An Effective Path Selection Strategy for Mutation Testing

Mike Papadakis and Nicos Malevris
Department of Informatics, Athens University of E.B.
Athens, Greece
{mpapad, ngm}@aueb.gr

Abstract— **Mutation testing has been identified as one of the most effective techniques, in detecting faults. However, because of the large number of test elements that it introduces, it is regarded as rather expensive for practical use. Therefore, there is a need for testing strategies that will alleviate this drawback by selecting effective test data that will make the technique more practical. Such a strategy based on path selection is reported in this paper.**

A significant influence on the efficiency associated with path selection strategies is the number of test paths that must be generated in order to achieve a specified level of coverage, and it is determined by the number of paths that are found to be feasible. Specifically, a path selection strategy is proposed that aims at reducing the effects of infeasible paths and conversely developing effective and efficient mutation based tests. The results obtained from applying the method to a set of program units are reported and analysed presenting the flexibility, feasibility and practicality of the proposed approach.

Keywords: software testing, unit testing, mutation testing, path testing, adequacy criterion, infeasible paths

I. Introduction

It is a well known fact that the cost of software testing can reach 50% or even 60% of the total software development cost. In order to reduce the cost overhead, a lot of effort has been put by the software engineering community, in an attempt to automate the testing activities and thus reduce the overall software development cost. Usually, the test data generation activity is performed by a tool or tools which partially or fully automate(s) the production of the sought test data. The generated data must then be driven to the software which is then checked in terms of its behaviour and validity with respect to the produced results. The usual way to evaluate the test thoroughness of a piece of software is to establish a collection of requirements that must be fulfilled when the software is executed against test cases. Such a set of requirements can be certain test coverage criteria for example, that guide and evaluate the effectiveness of the test data generated. Normally, the criteria selected to be fulfilled are structural, derived by a hierarchical structure which includes all the white box criteria such as: statements, branches, data flow etc.

Mutation testing is a powerful fault-based testing technique introduced by Hamlet [1] and the DeMillo, Lipton and Sayward [2] and forms the focus of this paper. Mutation analysis induces syntactical alterations of the code under test with the aim of producing mutant versions of the considered code. Each program version is called mutated version and contains one simple syntactic change from the original code. Test cases are used to execute these mutated versions with the goal of distinguishing them from the original one. A mutant is said to be killed if there is a test that distinguishes its output from the output of the original program whereas, it is said to be equivalent if there are not such distinguishing inputs. Assessing tests with the killing mutants' ratio is usually considered as a measure of the quality of the testing thoroughness.

Finding appropriate test data according to a selected criterion can be very laborious [3] and fully or partially automating this activity forms one of the major problems that interest the software engineering research community. Unfortunately, this argument holds for mutation too. Most of the progress in the area has been reported by DeMillo and Offutt [4] in a technique called Constraint Based Testing (CBT) and also by Offutt, Jin and Pan [5] in the Dynamic Domain Reduction approach (DDR). Both attempts rely on the use of paths and symbolic evaluation in order to construct sets of conditions under which inputs should execute and infect the program state of the considered mutant programs. These approaches differ in the way they solve the constraint sets and how they handle specific program elements such as loops, arrays and complex expressions. Although experiments have shown that both CBT and DDR are effective, i.e. more than 90% of mutants were killed in experiments based on either CBT or DDR approaches [4], [5], issues concerning their efficiency and effectiveness on considerable size programs, still remain.

Generating tests according to mutation can be a big burden because of the high number of the considered mutants. Despite the progress recorded in reducing that number [6] the problem remains mostly unresolved and it is estimated to be proportional to the product of the number of data references and the number of data objects [6] of the program under test. Additionally, proportional to the number of candidate mutants, are also the time-consuming stages of: mutation analysis, mutant generation and execution phase. Thus, in order to reduce the overall effort spent during mutation testing there is a need to generate a small number of tests that kills most of the considered mutants at first and then concentrate on a small number of the remaining live mutants. The proposed strategy begins by producing a generic set of tests that kill a large proportion of mutants and then continues on targeting incrementally to each of the remaining live ones. By doing so, the effort spent on mutation analysis and on trying to generate tests for equivalent and hard to kill mutants is reduced.

The proposed approach reduces the killing mutant's problem to a covering branches problem. Treating each mutant as a branch helps on focusing on specific mutants and selecting candidate paths in order to generate data aiming at killing the specified mutants. By using path analysis the strategy tries to reduce the incidence of infeasible paths on the one hand, and to utilise most of the already selected feasible paths and thus reduce and balance the overall effort to kill each mutant on the other.

The essence of our work can be summarised as follows: Efficiently select a proper subset of all paths as a candidate solution to the problem of generating mutation adequate test data. This task can be quite difficult and impractical if done in an ad hoc way. Heuristic approaches therefore that guide the selection of candidate paths and result in feasible sets are needed in order to practically employ mutation testing. Therefore, the proposed strategy will anticipate savings in time and effort required by both the test data generation and mutation analysis activities.

The rest of this paper is organised as follows: Section II introduces some background material. Section III outlines the path selection strategy and section IV details the proposed approach. Section V reports on the results obtained through the application of the proposed method. Sections VI and VII discuss some related to the present work and discuss its practical use together with some future directions. Finally in section VIII conclusions are given.

II. Background

Testing criteria have the goal of selecting a subset of all possible test cases that have a high probability of detecting errors. There are many types of testing requirements which can be formed according to various techniques such as functional, structural and fault-based. In general, structural testing criteria require the examination of the internal composition of the program's source code. Tests are derived from examining certain program elements such as basic blocks, branches, paths etc., of the program under test. The criteria requirements goal is both to guide and evaluate the quality of the test sets. Typically tests are produced until a predefined level of coverage is reached. The level of coverage according to a selected criterion is defined as the ratio of its test elements being covered to the sum of the elements introduced. In branch testing for example, a path set consisting of feasible paths that cover all the program branches is chosen and tests that execute the selected paths are produced. As pointed in [7], [8], [9] the incidence of infeasible paths is responsible for the major part of the testing effort. Ideally, all testing strategies should increase their efficiency by selecting candidate sets with a minimum number of infeasible paths.

Testing based on fault based criteria, requires the exposition of some introduced faults. These criteria seed a number of faults into the program's code and utilise the requirements for exposing them. The test coverage is defined according to the number of faults found by the test set to the number of faults introduced. Mutation testing is a fault based testing technique that introduces faults by making simple syntactic changes to the source code under test. The syntactic changes introduced are based on a set of rules called mutant operators. According to the literature, a seeded fault such as a mutant, in order to be exposed should satisfy three conditions known as Reachability, Necessity and Sufficiency [4], [10]. Based on these three conditions DeMillo and Offutt developed a test data generation technique called Constraint-Based Test data generation (CBT) [4].

The Reachability condition states that the mutant statement must be exercised with test data. It must be noted that mutation introduces one fault at a time and all the program's executable statements apart from the mutated one are the same to the original thus, the execution paths form the same execution computations for both the original and the mutated program versions. If tests can not execute the mutated statement, it is guaranteed that the tests have no chance of killing the seeded mutant [4]. The Necessity condition states that the execution of the mutated statement must cause a discrepancy of the original program state [4]. In other words the outcome of the execution of both the original and mutated statements must differ. Otherwise because of the syntactical equality of the rest of the two program versions they will never form different computations and never result in observable output differences. The sufficiency condition states that the infected program state must propagate up to the last program statement. Execution path and its computations must use the internal different value resulted at the mutated statement (necessity condition) and form a different observable computation from there onwards to the programs output.

Current test data generation approaches [4], [5] try to utilise directly the reachability and necessity conditions. The Sufficiency condition because of its high complexity is satisfied indirectly through the satisfaction of the reachability and necessity conditions. As found in [4], [11] tests that meet the reachability and necessity conditions have a 90% chance of meeting the sufficiency condition too. The proposed approach focuses on a direct handling of the first two conditions. The third condition (sufficiency) is handled indirectly through the fulfilment of the first two conditions and when needed through the selection of alternative paths. The proposed approach introduces an improved strategy over the CBT method by posing practical heuristics and limits on the test generation methods. The necessity of employing a heuristic approach stems from the need to reduce the overall testing effort especially when using a high demanding technique, such as mutation. Time and effort limits are thus necessary to practically perform such a testing activity.

III. Path Selection Method

A. The extended shortest path method

The problem of obviating infeasible paths during the testing process was stated by Gabow, Maheshwari and Osterweil [12]. Predicting the infeasibility of a program path a priori therefore by any means, must necessarily be heuristic. In an attempt to derive such a heuristic, Yates and Hennell [13] 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 [7], wherein it was concluded, with great statistical significance, that the feasibility of a path decays exponentially with the increasing number of predicates it involves. As a result, Yates and Malevris [7] proposed a path selection method, extending that of Yates and Hennell, 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 on a program path. Thus, the method does not seek to optimise, in any way, on any one testing criterion. Hence, it may be used with equal validity, and without bias, in an attempt to fulfil any other testing criterion. It is this path generation method that was adopted for the research reported in this paper.

The strategy for purposes herein can be detailed as:

```
1. Generate a set $\Pi$ of program paths - each containing a minimum number of predicates - that cover the test elements of code unit.
2. Derive the percentage value of Coverage resulting from the feasible paths in $\Pi$.

While Coverage<1 AND i<k,
repeat steps 3 and 4 once for each uncovered element e.
    3. Generate the next candidate path $\Pi_e^i$
    4. If path found to be feasible, then recalculate the value of Coverage.
```

Here, Π_e^i denotes the *i*th shortest path through element *e* and *k* = maximum number of practically viable paths (beyond which the generation of extra paths is prohibitive). The order of selecting the elements *e* at step 3 can be performed in various ways. Here, they were selected at random. The optimal use of these two parameters (the order of selecting test elements, and the use of appropriate *k* value) form a matter of future research.

B. The method used to derive Π_e^i for branch testing

Each program path in Π is a shortest path, in terms of the number of predicates it involves, through some branch of the program, say C. The problem of generating Π is thus that of generating a set of shortest paths through C which together cover all of C's branches. Moreover, if each arc ij of G_C, the Control Flow Graph (CFG) of C, is assigned a 'length' W_{ij} such that:

W_{ij} = the number of arcs emanating from vertex i - 1

The required path set can be identified as a set, Π^*, of shortest Start to Finish paths that covers the arcs of G_C.

As in [13], $\Pi_{xy}^{(1)}$, the shortest path from S to F in G_C that passes through arc xy can be decomposed using the principle of optimality as:

$$\Pi_{xy}^{(1)} = \Pi^{(1)}(S,x) \oplus xy \oplus \Pi^{(1)}(y,F) \tag{1}$$

Where $\Pi^{(1)}(p,q)$ denotes the shortest path from p to q for any pair of vertices p and q, and $\oplus$ denotes sequence in G_C.

In a similar fashion, Π_e^i, the *i*th shortest path in C through a branch, can be expressed as:

$$\Pi_{xy}^{(i)} = \Pi^{(r)}(S,x) \oplus xy \oplus \Pi^{(t)}(y,F) \tag{2}$$

Where $\Pi^{(j)}(p,q)$ denotes the *j*th shortest path from vertex p to vertex q of G_C, as is appropriate, and *r* and *t* are integers satisfying $1 \le r \le i$ and $1 \le t \le (i - r + 1)$, respectively, and again $\oplus$ denotes sequence.

IV. Approach Description

A. Overview

Our approach mainly relies on two observations about the testing effort needed to achieve adequate or close to adequate mutation coverage. The first observation points that there is a significant influence of the effort associated with mutation based test data generation and the number of test paths that must be generated in order to fulfil the reachability, necessity and sufficiency conditions of mutants as stated in section II. Thus, the major part of the test generation activity is associated with the number of the selected paths that are found to be infeasible. Effort, therefore, is depended upon the effectiveness of the path generation method/strategy in avoiding infeasible paths. Thus, any strategy which does not seek to reduce the incidence of infeasible paths is non-practical and leads to waste of effort proportional to the number of considerable mutants.

The second observation comes from the application of branch testing. Specifically, the branch coverage criterion fulfils the reachability condition of all mutants and the necessity and sufficiency conditions for the majority of them as presented in section V. Moreover, for the majority of the cases, there is no need to generate additional paths in order to achieve the combination fulfilment of reachability, necessity and sufficiency conditions. Therefore, the additional effort overhead to perform mutation when branch testing, can be considerably reduced.

Based on the above observations, we propose the use of an efficient path selection strategy in order to fulfil the branch coverage criterion first and then incrementally fulfil the necessity and sufficiency conditions of the remaining mutants. The use of such an incremental approach will reuse efficiently much of the information gained from the branch coverage – necessity condition fulfilment to the combinational fulfilment of the necessity and sufficiency conditions. The main idea of the proposed approach is to reduce the problem of fulfilment of the necessity conditions of mutants to that of covering "branches". This is done by transforming the required necessity "constraints" to branch predicates and representing them in the program control flow graph. The innovation of this approach is the use of an enhanced control flow graph and its respective branch predicates in order to proceed with the symbolic evaluation and test data generation phase. The benefit of doing so is the incorporation of mutation based criteria to existing path based test data generation methods inheriting all their merits.

The proposed approach aims at program branches first and consequently kills a larger number of the targeting mutants. Then the method tries to incrementally cover the remaining live mutants in turn. The reason for employing such an incremental strategy is twofold. First, the branch testing exercise is a well studied problem and forms almost a universal minimum requirement for testing. Its nature guides the test to a well formed subset of test requirements which normally includes limited infeasible elements. Second, the test generation process balances the effort between the various test elements. Thus, the test strategy can stop at any time resulting in a fairly balanced level of testing thoroughness, having exercised proportionally all program elements, at the respective time. The proposed approach tries to minimise the test generation overhead caused by the infeasible paths contained in the program and by avoiding trapping or spending too much time on equivalent mutants or too hard to kill mutants.

B. The method used

The representation of the structure of a code unit by its Control Flow Graph (CFG) is well known, understood and forms the basis of path testing for most of the white box testing criteria. Unfortunately, this is not so for mutation, since the graph structure does not represent or identify mutants. For mutation a more elegant model is needed. Additionally, mutation testing requires a complete execution path in both the original and the mutated program versions in order to cover-kill the concerned mutant. Following the literature suggestions about the weak mutation hypothesis [11] and by using mutants necessity conditions [4], it becomes possible to build a simple test model. By enhancing the programs' CFG with mutant related necessity constraints, a single graph representing-including the various mutants can be constructed. The enhanced graph is built simply by adding a special type of vertex for each mutant.

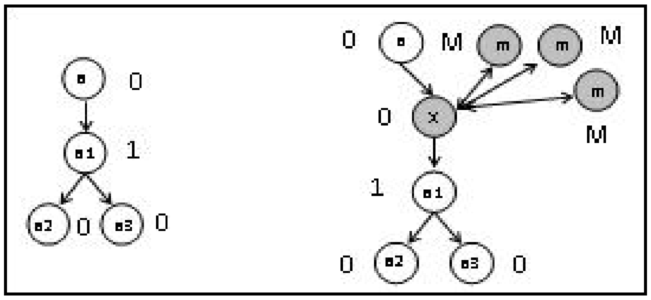

Figure 1. The Enhanced Control Flow Graph.

Every mutant related vertex is connected with its original corresponding node and represents the mutant related necessity constraint. Figure 1 demonstrates the construction of the enhanced control flow graph (the original CFG on the left and the enhanced including three mutants on the right). The proposed approach test model is completed by adding weights on the CFG nodes. The original CFG nodes are assigned a weight value calculated as defined in section III.*B*. The mutant related nodes are assigned an infinite weight value in order to be ignored through the path selection method. Recall that the selected candidate paths for a specific branch are the *k*-shortest ones passing through that branch that ignore the infinite weighted nodes.

Since a path not containing any mutant vertex represents an exact path on the test code unit, that path can form the basis for the fulfilment of 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. Thus, covering all enhanced control flow graph branches results in a straightforward coverage of the weak mutation criterion and the fulfilment of the reachability and necessity conditions of all program mutants. Based on the above observations the proposed strategy tries to efficiently cover all branches of the enhanced control flow graph.

As pointed in [9] when testing according to one method targeting at specific test elements there is a realised coverage on the elements of another testing method. The realised coverage is called "collateral coverage" [8], [9] and forms the basis of the proposed strategy. The strategy for present purposes can be detailed as:

1. *Perform **"Branch testing"** for some k-value as described in section III.*
2. *Generate the Enhanced Control Flow Graph.*
3. *Derive the percentage value of mutation coverage resulting from the "collateral coverage" of the branch tests.*

***While** Mutation Coverage < 1 AND i < k,*
***repeat:** steps 4 and 5 once for each uncovered mutant element m.*

4. *Generate the next candidate path* Π_m^i.
5. *If path is found to be feasible then recalculate the value of Mutation Coverage*

The above method embodies the philosophy of the Yates and Malevris method [7] as it reduces the incidence of infeasible paths in generating path sets to kill the mutants.

In comparison with the CBT [4] and DDR [5] methods, the proposed strategy uses a pathwise approach in order to overcome some of the technical problems imposed by symbolic execution, such as loops and arrays. Additionally as pointed in [14], it limits wasted effort by avoiding to symbolically execute unnecessary program paths. As pointed by Coward [15] the use of path set selection for guiding the symbolic execution process has many advantages over the "traditional" way of applying symbolic execution to the whole program, and concludes that it is better to integrate a path selection strategy within the symbolic executor to generate as few conditions as necessary to achieve a given level of coverage.

C. *Incremental selection of paths*

The path generation method used for mutation testing is in analogy to that used for testing branches. The sought path set consists of the *k*-shortest program paths, again in the sense of containing a minimum number of predicates, which together cover all of the program's mutants. Again with the recourse to the principle of optimality, using equation (2), the k^{th} shortest paths through a mutant vertex *m* and fulfilling the reachability and necessity conditions can be defined as:

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

Where again $\Pi^{(j)}(p,q)$ denotes the j^{th} shortest path from p to q for any pair of vertices in the enhanced flow graph. Note that the path condition derived is based on a selected path from the path set $\Pi_m^{(k)}$ in conjunction with the necessity condition related to mutant m.

The strategy targets to weak mutation and thus there is a possibility that generated tests may fail to also fulfil strong mutation. To handle such situations the proposed strategy generates new paths that fulfil the necessity and sufficiency conditions. All mutants are handled based on a *k*-value, the limit on the number of generated paths, the strategy uses. The practicality of the method relies on the selection of an appropriate *k*-value beyond which all left alive mutants are treated as equivalent ones. However, the optimal choice and handling of equivalent mutants based on the most appropriate *k*-value is a matter of ongoing research, see section VII.

D. *The criterion for selecting uncovered mutants*

When there are more than one remaining uncovered mutant elements to be killed, the question that is raised is: Which is the order they should be selected in an attempt to kill them? In other words, which mutant should we try to kill first? The answer to the above question is given by using the well established philosophy that: *a program path that involves $q \geq 0$ 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, the mutant that is selected to be killed is the one whose path is the shortest in length among the others. Ties are broken arbitrarily.

V. Preliminary Experiments

A. *Experimental planning*

This section describes the results obtained by applying the proposed method to a set of 8 java program units in an attempt to empirically assess the effectiveness of the proposed strategy. All the selected subjects have been used in previous studies such as [4], [11] but were rewritten in java for the purposes of this paper. Table I presents the selected test subject details.

TABLE I. Subject programs profile

Test Object	Description	Lines of Code	No. of Branches	Mutants	Equivalent Mutants
Triangle	Triangle classification	45	32	516	88
Newton	Newton algorithm	22	6	169	33
BSearch	Binary search algorithm	20	12	123	7
Bubble	Bubble Sort algorithm	14	8	119	17
Euclid	Euclidian algorithm	10	4	46	11
Cal	Calculate no. between two given days	22	12	187	10
Insert	Insertion Sort algorithm	28	16	207	12
Quad	Quadratic formula	12	4	156	5

Mutation analysis was performed based on the five mutation operators proposed by the Offutt, Lee, Rothermel, Untch and Zapf [6] (i.e. ABS, AOR, UOI, LCR and ROR) which are found to have near equal strength as all mutation operators previously proposed [16]. The experiments were performed using the specification of the above mutation operators as set in [16]. All equivalent mutants were detected by manual analysis and eliminated from the test subjects in order to relatively compute mutation scores.

B. *Branch coverage and necessity constraint enhancement*

This section reports results for the first iteration of the proposed approach. The first iteration stage consists of employing the Yates and Malevris method [7] to the selected set of programs for branch testing and using the selected paths in conjunction with the mutant related constraints (mutant vertices). The experiment examines the impact of performing branch testing and steps 4 and 5, of the algorithm in section IV.*B*, once to every live mutant. Table II reports on the number of feasible paths generated during branch testing (Test Paths column), the branch tests generated (Branch Tests column), the tests resulted through the conjunction of path conditions found during branch testing with the mutant related constraints, (Mutant Tests column) and the achieved mutation scores for both the branch and mutant test sets (M.S. Branch and M.S. Mutants column respectively). The average mutation score achieved for the experiment sample is 81.94% for Branch and 90.2% for the Branch-Mutant test sets using the same path sets.

TABLE II. BRANCH AND MUTANT COV. WHEN BRANCH TESTING

Test Object	Test Paths	Branch Tests	Mutant Tests	M.S. Branch	M.S. Mutants
Triangle	8	8	22	72.7%	94.4%
Newton	2	2	2	100%	100%
BSearch	4	4	9	65.5%	75%
Bubble	4	4	6	88.2%	90.2%
Euclid	2	2	3	68.5%	74.3%
Cal	3	3	5	92.7%	96.0%
Insert	5	5	6	87.8%	91.3%
Quad	2	2	5	80.1%	100%
Average	**30**	**30**	**58**	**81.94%**	**90.2%**

M.S. used for Mutation Score

C. *Leading to higher mutation coverage levels*

This section reports results from employing the proposed approach to the selected set of programs. Table III reports experimental details of the proposed strategy (section IV.*B*). The strategy for example applied on the triangle program at steps 1, 2 and 3 (performing branch testing), results in 8 paths and their tests which cover the 72.7% (table III M.S. Branch) of all non-equivalent mutants. Steps 4 and 5 when applied once to all live mutants result in a mutation score of 94.4% (table III M.S. Mutants) and form a 21.7% incremental coverage on the mutation score (table III M.S. Increment-Same Paths). In order to achieve adequate coverage the strategy generates 2 new paths (table III Path Increment) which result in a 5.6% increment on mutation coverage (table III M.S. Increment-New Paths), resulting in fully killing all the non-equivalent mutants. The same applies to all of the programs used except Newton and Quad which required no extra paths as 100% killing of the mutants was achieved with the initial test paths.

TABLE III. DETAILS ON EMPLOYING THE PROPOSED STRATEGY

Test Object	Starting Paths	Path Increment	M.S. Increment (Same Paths)	M.S. Increment (New Paths)
Triangle	8	2	21.7%	5.6%
Newton	2	-	-	-
BSearch	4	3	9.5%	25%
Bubble	4	2	2%	8.8%
Euclid	2	1	5.8%	25.7%
Cal	3	3	3.3%	4%
Insert	5	4	3.5%	8.7%
Quad	2	-	19.9%	-

M.S. used for Mutation Score

Figure 2, presents the average mutation score achieved per test case obtained from the application of the proposed strategy. The graph is a plot of the average mutation score achieved for the 8 programs used against the ordered generated test cases. For example all the 5 generated test cases achieved on average 86.5% mutation coverage. As it is apparent by looking at the graph, a significant number of test cases are needed in order to fully kill all the live mutants.

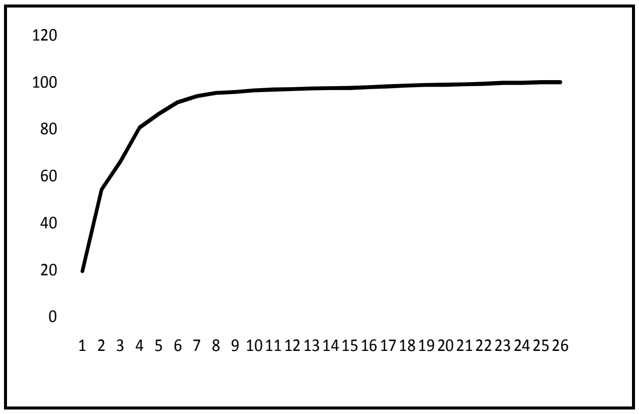

Figure 2. M.S. achieved per test case

VI. RELATED WORK

Despite the usage of mutation testing for a number of years, automated generation of tests based on mutation analysis has received little attention in the literature. Most of the progress in the area has been carried out by DeMillo and Offutt in a method known as the Constraint Based Testing (CBT) Technique [4]. This approach uses control flow analysis, symbolic evaluation [17], mutant related constraints and program execution in order to automatically generate mutation adequate test data. CBT is based on the observation that a test case that kills a mutant must satisfy three conditions (reachability, necessity and sufficiency) as stated in section II. The CBT technique has been implemented in a tool named *Godzilla* for the test of Fortran programs and integrated with the Mothra [16] mutation testing environment. *Godzilla* targets on the reachability and necessity conditions and describes them as mathematical systems of constraints. The reachability conditions are described by path expressions of all program paths that pass through a mutated statement. The necessity conditions are described by specific, to each mutant expression, additional expression(s) in order to infect the program's state immediately after the mutated statement. *Godzilla* conjoins and tries to solve for each mutant its reachability and necessity constraints in order to produce some tests. In this approach there is no straightforward attempt to automatically satisfy the sufficiency conditions. Although CBT has been empirically shown as an effective technique yet it suffers from many shortcomings associated with symbolic evaluation. Specifically, the handling of arrays, loops, non linear expressions and the path explosion are the main problems recognized in [5]. In an attempt to address some of the drawbacks of the CBT method, the Dynamic Domain Reduction (DDR) [5] method was proposed. In DDR the main part of the CBT approach, the generation of reachability, necessity and sufficiency constraints, remains, but enhancements have been made on the way these constraints are handled. The DDR method utilises the program control flow graph and based on a binary search method over the input variables domain, generates test data. In each branch predicate along a given path the variables involved in the predicates are reduced so that the appropriate predicates are true for any assignment of values from the domain. When all selected predicates have been examined, tests are produced based on the reduced domain set.

The DDR approach treats the mutation test generation problem as a pathwise test data generation problem. Its contribution relies on the generation of data for a specific path using a search heuristic on the input domain. Based on selected paths and using a backtracking mechanism Meudec [14] introduces an approach that integrates constraint logic programming and symbolic execution. In this attempt the benefits of the simplicity and back-tracking mechanisms of constraint logic programming are introduced and this resulted in the development of a tool for the structural coverage of Spark ADA programs.

Dynamic approaches based on searching input domain sets have also been proposed. In Sofokleous and Andreou [18], a framework that utilises two optimization algorithms is proposed, the Batch-Optimistic and the Close-Up for generating structural tests. This approach uses a domain control mechanism which starts with small domain spaces and modifies its boundaries to larger ones at subsequent phases. In [19] a hybrid approach that integrates Genetic Algorithms as a search engine of the input domain, a special form of the DDR method on selected paths and symbolic execution based on the Yates and Malevris method [7] was also suggested in the context of performing branch testing.

The benefits of mutation testing highly depend on the number of mutants involved. Strategies involving mutation should therefore attempt to limit the number of the mutants introduced on the one hand, and avoiding to introduce equivalent ones on the other. Such an approach is proposed in [20] where the construction of higher order mutants is discussed. In this work it is suggested that the number of mutants and equivalent ones can be dramatically limited by introducing two or more mutants at a time. A different approach to heuristically deal with equivalent mutants is proposed in [21]. According to the authors, dynamic invariants are introduced into the program under test. The mutants are assessed based on their impact with the invariants. By targeting to those with a higher level of impact, a good measure of the adequacy of the test suite is established, while limiting the number of considered mutants and the equivalent ones. However, both of these approaches rely on mutation analysis rather than on generating test data.

VII. Discussion and Future Work

The proposed method tries to effectively generate test sets by targeting on one of the most promising testing criteria. As suggested in [22] mutants can provide a good indication of the fault detection ability of a test suite and that mutants tend to be similar to real faults. Aiming at a fault based criterion such as mutation, results to be a quite difficult and effort demanding task [3]. Thus, there is a need for practical, effective and systematic techniques which limit the effort spent on the data generation and mutant execution phases. The proposed approach, heuristically deals with the first part of generating test data by simplifying and guiding the process in order to select tests with a limited waste of effort. The approach simplifies the problem of selecting test data by reducing it to generating test data according to a selected path as pointed by Coward [15] and Meudec [14]. Additionally, the proposed method in the present paper, limits the waste of effort by reducing the effects of infeasible paths as stated in section IV.

Targeting on practical issues of mutation testing there is a need for heuristic approaches to automatically reduce and bypass the equivalent mutants problem [3]. As stated by Yates and Malevris [7] it is unlikely to be cost-effective to generate more than (about) the first 40 shortest paths through a branch in order to cover it. Although their claim needs to be validated for the case of mutants, it is unlikely to be generally invalid. The incremental nature of the proposed approach results in cost-effective solutions not only in terms of time but also in terms of effort associated with each mutant. That way the effort spent on trying to kill equivalent or hard to kill mutants is balanced and reduced according to the strategy taken. In view of this, a naive heuristic approach is to treat each live mutant as an equivalent one after the *k* tries of the proposed method.

As presented in section VI there is a limited number of automated tools for generating tests according to mutation analysis. The method presented in this paper can be easily incorporated into an automated tool towards this effect. Effort is made to this direction.

Future work is directed towards conducting more experiments in order to statistically asses the effort, in terms of the paths required to generate mutation test data. A series of experiments is also planned in order to determine the optimal *k*-value in a cost effective strategy. Experiments considering some large programs are also planned.

VIII. Conclusion

This paper presents a practical approach to generate mutation adequate test data. The proposed approach transforms the problem of killing mutants to a path based test data generation problem and applies an efficient path selection strategy in order to achieve adequate coverage. The presented strategy targets on generating mutation adequate test sets in an effective way. This is achieved by a strategy that reduces the effects of infeasible paths and equivalent mutants. The employed strategy not only seeks to minimise the test data generation activity but also tries to balance the effort invested on killing each mutant by stopping the data generation engine based on cost overhead limits.

Transforming the problem of mutation into a path based problem has the advantage of using existing path analysis strategies and tools. Thus, automated tools and testing strategies could be easily applied to fault based testing and improve its effectiveness. Such a strategy with an appropriate adaptation is the one presented here. A preliminary experimentation and association of the effort needed with the number of test paths used for branch testing was performed. Results from some benchmark programs were reported along with an association on the effort needed to achieve mutation coverage with branch testing.

The results were very encouraging, as they showed that the initial path set obtained from the branch testing, on the eight programs used, provided a very good initial path set for killing a good proportion of the mutants straight away, whereas only a small number of extra paths was necessary to be considered in order to fully kill all the mutants for the six of the eight programs. Additionally, for the remaining two programs, the initial path set was capable to generate tests that adequately cover all the programs' mutants. This can be seen as the ability of the proposed method to kill the majority of the selected mutants, as one can continue generating extra paths until such a task is fulfilled. However, and as this was displayed by the appropriate graph, the effort involved in increasing such a coverage is not always advisable. Although the approach relies mostly on the weak-firm mutation hypothesis [11] it is clearly an effective and practical solution in order to generate high quality test sets.

REFERENCES

[1] R. G. Hamlet, "Testing program with the aid of a compiler", IEEE Trans. Softw. Eng., vol. 3, 1977, pp. 279-290.

[2] R. A. Demilo, R. J. Lipton, and F. D. Sayward, "Hints on test data selection: Help for the practicing programmer", IEEE Computer, vol. 11, 1978, pp. 34-41.

[3] A. J. Offutt, R. H. Untch, "Mutation 2000: Uniting the Orthogonal", In Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, October 2000, pp. 45-55.

[4] R. A. Demilo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation" IEEE Trans. Softw., Eng., vol. 17, 1991, pp. 900-910.

[5] A. J. Offutt, Z. Jin and J. Pan, "The dynamic domain reduction approach to test data generation", Softw. Pract. Exper., vol. 29, 1999, pp. 167-193.

[6] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf "An experimental Determination of Sufficient Mutation Operators", ACM Trans. Software Eng. Method. vol. 5, 1996 pp. 99-118.

[7] D. F. Yates, N. Malevris, "Reducing the effects of infeasible paths in branch testing", ACM SIGSOFT Software Engineering Notes, vol. 14, 1989, pp. 48-54.

[8] N. Malevris, D. F. Yates, "The Collateral Coverage of data flow criteria when branch testing", J. of Information & Software Technology, vol. 48, 2006, pp. 676-686.

[9] D. F. Yates, N. Malevris, "An objective comparison of the cost effectiveness of three testing methods", J. of Information & Software Technology, vol. 49, 2007, pp. 1045-1060.

[10] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem", Softw. Test., Verif. Reliab., vol. 7, 1997, pp. 165-192.

[11] A. J. Offutt and D. S. Lee, "An Empirical Evaluation of Weak Mutation", IEEE Trans. Software Eng., vol. 20, 1994, pp. 337-344.

[12] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On two problems in the generation of program test paths", IEEE Trans. on Softw. Eng., vol. SE-2, 1976, pp. 227-231.

[13] D. F. Yates and M. A. Hennell, "An approach to branch testing" In: 11th Workshop on Graph Theoretic Techniques in Computer Science, Wurtzburg, West Germany 1985, pp. 42-433.

[14] C. Meudec, "ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution", J. Software Testing, Verification and Reliability, vol. 11, 2001, pp.81-96.

[15] P.D. Coward and D. Ince, "The symbolic execution of software: The SYM-BOL system", 1995, London: Chapman & Hall.

[16] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, K. N. King, "An extended overview of the Mothra software testing environment", In proceedings of Symposium on Software Testing, Analysis and Verification 2, July 1988, pp.142-151.

[17] J.C. King, "Symbolic execution and program testing", Comm. ACM vol. 19, 1976, pp. 38-94.

[18] A. A. Sofokleous, A. S. Andreou, "Automatic, evolutionary test data generation for dynamic software testing", J. of Systems and Software, vol. 81, 2008, pp. 1883-1898.

[19] M. Papadakis and N. Malevris, "Improving Evolutionary Test Data Generation with the Aid of Symbolic Execution", AIAI 2009 2nd Artificial Intelligence Techniques in Software Engineering Workshop (AISEW 09), April 2009.

[20] M. Polo, M. Piattini, I.G. Rodriguez, "Decreasing the cost of mutation testing with second-order mutants" J of Software Testing, Verification, and Reliability, vol. 19, 2009, pp. 111-131.

[21] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations". In Proceedings of the Eighteenth international Symposium on Software Testing and Analysis (ISSTA '09), July 2009, pp. 69-80.

[22] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Exeperiments? In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), May 2005. pp. 402-411.