the ith shortest path through element e and k = maximum number of practically viable paths (beyond which the generation of extra paths is prohibitive). Steps 2 and 3 actually generate and execute the required test data against the live mutants.

The basics of this method are derived directly from the propositions about path feasibility and the number of predicate constraints they involve [39,14]. The validity of the analogy between predicates and mutant constraints in respect of selecting likely to be feasible path sets is well established. Mutant related constraints are constructed through simple syntactic source code alternations based on arithmetic, relational and logical operators, ingredients of predicate constraints. Moreover, the majority of the considered conditions contained in candidate test paths are predicate conditions. Nevertheless, due to the use of mutant constraint sets in conjunction to predicate ones, may naturally result in an increase of the incidence of infeasible paths. This increase is totally attributed to the nature of mutation testing affecting its overheads. Accordingly, it is imperative that any method applied to fulfil a high demanding criterion such as mutation, must be able to alleviate the undesirable effects of infeasible paths.

There exists a number of fundamental matters that need to be detailed in order to proceed with the employment of the proposed method and are summarised as follows:

1. Path based mutant representation and selection mechanism.
2. The method used to generate the Π_e^i.
3. The interpretation of the criterion "$MS(C, D) < 1$".
4. The criterion for selecting uncovered mutants.

Their details are given in the succeeding subsections.

3.2. Mutant representation and path selection

The representation of the structure of a code unit by its CFG is well known, understood and forms the basic model of path testing for most of the white box testing criteria. Unfortunately, this cannot be considered to be the case for mutation since mutants are not represented or identified by the CFG structure. Additionally, mutation testing requires an execution path in both the original and the mutated program versions in order to cover-kill the corresponding mutant. For mutation a more elegant model is needed [15]. In accordance with the suggestions made in the literature concerning weak mutation [8] and by using mutants' necessity conditions [21], it is possible to construct an augmented CFG by enhancing it with mutant related constraints together with the corresponding mutants. The enhanced graph is built simply by adding a special type of vertex for each mutant representing its necessity constraint. Thus, for each introduced mutant one additional node is introduced in the enhanced graph.

Every introduced mutant is associated with a CFG program node, the one that the mutant appears in the source code. Every mutant related vertex is connected with its original corresponding node (original CFG) and represents a special mutant related necessity constraint. A dummy node is introduced in order to connect all the mutant vertices with the original CFG and in order to keep the original CFG unaffected. Fig. 1 demonstrates the construction of the enhanced control flow graph where both graphs are displayed. The proposed approach test model is completed by assigning weights to the CFG nodes. The original CFG nodes bear a weight value calculated as defined in Section 2. The mutant related nodes are assigned an infinite weight value in order to be ignored through the path selection method. It is noted that the selected candidate paths for a specific branch are the kth-shortest ones that reach the considered branch. The infinite weighted nodes will thus be ignored by the selected paths as by generation they must be shortest.

Since a path not containing any mutant vertex represents an exact path in the original test code unit, that path can form the basis for the fulfilment of the mutant's reachability conditions. Additionally, every feasible path containing one or more mutant vertices fulfils both the reachability and necessity mutant conditions of the respective mutants. Conversely, an infeasible path from the beginning of the graph up to and including a mutant, say $m(a)$, indicates the absence of possible test data to reveal it, hence, if no such a feasible path exists, this leads to the determination of the mutant as being equivalent. Covering all enhanced control flow graph mutant nodes, results in a straightforward coverage of the weak mutation criterion and the fulfilment of the reachability and necessity conditions of all program mutants too. In the remainder of the paper the term coverage is used to indicate the coverage of the enhanced control flow graph mutant nodes which is actually the mutation score achieved with respect to weak mutation. Based on the above observations the proposed strategy tries to effectively select paths that will cover the corresponding mutant vertices of the enhanced control flow graph. In order to avoid confusion it must be noted that the selected paths are used in order to guide the test generation process on producing tests and not on indentifying infeasible ones. Infeasible path identification is performed by the test generation method utilised, which in our case is the symbolic evaluation. By selecting the kth-shortest paths, a higher probability of the path feasibility is established.

One possible concern about the enhanced control flow graph is its size, which grows in relation to the mutants' number. This practice may result in huge graphs as the number of the introduced mutants can be an excessive one. Despite this, the graphs' size is not a problem as it forms only a representation of the program under test. Additionally, the use of kth-shortest paths along with the infinite weights of the mutant nodes on the enhanced control flow graph results in considering only the original program CFG nodes. Thus, the introduced complexity of the mutant vertices is alleviated and their impact on the path selection method is negligible.

3.3. Incremental selection of paths

The path generation method used for mutation testing is in analogy to that used for testing branches. The targeted candidate path set consists of the kth-shortest program paths, again in the sense of containing a minimum number of predicates, which together cover the inserted mutants. These kth-shortest paths through a mutant vertex m can be defined as follows:

$$\Pi_m^{(k)} = \Pi^{(k)}(S, x) \oplus xm$$

where $\Pi^{(k)}(S, x)$ denotes the kth shortest path from the starting vertex S to the target vertex x of G_C. The introduced arc that connects the vertex x with the mutant node m is denoted as xm, and $\oplus$ denotes a sequence. All mutants are handled based on a k-value, the limit on the number of generated paths, used by the strategy. The practicality of the method relies on the selection of an appropriate k-value beyond which all left alive mutants are treated as being equivalent. Also, some additional advantages of embodying this method are that it uses the philosophy of suggesting paths with the least number of predicates, and in particular, it even considers shorter paths to symbolically execute as it only considers the paths up to the mutated node and not entire entry to exit paths. This incident not only increases the possibility of such subpaths being even more likely to be feasible, as they contain fewer predicates, but also reduces the execution cost per path when symbolically executed as it will be discussed.

3.4. Handling infeasible requirements

Generally the goal of full coverage (killing all mutants) can be accomplished only when $MS(C, D) = 1$. $MS(C, D)$ is the ratio of the killed mutants constituting thus a test effectiveness measure. Unfortunately, in practice, to fulfil $MS(C, D) = 1$ is a very tedious exercise. Even with less demanding criteria such as node coverage where a similar test effectiveness ratio needs to be justified, full coverage of the nodes can be difficult to achieve due to the presence of infeasible paths. In mutation testing the existence of equivalent mutants makes it rather impossible to kill them, hence, making it difficult to achieve full coverage (killing of all mutants). This compels the necessity of defining the relative coverage $MS^*(C, D)$ as the ratio of killed mutants to the number of not equivalent mutants. Such being the case, it is required to deduce and

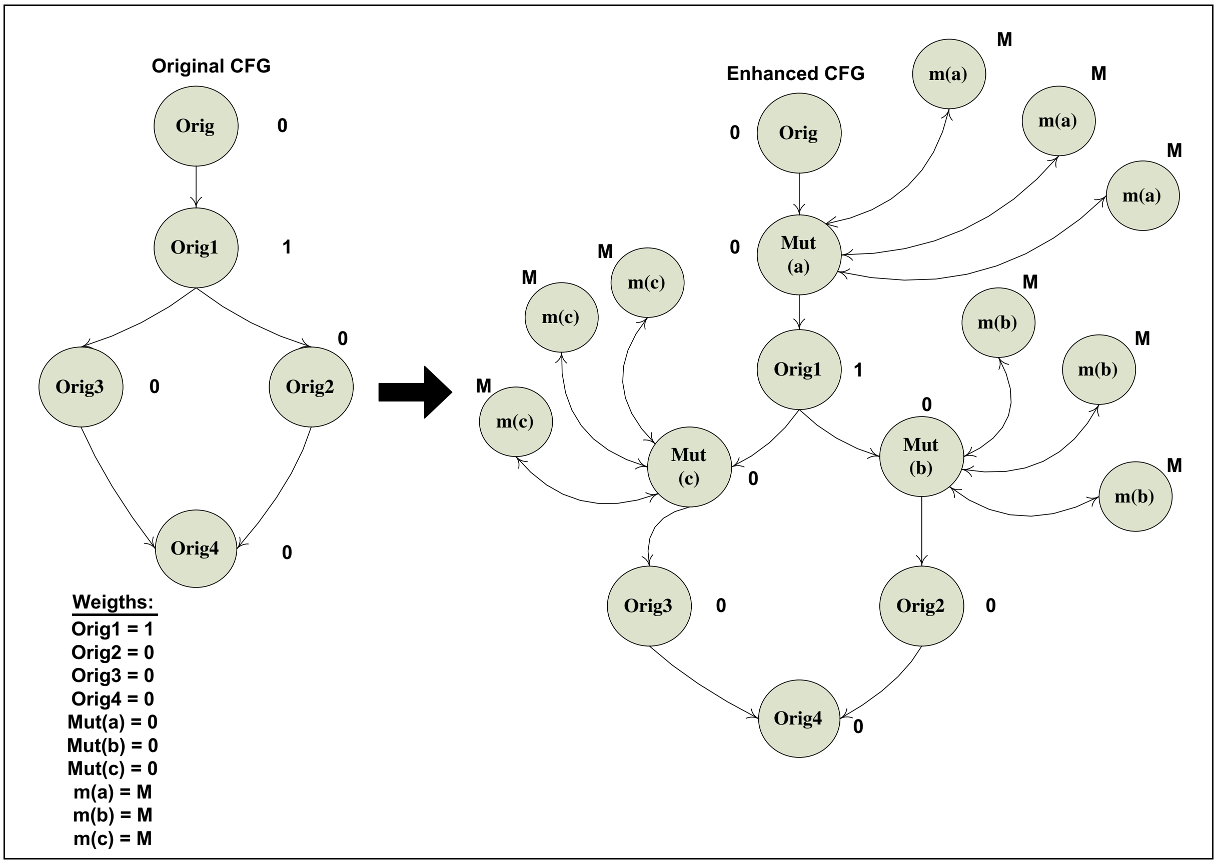

Fig. 1. The enhanced control flow graph.

eliminate all equivalent mutants in order to proceed with the calculation of $MS^{*}(C, D)$. This task can only be performed heuristically, as it is not possible to directly deduce all the equivalent mutants. As a consequence, in the experimental study presented in the next section it was necessary to adopt an effective and practical way of deriving the equivalent and killable mutants. Following the spirit of the proposed approach a reasonably high threshold of approximately 50,000 paths per mutant was assumed. Beyond that threshold all mutated versions were considered to be equivalent with the original. The reason for assuming such a threshold is based on the nature of the effort-coverage relation which indicates the exponential trend of the effort as recorded by the experiment performed in Section 5. Also the contribution to the totally killed mutants beyond the 30,000 paths (Table 4) is insignificant.

3.5. The criterion for selecting uncovered mutants

At each iteration of the proposed approach there is a set of live mutants needed to be covered. In almost every case the set contains more than one uncovered-live mutant elements. The question that is raised is: Which is the order they should be selected in an attempt to be killed? Alternatively, which mutant should be tried to be killed first? The answer to the above question is given by adhering to the well established philosophy that: *a program path that involves $q \geqslant 0$ predicates is more likely to be feasible than one involving $p > q$*. Specifically, the remaining live mutants are classified according to the length of the shortest path each one lies on. Thus, attempts to increase coverage are always ordered in respect of a path's length. Due to this ordered rule, the candidate mutant that is selected to be killed next is the one whose path containing it is the shortest in length among the others. Ties are broken arbitrarily.

4. A unified mutation testing system

The presented approach and the succeeding experiments were conducted based on a prototype tool, implemented for the present study. The tool automatically generates and executes test cases by employing the concepts presented in the preceding sections. It has been entirely built using the java programming language. It also incorporates the LpSolve [40] package for solving path conditions. It operates on programs written in the SymExLan [41] script language and utilises the experience gained from the previously developed tool Volcano [42]. A similar in philosophy tool is proposed in [43] with the aim of producing test cases for branch testing. The preset paper expands the suggestions made in [43,15] for targeting mutants.

4.1. Overview

A high level view of the prototype architecture is presented in Fig. 2. The prototype is composed of the following five modules which are represented as rectangles:

Parser: This module analyses the SymExLan scripts and generates an appropriate test model.

Mutant Generator: It enhances the test model with mutant related constraints.

Path generator: This module generates candidate paths for a given target (branch or node).

Symbolic Executor: The unit that symbolically executes selected paths, constructs path expressions and transforms them to linear programming problems

Interpreter Verifier: It executes and enhances with random values the actual test cases generated. Its main use is to calculate the achieved coverage.

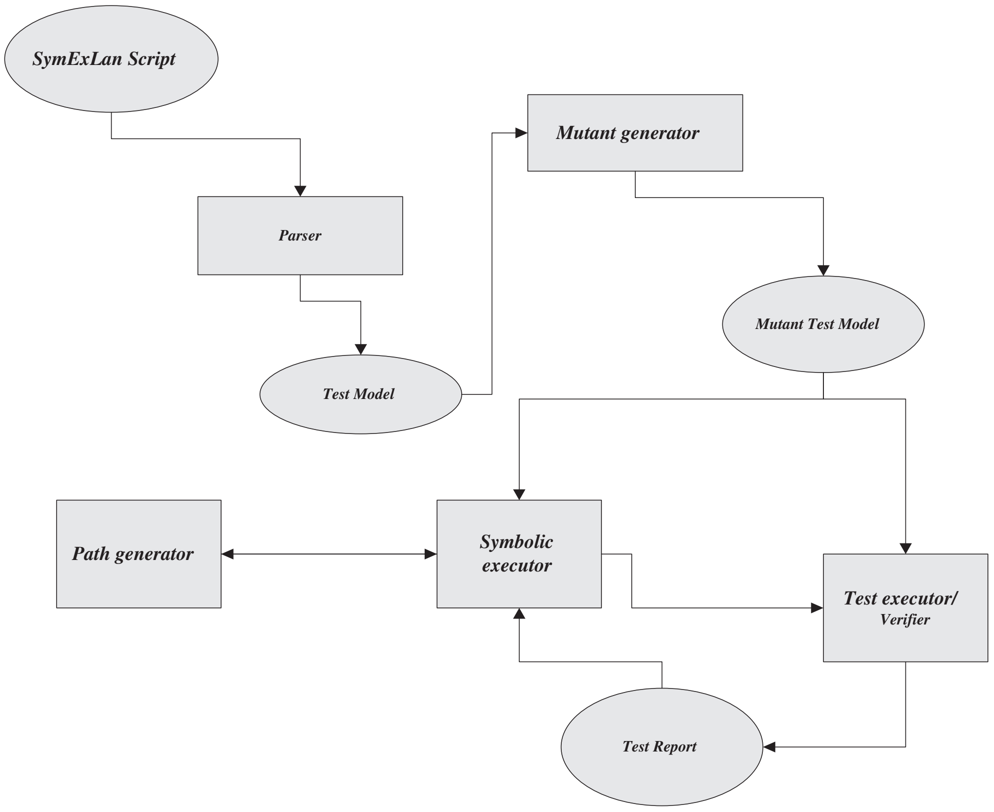

Fig. 2. Tool's architecture.

It also embodies some intermediate structures and reports, represented as ellipsis, such as:

Test Models: The model of CFG and Enhanced CFG with their corresponding computations.
Test Reports: Reports containing the test execution path and a list of the killed mutants by a test case.

4.2. Detailed description

4.2.1. Parser

Parsing is handled by a program that reads SymExLan scripts [41] and generates the system internal test model, containing the original Control Flow Graph and a set of algebraic computations corresponding to each basic block-node. The SymExLan scripts are automatically produced from external source to source compiler systems. For integrity reasons, Fig. 3 gives an example of such a script for a method that calculates the absolute expression of a float number. Further details about the SymExLan scripts can be found in [41].

4.2.2. Test models

The test models (original and enhanced models) contain a graph representation of the program's control flow containing in each node its respective symbolic computations. All symbolic computations have been transformed to simple ones, by transforming all compound predicates into multiple simple computations. By doing so, during symbolic evaluation, the path conditions of all selected paths are represented by a conjunctive formula of symbolic computations. Thus, all path expressions can be straightforwardly transformed into linear programming problems with a relative small and equal to the number of the programs input variables. An additional argument behind keeping all symbolic computations is that during symbolic evaluation the number of variables is kept to a minimum. This results in a higher performance when employing optimisation methods such as linear programming, to solve the path constraints.

4.2.3. Mutant generator

The mutant generator module is responsible for generating mutant constraints and enhancing the test model appropriately. The generated mutant constraints are the necessity ones as suggested by the CBT approach [21]. Thus, every mutant constraint follows the constraint template of " e ! = e' ", where e adheres to the original expression and e' to the mutated one. Every mutant vertex corresponds to its necessity constraint in conjunction with some sufficiency ones. The path condition generated for a selected path that reaches a mutant vertex corresponds to a conjunction of both the reachability and necessity conditions. In order to reduce the weak mutation inefficiency some additional sufficiency constraints based on the propositions of firm mutation [20,19] are followed. In accordance with the suggestions made by Offutt and Lee [19] the necessity constraints are extended in order to produce differences in the program predicates and in particular at the end of the basic block that contains the mutated statement. In occurrences of logical operators some additional complementary constraints are used in order to cause the mutant predicate differences to propagate to the whole decision (avoiding masking through the remaining logical conditions). These additional constraints are formed based on the consideration of the mutated part of the logical expression as the basic one, which should independently affect the decision's outcome, while treating the rest of the expression in accordance to the MCDC criterion. These conditions are generated by using the tree method as proposed by Offutt et al. [44].

In the present paper an experimental study of the effort entailed by the weak mutation testing method has been undertaken. For

```
NAME   ABSOLUTE
TYPE   float
IDENT  1
GRAPH   [Start] true [1]
       [Start] false [2]
       [1] always [Final]
       [2] always [Final]
PATHS  Path1 = [Start]-[1]-[Final]
       Path2 = [Start]-[2]-[Final]
NODES  [Start:IF]
             COND  : (x<0)
             REFS  : 1,2
       [1:SIMPLE]
             ASSN  : absolute=-x
                   REFS  : 3
       [2:SIMPLE]
             ASSN  : absolute=x
                   REFS  : 4,5
       [Final:SIMPLE]
             REFS  : 6
DECLARATIONS       PVAL: X is float in PROCS(1)
```

Fig. 3. SymExLan script example.

Table 1
Utilised necessity conditions.

Operator	Conditions
Relational expression (*a* op *b*)	$a > b$ $a < b$ $a == b$
Logical expression (*a* op *b*)	$a == T$ && $b == T$ $a == F$ && $b == T$ $a == T$ && $b == F$ $a == F$ && $b == F$
Absolute arithmetic variable (*a*)	$a == 0$ $a > 0$ $a < 0$
Arithmetic expression (*a* op *b*), mutant operator (*m*(op))	*a* op *b* ! = *a* *m*(op) *b*
Unary predicate expression (exp)	exp ! = exp + 1 exp ! = exp − 1

experimental purposes the effort is directly associated with the considered candidate mutant set. Of the possibly excessive number of mutants, the considered set is a reduced one, as this has been proposed in [45]. Based on this study, the candidate set is similar to the one proposed by Offutt et al. [32]. The discrepancy between the two is that the set as in [45] produces a smaller sample as it considers additional restrictions that remove some redundant mutants. Thus, the selected mutants have the same strengths with the full mutant set. In other words, it was found in [45] that tests that are capable of killing the reduced set of mutants are also capable of killing all the introduced mutants of the whole set, as proposed by Offutt et al. [32]. This is not contrary to the validity of the experiment since the main issue is to concentrate on effective ways of killing the mutants irrespective of their number. The existence of more mutants would have possibly added only minor additional effort. Table 1 describes the produced necessity constraints of the produced mutant set. The resulting set uses all five mutation operators (Relational, Logical, Arithmetic, Absolute and Unary), as proposed by Offutt et al. [32]. Thus, a set of three mutants is produced based on the Relational operator; a set of 2^n mutants is produced based on the number of operands n in the compound expression due to the Logical operator; a set of three mutants is produced according to the Absolute operator (following the original description [46]); a set of five mutants is produced according to the Arithmetic operator; a set of two mutants based on the Unary operator is produced for each occurrence in a decision statement, as in the rest of the occurrences in other statements, as it has been argued by DeMilo and Offutt [21] killing the mutants is a fairly elementary task.

4.2.4. Path generation

The path generation module generates a set of *k*th-shortest paths from the graph's start point to a given target node. There are various graph theoretic algorithms in the literature for tackling this problem. The system actually uses an adaptation of the method suggested by Yates and Malevris [14]. It is noted that in both the original and the enhanced test models the paths generated are identical with the addition of the mutant node and the arc connecting it in the case of the enhanced CFG.

4.2.5. Symbolic executor

The symbolic executor module is integrated with the path selection strategy and performs symbolic evaluation for a given/selected path in order to derive its path expression in terms of input variables. This expression is then transformed into a linear programming problem and solved based on linear optimisation techniques [47]. In case the linear programming problem results in a solution, its solution values do form the required inputs. In the opposite case where no solution can be found, the path is determined to be infeasible. The Linear Programming problem is formed in a straightforward way as the test model adopted uses only simple constraints which are directly incorporated into the linear problem formulation. The objective function to be max(min)imised can be a trivial one as it can be satisfied by any solution of the linear problem in the solution space. Here, it must be noted that the logical expressions are handled based on the selected paths made on the adopted graph model. As stated in the previous sections, compound – logical predicates have been split into multiple simple computations and thus, their handling is based on the path selection method used. Different selected sub-paths of the simple computation nodes that correspond to one compound predicate represent different logical computations.

One of the general problems encountered is that of handling non-linear expressions. To tackle this problem the prototype tries to isolate these expressions by eliminating them and based on actual executions to overcome it in an indirect way as suggested by [42]. Other approaches for handling non-linear expressions can be found in [48,49]. Other known problems of the linear programming and general to the constraint solving for the symbolic execution are the handling of string constraints. This is a matter beyond the scope of the present research and not considered. A possible approach for handling it can be found in [50] where a constraint solver over string constraints is proposed. Other difficulties may arise due to the presence of pointers and tables. Their handling is based on the approach proposed by the work of Koutsikas and Malevris [42] who suggest the use of a memory table. A similar approach based on enumeration and propagation of the alias relations is described in [51]. These approaches deal with the ambiguous arrays and pointer references effectively. In fact they achieve to solve all the encountered ambiguous references. However, they have the drawback of introducing additional constraints to the produced path conditions. In the performed experiment, presented in the next section, the employed approach [42] achieved to solve the encountered ambiguous references and thus, determining the generated feasible and infeasible paths accurately.

4.2.6. Mutant execution

Test execution tasks are performed based on an independent to the test generation system, responsible for verifying the killed mutants and enhancing with random values the redundant to test generation input variables. The system is based entirely on an interpreter that simulates the actual executions and computations performed by the considered test program. Based on the appropriate input values (determined by the symbolic execution module) a simulated execution is performed on each graph node of the test model and its execution path is traced. As the simulated execution continues beyond the mutated point in the program under test, it traverses code that has not been executed symbolically. This fact may result in a need for new input values, these are given based on their type and application input domain through random selection.

All introduced mutants are evaluated with only one execution run per test case. This is an advantage of performing weak mutation instead of strong one. This is in accordance to the suggestions made by Howden [8] who advocates that in weak mutation it is acceptable to execute all reachable mutants in one program place. The proposed approach generalises this argument to all the reachable mutants by the traversed execution path. This is achieved by performing the simulated execution and evaluating all the reachable by the test cases mutants. When the simulated execution reaches a mutant related node it performs an evaluation check on all its mutant constraints in order to determine if they would be killed or not. Then, the execution continues by returning to the mutant related node (dummy node) in order to keep on with the original program execution. This way, all mutants encountered in the examined execution path are evaluated.

The use of the test executor module compliments the symbolic execution method by embedding dynamic features such as random testing and actual execution evaluations. It also serves as a validation system of the generated tests and helps in verifying the killed mutants.

5. Experimental regime

The experimental results reported in this section were obtained by the application of the proposed strategy to a sample of 30 program units, a list of which is given in the Appendix. All the selected units have no calls to other functions as these have been "inlined" in the units under test. The sample consists of programs written in various programming languages some of which have been referenced in the literature [21,6,43]. The units were selected based on their use in the literature, the observation that they contained certain important features such as: nesting, loop constructs, use of arrays and also many infeasible paths. The sample consists of 10 code units written in Delphi (units 1–10) and 20 code units written in C and other languages (units 11–30). In Table 2 details of the selected sample units are presented, where the LOC denotes the number of lines of code that the units are composed[1] of, the number of generated mutants and the number of live mutants they contain. The column housing the live mutants records the number of mutants remaining alive after the completion of the experiment. For the reasons analysed before, i.e. in Section 3.3, they are treated as equivalent. Each program unit was first transformed into the SymExLan [41] script language (the intermediate representation used in the tool). Following, they were assessed by the tool that uses the proposed method for path selection for performing weak mutation testing as this has been presented in detail in the preceding sections. The application of the proposed method to these units resulted in the generation of well over 55 million program paths and in a total of 4517 mutants.

Table 2
Unit details.

Program Units	LOC	No. of branches	No. of mutants	No. of live mutants
overstrike	60	34	143	51
entab	48	31	123	35
getkbd	74	40	216	57
shell	40	21	127	52
dodash	59	27	164	48
triangle	44	31	170	43
newpas	59	33	158	52
newcats	51	26	112	36
catsub	96	56	224	42
gtext	80	48	171	51
tritype	70	51	281	81
try	22	13	102	8
atoi	46	27	124	23
entab2	40	22	99	35
91_function	24	12	72	24
calcnum	39	25	128	57
findmax	29	18	102	11
includenum	76	32	151	44
addqueue	51	31	135	32
findpos	45	28	97	14
checkdigit	42	33	126	19
checksides	27	19	52	0
seekpos	50	29	135	13
stringcount	42	25	99	19
wc	33	18	86	19
getop	72	48	238	51
remainder	74	38	289	116
triangle2	93	62	310	130
newdash	52	24	152	34
91_function2	38	23	131	37

5.1. Experimental results

The results obtained by applying the method are depicted in Table 3. There are three major categories for the different number of k paths considered in the experiment. More specifically, the k values of 1, 50, 300, 5000, and 50,000 are presented in the table, each sub containing three categories namely Coverage, Paths and Tests. These reflect the level of relative mutation score coverage

[1] The LOC were measured after their conversion to the SymExLan language by considering the Nodes and Declaration sections of the scripts [41].

Table 3
Results derived from the application of the proposed strategy.

Unit	*k* = 1			*k* = 50			*k* = 300			*k* = 5000			*k* = 50,000		
	Coverage (%)	Paths	Tests	Coverage (%)	Paths	Tests	Coverage (%)	Paths	Tests	Coverage (%)	Paths	Tests	Coverage (%)	Paths	Tests
1	40.22	165	4	93.48	5258	8	96.74	23,775	9	100.00	348,546	10	100.00	34,12,626	10
2	59.09	98	3	79.55	3680	7	79.55	20,180	7	100.00	239,052	8	100.00	22,66,752	8
3	98.11	108	25	99.37	4078	26	100.00	24,182	27	100.00	400,182	27	100.00	40,04,982	27
4	76.00	84	1	100.00	3404	7	100.00	16,405	7	100.00	232,605	7	100.00	23,05,365	7
5	61.21	122	3	68.97	5831	6	89.66	31,040	11	94.83	382,231	13	100.00	31,07,984	15
6	96.85	81	13	100.00	3071	14	100.00	18,321	14	100.00	305,021	14	100.00	30,53,681	14
7	77.36	93	1	86.79	4003	6	86.79	22,526	6	100.00	290,804	11	100.00	28,14,164	11
8	100.00	61	12	100.00	2462	12	100.00	14,712	12	100.00	245,012	12	100.00	24,52,952	12
9	78.02	129	12	89.56	5458	15	96.15	24,096	20	99.45	318,036	25	100.00	30,89,806	26
10	68.33	134	12	88.33	3914	14	94.17	19,973	16	97.50	277,532	19	100.00	25,94,710	21
11	52.00	255	14	100.00	1730	36	100.00	3022	36	100.00	3022	36	100.00	3022	36
12	100.00	51	20	100.00	737	20	100.00	4225	20	100.00	70,025	20	100.00	700,865	20
13	71.29	75	8	94.06	2053	23	94.06	10,463	23	100.00	92,715	25	100.00	674,015	25
14	98.44	53	5	100.00	2367	6	100.00	14,117	6	100.00	235,017	6	100.00	23,52,837	6
15	100.00	34	6	100.00	1308	6	100.00	7248	6	100.00	110,648	6	100.00	11,01,968	6
16	97.18	82	9	100.00	3562	10	100.00	21,312	10	100.00	355,012	10	100.00	35,54,272	10
17	83.52	49	14	100.00	738	17	100.00	4238	17	100.00	70,038	17	100.00	700,878	17
18	87.85	93	12	100.00	3177	17	100.00	16,681	17	100.00	235,617	17	100.00	23,08,377	17
19	63.11	97	6	95.15	2609	15	95.15	12,734	15	95.15	149,034	15	100.00	12,90,262	17
20	95.18	40	10	100.00	814	14	100.00	4574	14	100.00	75,074	14	100.00	750,974	14
21	99.07	84	26	100.00	1165	27	100.00	6650	27	100.00	110,050	27	100.00	350,369	27
22	100.00	22	16	100.00	22	16	100.00	22	16	100.00	22	16	100.00	22	16
23	100.00	53	17	100.00	886	17	100.00	5123	17	100.00	85,023	17	100.00	851,043	17
24	93.75	38	9	98.75	1226	13	98.75	7226	13	98.75	120,026	13	100.00	11,81,935	14
25	95.52	34	8	98.51	1115	10	100.00	6491	11	100.00	105,191	11	100.00	10,51,451	11
26	100.00	205	44	100.00	3439	44	100.00	19,873	44	100.00	330,073	44	100.00	33,04,033	44
27	99.42	147	14	100.00	3124	15	100.00	18,228	15	100.00	132,492	15	100.00	12,13,932	15
28	91.67	212	20	91.67	6239	20	100.00	32,719	23	100.00	371,115	23	100.00	28,49,415	23
29	60.83	120	4	70.00	5731	7	90.00	30,440	12	95.00	372,231	14	100.00	30,07,864	16
30	85.11	81	15	91.49	2486	18	96.81	12,273	19	100.00	135,341	20	100.00	13,06,901	20
Total	–	2900	363	–	85,687	466	–	452,869	490	–	61,96,787	512	–	576,57,457	522
Mean	83.35	96.67	12.10	94.52	2856.23	15.53	97.32	15095.63	16.33	99.33	206559.57	17.07	100	1921915.23	17.40

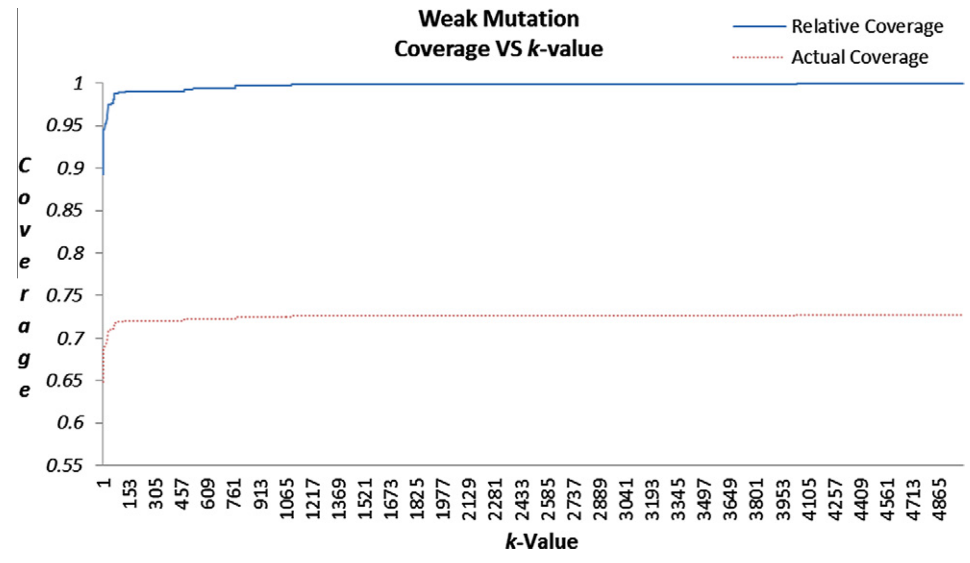

Fig. 4. Mean coverage for all k values.

$MS^*(C, D)$ achieved, the number of the paths generated and the number of feasible paths encountered, i.e. used for generating tests (tests entry in the table). Also, the total and mean values for all categories are tabulated. It must be noted here that for the purposes of the experiment and as these have been analysed in detail, a maximum of 50,000 paths were used beyond which, as it will be discussed in Section 6, it is prohibitive to generate more paths. Beyond that point, all live mutants were treated as being equivalent. The application of the method to each program unit resulted in a total of 522 feasible paths – tests cases. The number of so few feasible paths in such a big path set considered is attributed to the anomalies introduced by the insertion of the mutants which forced even feasible paths in the initial graph, to become infeasible as their presence created conflicting situations. The first iteration ($k = 1$) of the proposed approach realised full (100%) relative coverage for 6 units while for 16 units it achieved over 90% of relative coverage, whereas for the remaining 14 units the coverages achieved varied between 40.22% and 87.85%. The mean coverage achieved for all units reached the value of 83.35%. Subsequent iterations resulted in higher coverage values for all program units. In the first 50 ($k = 50$) iterations of the strategy, full relative coverage was achieved for 16 units and for only 6 (of the remaining 14 units) the score fell under 90%. It must be noted that the lowest achieved coverage was that of 68.97% for unit 5 and the mean value recorded was 94.52%. Beyond 50 iterations there is a high escalation of the number of infeasible paths generated and that resulted in a low coverage increment in the succeeding iterations. Thus, the mean increase of coverage for the paths generated between $k = 50$ and $k = 300$ increased by only approximately 2.80%. Coverage increase was experienced for 9 units where the values recorded varied between 0.63% and 20.69%. In total full relative coverage was experienced for 19 units with a value range of [79.55%, 100%]. It must be emphasised here that there is only a marginal increase in the coverage achieved despite the considerable amount of paths generated. This matter is discussed in Section 6, however, it must be highlighted that the effort related to the coverage changes, increases exponentially and may not be advisable to employ the method beyond specific k-values.

The results of the experiment for higher k-values ($300 < k < 50{,}000$) resulted in a 2.68% coverage increase. This was achieved

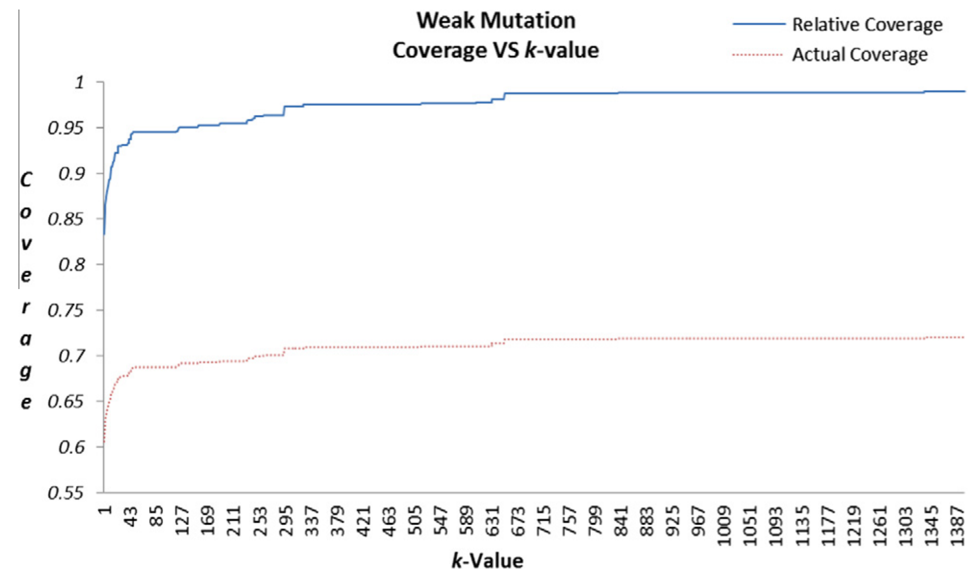

Fig. 5. Reduced mean coverage graph.

with only 32 total test cases (feasible paths) contributed in the above range of k values. In conclusion, for the last two columns of the table ($k = 5000$ and $k = 50{,}000$), it can be argued that for the first 5000 iterations 6 units still remain not fully covered, resulting in coverage values for all 30 units in the range [94.83%, 100%] with a mean coverage value of 99.33%. Additionally, 22 more test cases are produced at the iteration increment range from 300 to 5000, making a total of 10 test cases only found, i.e. 10 feasible paths only in the range from 5000 to 50,000.

The results obtained are summarised in Fig. 4, where the mean actual C_k and the mean relative C_k^* coverages for all values of k in the range [0, 50,000] against k are plotted. The values of C_k and C_k^* are defined as:

$$C_k = \frac{1}{30}\sum_{i=1}^{30} MS(i, D_k) \quad \text{and} \quad C_k^* = \frac{1}{30}\sum_{i=1}^{30} MS^*(i, D_k)$$

where $MS^*(i, D_k)$ and $MS^*(i, D_k)$ are the values of MS^* and MS and D_k the data set after k iterations of the proposed method for unit i.

It is self evident from the graph that the increase in coverage is quite considerable for minor values of k and beyond the 1400 value, only a subtle increase occurred. This enables the argument, without any loss of generality, that the significant contribution in the coverage achieved when attempting to kill the mutants is attributed to the first part of the graph where up to 1400 paths are recorder beyond which, the increase is insignificant hence can be ignored. Fig. 5 displays the reduced graph deduced from that of Fig. 4 where the values of coverages for k in the range [1, 1400] are considered. This has allowed to make the entries of smaller path values more apparent. As pointed before after the 50 iterations the coverage rate begins to slow and in fact, no additional increase is observed until the 120 iterations. The rate of coverage increase slows down and this becomes even more evident for k in the range of [630, 50,000], where the relative values attained are: for $C_{632}^* = 98.14\%$, $C_{913}^* = 98.90\%$, $C_{1605}^* = 99.00\%$, $C_{4764}^* = 99.33\%$, $C_{6912}^* = 99.45\%$, $C_{7740}^* = 99.76\%$, $C_{20296}^* = 99.85\%$, $C_{28110}^* = 99.88\%$, $C_{40295}^* = 99.91\%$, $C_{40396}^* = 100\%$. The actual values are spread in a similar fashion.

The reduction of the graph also makes more evident that the significant increase in the coverage achieved is recorded for the ranges [1, 50], [51, 150] and [151, 300]. The total coverages recorded in these ranges being 94.52%, 95.1% and 97.32% respectively. The next point of significant increase is for $k = 632$ where a relative coverage of 98.14% was achieved. Full relative coverage was achieved after the generation of paths for $k = 40{,}396$ making it more debatable that the benefit of such a subtle increase is worthless with respect to the huge number of paths needed to be considered in order to achieve it, hence involving more effort too.

5.2. The effort required by the strategy

Clearly as it has already been established there is a direct relationship between the number of paths generated and the coverage achieved when seeking to kill the mutants. The initial path sets do provide the basis in order to achieve a fundamental first coverage. Then onwards, a significant number of paths is required to increase the coverage and hence the percentage of killed mutants. The reason behind this, is due to the existence of equivalent mutants and infeasible program paths. The incidence of both equivalent mutants and infeasible paths require an incremental number of tries from the tester in order to find suitable tests-feasible paths or determine, if possible, the equivalence of the live mutants. This can only be performed in an exhaustive way requiring a possibly infinite number of paths–test cases to be generated. This fact also indicates the suitability of the kth-shortest paths heuristic in order to make the mutation testing criterion applicable. The results obtained and the nature of the graph seem to suggest that it may not be cost-effective to generate more than approximately 50 candidate paths per mutant constraint in an attempt to raise the mutation score. This result is along the same lines with the results derived in previous studies using the k-paths strategy in the context of structural testing [14]. However, the use of the best k-value is a matter of the undertaken policy and in view of this a high value could be perhaps advisable if critical applications were to be considered or if the cost-effect is not an issue.

The reason for considering such a testing strategy according to a specific k-value is twofold. First, it places practical bounds on the effort associated with the employment of the strategy, note that the actual number of paths in most program units tends to be infinite. Second, the equivalent mutants do result in infeasible paths when attempting to kill them. Such being the case, any attempt to generate paths to kill them, emerges the problem of an infinite number of infeasible paths (in the worst case and if the structure of the program supports it, e.g. loops). By employing such a stopping rule, this result in a practical heuristic that overcomes the equivalent mutant problem based on the effort investment per each mutant.

In order to substantiate the benefits of the proposed method, it is needed to establish the effort per each mutant considered. This will provide a useful yardstick for estimating the overall test data generation procedure effort. For this reason, the average values of the numbers of paths needed to be generated per mutant are presented in Table 4; together with the mean values of the actual and relative coverages achieved as well as the number of paths generated.

It must be noted here that the presence of equivalent mutants and infeasible paths not only does affect the practical matters of employing the mutation testing criterion, but also the theoretical ability to accurately predict the number of test cases needed, and therefore the effort too, on the following grounds. The absence of both equivalent mutants and infeasible paths would require in the worst case the generation of at most 4517 paths (number of mutants) if a distinct path were needed per mutant. In the best case as the problem can be converted into that of covering the arcs of the enhanced CFG, a method such as the one presented in [52] can provide a minimum number of paths needed. Thus, providing

Table 4
Total and per mutant paths towards coverage.

k-Value	Mean value of MS. actual coverage (%)	Mean value of MS^*. relative coverage (%)	Number of paths generated	Number of paths per mutant
1	60.62	83.35	2900	0.642019
10	64.93	89.28	20,319	4.49834
30	67.72	93.12	53,991	11.95284
50	68.74	94.52	85,687	18.96989
100	68.74	94.52	163,864	36.27718
300	70.78	97.32	452,869	100.2588
1000	71.93	98.90	1,336,153	295.8054
10,000	72.55	99.76	12,061,983	2670.353
30,000	72.64	99.88	35,017,392	7752.356
50,000	72.73	100.00	57,657,457	12764.55

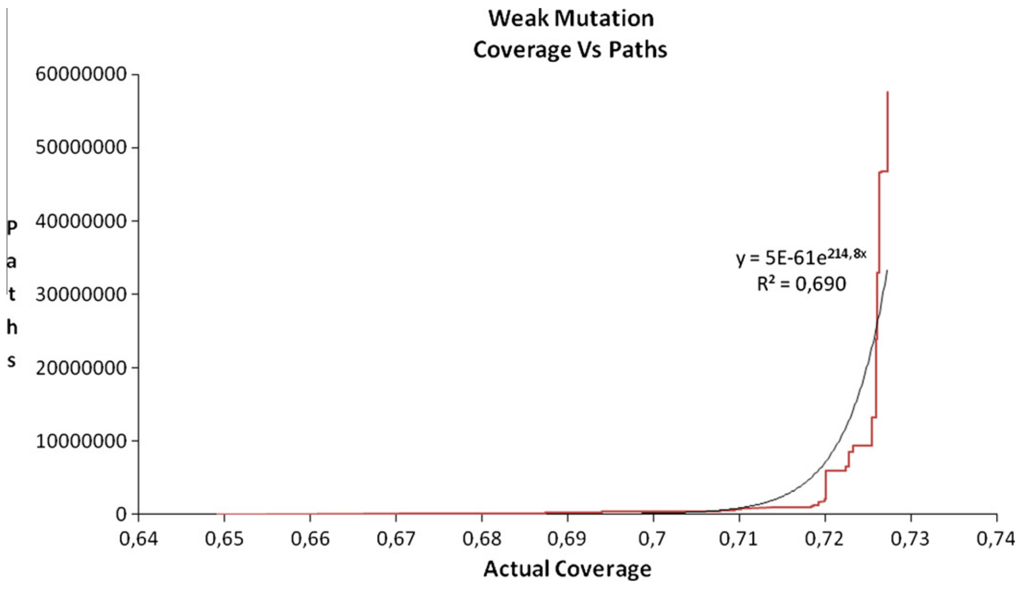

Fig. 6. Coverage per total paths.

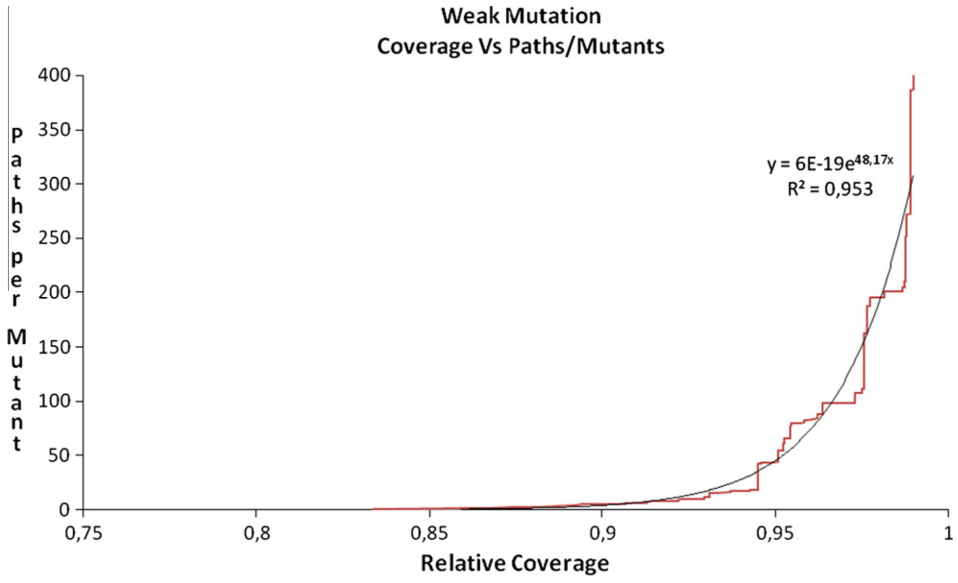

Fig. 7. Coverage per paths/mutants.

both upper and lower bounds for the total effort needed to kill the mutants.

Unfortunately, in practice the number of the required paths is substantially greater due to both equivalent mutants and infeasible paths making full coverage unachievable and the theoretical bounds inactive. The number of mutants can be easily calculated and as it can be seen in Table 4., a minimum of 2900 paths were needed initially to be generated in total, i.e. 2.900/4.517 = 0.64 paths per mutant. This value is between the two bounds referred to above and this is due to the presence of both equivalent mutants and infeasible paths.

Examining the values of Table 4., it can be noted that an actual coverage of approximately 60.62% and a relative one of 83.35% were obtained at the entry level for $k = 1$. In order to increase these coverages to 67.72% and 93.12% (k-value 30) respectively, an effort of approximately 18.6 times the initial one, is needed. The effort, in fact, increases markedly, at an exponential rate as it can be perceived by the graph of Fig. 6. Fig. 6 presents a plot of the number of paths generated against the actual coverage C_k. Fig. 7 presents the first part of (up to k-value 1400) of the relative coverage C_k^* achieved versus the number of paths generated per mutant.

The noticeable effects for both Figs. 5 and 6, with respect to the rate of the effort involved is that they both follow an exponential model of the form:

$$y = ae^{bx}, \text{ where } a \text{ and } b \text{ are parameters of the model that depend on the data.}$$

In the case of Fig. 6, the $R^2 = 0.6907$ provides a weak fit to the data. This can be explained by the large number of infeasible paths encountered. However, the plot does undoubtedly indicate an exponential trend. In the case of Fig. 7, the $R^2 = 0.953$ does indicate a good fit to the data providing values for $a = 6\text{E}{-}19$ and for $b = 48.174$. Although this is specific to the results obtained by using the 30 units, it does provide a basis that suggests the exponential relation between the number of paths/mutant and the coverage achieved.

From the graph of Fig. 7 and Table 4., as can be observed, the 19 paths (attained for $k = 50$) generated per mutant lead to a coverage of almost 95% of the mutants, while doubling the number of paths to 36 per mutant does not provide any increase at all, therefore it can be argued that it is not cost-effective to generate more than about 19 paths per live mutant in order to achieve a descent mutation coverage level. Also, the 95% of the relative coverage achieved, can be interpreted as the 95% of the achievable coverage when all 50,000 are considered. Thus, the $k = 50$ value is adequate in assessing the 95% of what would be achieved by employing all the $k = 50{,}000$ value. This can be also recorded as the strength of the strategy proposed.

It is worth mentioning that the experimental results are consistent with the results obtained in previous studies conducted for structural testing [14]. Both the present and previous studies [14] were founded on the same assertion about the feasibility of program paths, applied to totally different test samples however resulting in similar conclusions concerning not only the testing effort trend, but also the cost effective k-value point for achieving a high level of coverage.

6. Discussion of the results

This paper considers the practical problems encountered when employing the weak mutation testing criterion and eventually suggest using a path selection strategy to tackle them. An experimental assessment on the effort needed towards an effective path selection strategy was investigated and results were derived based on a purposely built tool. The results were derived from a sample of 30 programs, containing a total of 4517 mutants, by investigating approximately over 55 million program paths. The experiment contribution relies on the practical aspects of employing mutation, i.e. the path generation method used, its reflected effort and practical approaches to circumvent the equivalent mutants problem.

6.1. Effectiveness of the path generation method

In order to generally adopt the practical issues established by the experiment, some points should be addressed. Two factors must be considered when setting up such an experiment. These are: the adopted path selection strategy and the handling of the equivalent mutants. These factors are outlined in the following questions: "*how typical is the effectiveness of the path selection strategy?*" and "*in case of employing an equivalent mutant detection system that could eliminate all or most of the equivalent mutants are the results still sound?*".

Concerning the effectiveness of the employed path generation strategy, it is again essential to consider the fundamental assumption of the method used. Basically, as the number of infeasible paths in a program tends to be rather significant, any path generation method should consider their presence upon their generation. Based on the validity of this fundamental assumption the method adopted in this study is perhaps the only method that suggests a methodical way of generating the paths by trying to avoid the infeasible ones. Therefore, the proposed method establishes a clear advantage against any other possible method that ignores the presence of infeasible paths. Consequently, it should be anticipated that adopting the proposed strategy will be more efficient than its alternatives.

In view of the difficulties imposed by the influence of the various equivalent mutants on the soundness of the results reported in this paper, by focusing on the results reported in Section 5., even in the hypothetical absence of equivalent mutants, there should still necessitate a high number of iterations cycles (the k-value) in order to adequately achieve the required levels of coverage. It is noted that the k-value is completely independent of the presence of equivalent mutants. Thus, the influence of infeasible paths on the overall effort is substantial even in such a hypothetical case. Indeed, a k-value even higher than 40,396 (where full relative coverage was achieved) should be employed which in any case is prohibitively high. Despite all these considerations, the present paper targets on practical aspects of employing weak mutation testing and thus it is suggested that the remaining live mutants after k tries to be treated as equivalent ones. Although this requires settling with less than full coverage, the testing thoroughness will not be greatly affected.

Following the suggestions made by Offutt and Untch [5] and Yates and Malevris [7] the application of a coverage criterion must be cost-effective. "Software testing is an imperfect science and there is no reason for coverage to be exact" as advocated in [5]. Additionally, based on the present experiment and on the above arguments, the proposed method will on average outperform the other path selection strategies. Consequently, the extent of the effort (as measured by the number of generated paths per mutant) to achieve the various levels of coverage for the sample used, is likely to be the minimum possible that can be achieved. The exponential behaviour of the effort has been shown to be in direct relation with the number of infeasible paths. This would only be alleviated if there existed a method that could avoid and ignore the presence of infeasible paths. As such a method does not exist, all methods considered will be subjected to exponential effort, with perhaps the one adopted here which tries to reduce the incidence of infeasible paths thus, diminishing the effort involved.

6.2. Issues about weak, strong and higher order mutation

This paper considers the employment of weak mutation as a testing criterion in order to produce high quality test cases. The proposed approach relies on the foundations of path feasibility as proposed by Yates and Malevris [14]. Generally, the path selection strategy proposed here can be extrapolated to include other mutation based testing criteria such as strong mutation or higher order mutation. Although, such an approach should require more effort due to the increased length and complexity of the selected paths, this effort is substantial to the increased strengths of the higher order and strong (first order) mutation. The question that it is discussed here is how can these approaches be tackled by possible extensions of the proposed method?

Generally, in order to include strong mutation there is a need to handle the sufficiency condition [21]. This is a very difficult task [21] and could be achieved by generalising the considered conditions, utilising the same template " $e \mathrel{!}= e'$ " for the data state variables that are influenced by the introduced mutant. Thus, by requiring the various introduced branches to be true for every node of the program execution path that is affected by the introduced mutant can lead to its propagation (fulfil the sufficiency condition). Finding such a feasible path will probably result in test cases that could effectively kill the introduced mutants. A similar approach has been proposed by Santelices et al. [53] in the context of regression testing for testing the introduced code changes. In their approach a set of paths is selected based on dependence analysis. Santelices et al. [53] used partial symbolic execution on the selected paths in order to identify probably feasible testing requirements that are capable of propagating the introduced program changes. The same approach has also been proposed for handling multiple changes in the code under test. This approach can be easily adopted in order to tackle the mutant killing problem for higher order mutants.

6.3. The influence of infeasible paths on the required effort

Path analysis forms a well established method with many applications on all aspects of software engineering activities including various forms such as testing, program analysis [48] and program profiling [54]. All these activities are mainly influenced by the existence of infeasible paths. This should be obvious for all the methods that select a set of paths prior to their analysis, as the probability of selecting an infeasible one is quite high. Specifically, in our case where the aim is to kill a hard to kill mutant, the path selection method must select many alternative paths that reach the targeted mutant. Every such candidate path must be checked for its feasibility which is a very costly activity. If that path is found to be infeasible, the effort spent on checking the path' feasibility is wasted. Thus, it is natural to expect that the higher the number of the encountered infeasible paths the higher the amount of the wasted effort is. Based on the conducted experiment, when aiming at killing the mutants of unit No. 1 with a maximum number of tries being 1000 paths per live mutant, a number of 23,775 paths were selected in total. From these paths, only nine where found to be feasible thus, resulting in 23,766 useless checks for their feasibility. Put it in another way, consider the required adopted *k*-value for killing the targeted mutants. It must be recalled that the *k*-value represents the maximum number of the selected paths per live mutant. Thus, in our experiment one and only one ‘‘hard to kill’’ mutant required 40,396 paths to be considered in order to be killed. This mutant provides a good example that shows the impact of the infeasible program paths on the test generation effort.

6.4. Scalability issues and limitations

The automatic generation of test cases constitutes one of the most difficult problems encountered in the software testing activity. To address this problem many methods have been proposed by the research community. The present paper considers a path based approach. The main challenge faced in this area is the scalability of these methods due to the encountered infeasible paths as discussed in Section 6.3. The presently proposed approach goes a step forward by considering paths in a systematic way with the aim of avoiding the infeasible ones (see Section 6.1 for a discussion on this matter). Therefore, the proposed method establishes a clear advantage against any other possible method that ignores the presence of infeasible paths. Consequently, it should be anticipated that adopting the proposed strategy will be more effective at selecting feasible paths than its path selection alternatives. However, scalability issues still remain and require further research. These issues are attributed to the following two factors: the use of pure static symbolic execution and the use of mutation testing.

Static symbolic execution has many limitations [48] such as the handling of function calls, calls to external libraries, solving of complex non-linear expressions etc which withhold its practical application. However, emergent research in this area [55,13,49,48] give answers to those problems making symbolic execution feasible and scalable. Here, it must be mentioned that since the proposed approach relies on path selection, it provides an independent to the symbolic execution way of producing the sought test cases. Generally, the efficient and effective determination of the feasibility of the selected paths is independent to their selection. Thus, no matter how the symbolic execution engine works the proposed approach guides the selection of paths for killing mutants. These paths must definitely be explored in order to kill the aimed mutants.

Automatically, producing tests for mutation testing forms a difficult task [5]. Recall from Section 2.4.3 that mutation-based test cases must adhere to the reachability, necessity and sufficiency conditions. A significant amount of resources is needed by any test generation method in order to jointly fulfil these conditions. Further, the incidence of equivalent mutants (see Section 6.1 for a discussion on this matter) considerably impacts the cost of such a method. This problem can be tackled by using various mutation testing alternatives [5,18] on the one hand and by using effective test generation approaches as discussed above on the other.

The approach proposed in the present paper suggests the use of program paths in order to effectively kill mutants. Although, it uses symbolic execution in order to examine the feasibility of the selected paths it bases its characteristics for alleviating the problem of infeasibility on the selection of paths rather than their symbolic evaluation, which might be performed in different ways. Therefore, by providing an effective heuristic, such as the one proposed in this paper, to deal with the undesirable effects caused by the incidence of infeasible paths substantial benefits can be gained.

6.5. Threats to validity

The present paper focuses on effectively automating the test case generation using a path selection strategy for mutation testing. One possible threat to the validity of the obtained results is due to the utilised mutants. Thus, the strategy's effectiveness may vary when considering other mutant operators. Although this may hold, the utilised operators form the current practice in the various experimental studies found in the literature. There are many experimental studies concerning mutants' effectiveness and the majority adopts a similar to the present study, well established set of operators [32,18] not only in theory but also in practice. Although such a set has not been widely used for weak mutation, Offutt and Untch [5] suggest that it should be considered. Additionally, this set was formed as a result of an experimental study [32] in which test sets were derived with reference to the Godzilla tool, which generates tests based on weak mutation. In any case, the addition or even the elimination of some mutants should not alter the overall approach employment pattern or effort assessment, derived by the present study, as their presence will only add or remove similar simple additional constraints. Furthermore, the underlying path selection method, makes use of a fundamental assertion detailed in Section 3, that takes into consideration when choosing the paths (aiming at feasible ones), only the total number of constraints contained in a path and not the nature of those constraints. This can be regarded as the method's ability to generate effective results irrespective of the mutants considered.

Another possible threat is due to the generalisation of the utilised sample. This is true for all the experimental studies. Therefore, it cannot be claimed that the selected units represent a generalisation of the results produced in this study. Further, considering the units' size another issue also appears, that is the application on larger program units. However, the method does not use the special features and specifications of the units, making its application rather indifferent to the specifics of the units. Moreover, typical program units are not of a considerably large size. Thus, the selection of another set of programs should not affect the application of the proposed method. Nonetheless, consider Fig. 8 which is a histogram of depicting the proportion of feasible *k*th shortest paths for the sample units. From the histogram it can be seen that as *k* increases, the proportion of feasible paths tends to decrease, therefore the proportion of infeasible ones increases reciprocally. This is in accordance with the proposition made in Section 3, and the statistical evidence provided in [14] that path feasibility tends to decrease with path predicate involvement. Under this assumption then, the nature of Fig. 8 supports the belief that the sample used in the experiment has a representative behaviour with the independent sets used in the previous studies such as in [14]. These results are also verified by Vergilio et al. [56] study. In their study the authors repeated an experiment involving the number of predicates and feasibility, and they concluded the same findings.

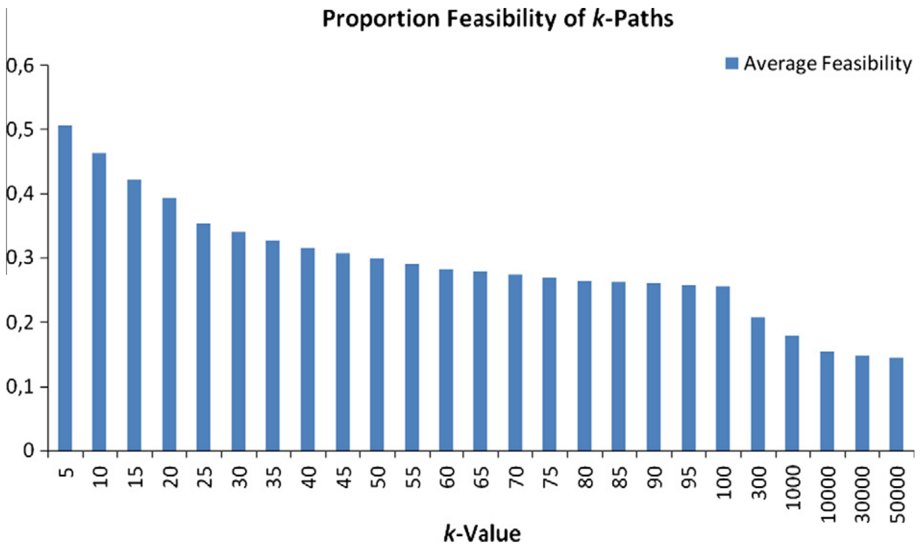

Fig. 8. Proportion of feasible *k*th shortest paths.

Finally, another possible threat to the validity of the present experiment maybe based on the use of software systems. Thus, possible bugs, program conversions and differences of the test subjects from their language to the SymExLan one may have also affected the obtained results.

Despite all the abovementioned issues, the main objective of the present paper was to present the strategy's feasibility and applicability to a set of programs when performing mutation, rather than comparing the effectiveness of the proposed approach with various rival ones. In view of this, the proposed method, does give an answer to the general problem of generating test data able to kill mutants and it does so in an effective way. Further, the obtained results also suggest practical guidelines for the production of high quality test cases with reasonable resources by using practical heuristics for circumventing the equivalent mutants' problem.

7. Conclusion

This paper addresses application issues of the weak mutation testing method, and in particular, the extent of the effort that in practice is entailed by employing it. To assess the effort involved, an experiment with 30 unit programs was conducted that uses an effective path selection strategy that alleviates in an a priori basis the unnecessary effort spent on infeasible paths. The sample of the 30 units is formed from the ones used in previous studies such as in [21,32,6,43]. The method employed in the present work, has been proved to be effective in an attempt to diminish the effort involved in mutation testing by avoiding the presence of infeasible paths. This is achieved by transforming the mutation coverage problem into a node coverage one by introducing the Enhanced CFG. Covering all nodes in this graph is equivalent to covering all inserted mutants. Based on this, all equivalent mutants signify infeasible nodes in the Enhanced CFG. Thus, in view of the above, path based methods can be utilised to support mutation testing.

As discussed in previous sections the presence of infeasible paths and equivalent mutants, results in a waste of effort that gradually dominates the entire process in terms of the total effort spent. The results obtained by analysing the 30 units with the prototype tool developed, showed a mostly acceptable level of mutation coverage. For 6 of the 30 units their mutants were covered completely, and a mean relative mutant coverage of 83.35% was achieved by generating, on average, a quite modest number of 0.64 paths per mutant element. The respective actual mutant coverage value achieved for the same effort investment was 60.62%. Raising the coverage to an acceptable level of 94.52% for the relative mutant coverage was also possible with a reasonable amount of additional effort. Thus a 68.74% and 94.52% of actual and relative mutant coverages were obtained with a mean effort of approximately 19 paths per mutant element. Beyond that point the trade off between the effort and coverage starts to become unprofitable. Nevertheless, in general, the trade relation between coverage and effort is exponential. For small *k*-values the rate of effort is almost non exponential rising exponentially, after the $k = 50$ value. For 19 of the 30 units full mutant relative coverage was achieved and a mean value of 97.32% attained, by generating a total of 452,869 paths which is translated to 100 paths per mutant element. Full mutant relative coverage was accomplished by a mean effort of 10,358 per mutant element. It is noted that the remaining live mutants, which are treated as equivalent ones, may not in fact be equivalent. This is an assumption made for the purposes of the experiment as it is unlikely to kill any mutant beyond the threshold of 50,000 paths. Past this point all mutants were treated as equivalent. This raises the matter of infeasible paths as mutant equivalence and path infeasibility are directly related. In any case, the achievement of a mutant relative coverage at any stage and for any $k < 50{,}000$, predefines the final actual mutant coverage attainable at the 50,000 point as the ratio $MS(C, D)/MS^*(C, D)$ at *k*. Evidently, not having the precise measures of relative coverage the results indicate that a very reasonable level of weak mutation coverage can be achieved with quite an acceptable expenditure of effort. This fact attests to the effectiveness of the proposed method's ability to reduce the incidence of infeasible paths.

The empirical results presented here indicate the influence of infeasible paths and of infeasible test elements, such as equivalent mutants, in applying testing criteria in practice. For all these arguments it is suggested, and this constitutes the major conclusion of the present work, to generate up to 19 paths per mutant (achieved for $k = 50$) as the additional paths beyond this value increase the effort rate dramatically.

Acknowledgments

This research was funded by the Research Centre of the Athens University of Economics and Business. The authors would like to thank Dr. D. F. Yates for his valuable comments.

References

[1] R.G. Hamlet, Testing programs with the aid of a compiler, IEEE Trans. Softw. Eng. 3 (1977) 279–290.
[2] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, Computer 11 (1978) 34–41.
[3] A.J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evaluation of data flow and mutation testing, Softw. Pract. Exper. 26 (1996) 165–176.
[4] N. Li, U. Praphamontripong, A.J. Offutt, An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage, in: Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09), Denver, Colorado, 2009, pp. 220–229.
[5] A.J. Offutt, R.H. Untch, Mutation 2000: uniting the orthogonal, in: Mutation Testing for the New Century, Kluwer Academic Publishers, 2001, pp. 34–44.
[6] N. Malevris, D.F. Yates, The collateral coverage of data flow criteria when branch testing, Inf. Softw. Technol. 48 (2006) 676–686.
[7] D.F. Yates, N. Malevris, An objective comparison of the cost effectiveness of three testing methods, Inf. Softw. Technol. 49 (2007) 1045–1060.
[8] W.E. Howden, Weak mutation testing and completeness of test sets, IEEE Trans. Softw. Eng. 8 (1982) 371–379.
[9] M.R. Woodward, M.A. Hennell, On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC, Inf. Softw. Technol. 48 (2006) 433–440.
[10] R.S. Boyer, B. Elspas, K.N. Levitt, SELECT a formal system for testing and debugging programs by symbolic execution, SIGPLAN Not. 10 (1975) 234–245.
[11] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (1976) 385–394.
[12] P. McMinn, Search-based software test data generation: a survey, Softw. Test. Verif. Reliab. 14 (2004) 105–156.
[13] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, T. Vos, Symbolic search-based testing, in: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp. 53–62.
[14] D. Yates, N. Malevris, Reducing the effects of infeasible paths in branch testing, in: Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, ACM, Key West, FL, USA, 1989, pp. 48–54.
[15] M. Papadakis, N. Malevris, An effective path selection strategy for mutation testing, in: Proceedings of the 16th Asia–Pacific Software Engineering Conference, IEEE Computer Society, 2009, pp. 422–429.
[16] Y. Jia, M. Harman, An Analysis and Survey of the Development of Mutation Testing, Technical, Report, 2010.
[17] Y. Jia, M. Harman, Higher order mutation testing, Inf. Softw. Technol. 51 (2009) 1379–1393.
[18] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 99 (2010).
[19] A.J. Offutt, S.D. Lee, An empirical evaluation of weak mutation, IEEE Trans. Softw. Eng. 20 (1994) 337–344.
[20] M.R. Woodward, K. Halewood, From weak to strong, dead or alive? An analysis of some mutation testing issues, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988, pp. 152–158.
[21] R.A. DeMillo, A.J. Offutt, Constraint-based automatic test data generation, IEEE Trans. Softw. Eng. 17 (1991) 900–910.
[22] H.N. Gabow, S.N. Maheshwari, L.J. Osterweil, On two problems in the generation of program test paths, IEEE Trans. Softw. Eng. 2 (1976) 227–231.
[23] T.A. Budd, D. Angluin, Two notions of correctness and their relation to testing, Acta Inform. 18 (1982) 31–45.
[24] A.J. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths, Softw. Test. Verif. Reliab. 7 (1997) 165–192.
[25] R.M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Softw. Test. Verif. Reliab. 9 (1999) 233–262.
[26] M. Papadakis, N. Malevris, Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing, Softw. Qual. J. 19 (2011) 691–723.
[27] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, IEEE Trans. Softw. Eng. (2011). 1–1.
[28] M. Harman, Y. Jia, W.B. Langdon, Strong higher order mutation-based test data generation, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, Szeged, Hungary, 2011, pp. 212–222.
[29] M. Papadakis, N. Malevris, Automatic mutation test case generation via dynamic symbolic execution, in: IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), 2010, pp. 121–130.
[30] Y.-S. Ma, J. Offutt, Y.R. Kwon, MuJava: an automated class mutation system, Softw. Test. Verif. Reliab. 15 (2005) 97–133.
[31] K.N. King, A.J. Offutt, A Fortran language system for mutation-based software testing, Softw. Pract. Exper. 21 (1991) 685–718.
[32] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Trans. Softw. Eng. Methodol. 5 (1996) 99–118.
[33] R.H. Untch, A.J. Offutt, M.J. Harrold, Mutation analysis using mutant schemata, in: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, ACM, Cambridge, MA, USA, 1993, pp. 139–148.
[34] R.A. DeMillo, E.W. Krauser, A.P. Mathur, Compiler-integrated program mutation, in: Proceedings of the 5th Annual Computer Software and Applications Conference (COMPSAC'91), Tokyo, Japan, 1991, pp. 351–356.
[35] A.J. Offutt, Z. Jin, J. Pan, The dynamic domain reduction procedure for test data generation, Softw. Pract. Exper. 29 (1999) 167–193.
[36] K. Ayari, S. Bouktif, G. Antoniol, Automatic mutation test input data generation via ant colony, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, ACM, London, England, 2007, pp. 1074–1081.
[37] M. Papadakis, N. Malevris, An empirical evaluation of the first and second order mutation testing strategies, in: Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010, pp. 90–99.
[38] E.J. Weyuker, More experience with data flow testing, IEEE Trans. Softw. Eng. 19 (1993) 912–919.
[39] D.F. Yates, M.A. Hennell, An approach to branch testing, in: 11th International Workshop on Graph Theoretic Techniques in Computer Science, Wurtzburg, 1985.
[40] lp_solve, in. Available from: <http://sourceforge.net/projects/lpsolve/, accessed 13/02/2012>.
[41] C. Koutsikas, N. Malevris, A new script language applicable to symbolic execution systems, Int. J. Comput. Appl. 28 (2006) 1–11.
[42] C. Koutsikas, N. Malevris, A unified symbolic execution system, in: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, IEEE Computer Society, 2001, pp. 466.
[43] M. Papadakis, N. Malevris, A symbolic execution tool based on the elimination of infeasible paths, in: Proceedings of the 2010 Fifth International Conference on Software Engineering Advances, IEEE Computer Society, 2010, pp. 435–440.
[44] J. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications, Softw. Test. Verif. Reliab. 13 (2003) 25–53.
[45] M. Papadakis, Methods for Detecting Errors in Java Programs with the Use of Mutation Testing Method, MSc thesis, Department of Informatics, Athens University of Economics and Business, Athens, Greece, June 2005 (in Greek).
[46] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, A.J. Offutt, An extended overview of the Mothra software testing environment, in: Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88), Banff Alberta, Canada, 1988, pp. 142–151.
[47] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem, P. Tonella, Automatic unit test data generation using mixed-integer linear programming and execution trees, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, 1999, pp. 189.
[48] C. Păsăreanu, W. Visser, A survey of new trends in symbolic execution for software testing and analysis, Int. J. Softw. Tools Technol. Transfer (STTT) 11 (2009) 339–353.
[49] C. Cadar, P. Godefroid, S. Khurshid, C.S. Păsăreanu, K. Sen, N. Tillmann, W. Visser, Symbolic execution for software testing in practice. preliminary assessment, in: Proceeding of the 33rd International Conference on Software Engineering, ACM, Waikiki, Honolulu, HI, USA, 2011, pp. 1066–1071.
[50] A. Kiezun, V. Ganesh, P.J. Guo, P. Hooimeijer, M.D. Ernst, HAMPI: a solver for string constraints, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ACM, Chicago, IL, USA, 2009, pp. 105–116.
[51] N. Kosmatov, All-paths test generation for programs with internal aliases, in: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering, IEEE Computer Society, 2008, pp. 147–156.
[52] S.C. Ntafos, S.L. Hakimi, On path cover problems in digraphs and applications to program testing, IEEE Trans. Softw. Eng. SE-5 (1979) 520–529.
[53] R. Santelices, P.K. Chittimalli, T. Apiwattanapong, A. Orso, M.J. Harrold, Test-suite augmentation for evolving software, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2008, pp. 218–227.
[54] T. Ball, J.R. Larus, Efficient path profiling, in: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, IEEE Computer Society, Paris, France, 1996, pp. 46–57.
[55] C.S. Păsăreanu, N. Rungta, W. Visser, Symbolic execution with mixed concrete-symbolic solving, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ACM, Toronto, Ontario, Canada, 2011, pp. 34–44.
[56] S.R. Vergilio, J.C. Maldonado, M. Jino, Infeasible paths in the context of data flow based testing criteria: identification, classification and prediction, J. Braz. Comp. Soc. 12 (2006) 71–86.