Automatic Mutation Test Case Generation Via Dynamic Symbolic Execution

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

***Abstract*—The automatic test case generation is the principal issue of the software testing activity. Dynamic symbolic execution appears to be a promising approach to this matter as it has been shown to be quite powerful in producing the sought tests. Despite its power, it has only been effectively applied to the entry level criteria of the structural criteria hierarchy such as branch testing. In this paper an extension of this technique is proposed in order to effectively generate test data based on mutation testing. The proposed approach conjoins program transformation and dynamic symbolic execution techniques in order to successfully automate the test generation process. The propositions made in this paper have been incorporated into an automated framework for producing mutation based test cases. Its evaluation on a set of benchmark programs suggests that it is able to produce tests capable of killing most of the non equivalent introduced mutants. The same study also provides some evidence that by employing efficient heuristics it can be possible to perform mutation with reasonable resources.**

Keywords- automated test case generation, dynamic symbolic execution, mutation testing, mutant schemata

I. INTRODUCTION

Software testing forms the main technique for identifying and removing software's defects. Its application is performed by selecting candidate test cases and using them in order to exercise the software's behavior. Thus, software testing can be seen as a test case sampling process. The process selects from the whole set (maybe infinite in size) of test cases a small but representative set. This is approximated by selecting tests based on an adequacy criterion. Testing criteria purpose is to guide the tester on selecting a specific set of test cases. That set is the one that covers program elements specified by the criterion. A more demanding (higher in the criteria hierarchy) criterion, usually results at higher quality test case sets. Hence, a higher level of confidence on the tested software can be established.

Mutation testing is widely recognized as one of the most effective techniques for detecting faults. This was reinforced by the studies of Andrews et al. [1] and [2], by providing evidence that mutation testing is able to produce realistic fault detection estimates of a test suite. Additionally, mutation is regarded as rather expensive for practical use. This is attributed to the high number of test elements (mutants) that it introduces, on the one hand and the high number of infeasible ones (equivalent mutants), on the other. Moreover, the lack of efficient and also automated tools for producing the sought mutation based tests is also a fact that prohibits mutation's practical use. These observations indicate that by automating efficiently the generation of mutation tests one can achieve a considerably high level of testing quality and hence establish a high level of confidence with reasonable resources. This forms the main issue of the present paper.

Mutation testing is a fault based testing technique, originally proposed by Hamlet [3] and DeMillo [4]. Mutation works by injecting faults into the source code of the program under test. The testing quality is then measured based on the exposition ratio of these faults, a measure that is called mutation score. The utilized faults are called mutants and they are produced based on syntactic rules called mutation operators. In order to be practical [4], traditionally mutation operators introduce only simple faults i.e. by making one syntactic change at a time. The mutation testing process is performed by executing with the candidate test cases both the original and mutated programs versions. By observing the program outputs one can categorize the mutants into the killed and live categories. Killed are those that result in distinguished outputs while live are those of the opposite case. Generally there are two possible reasons why a mutant has been left alive. The first is that the utilized test data were not capable of revealing it, while the second is that there is no such data. In the second case the mutanted program is functionally equivalent to the original one and called equivalent.

Automatically producing test cases is a very tedious task especially if done in a non automated way. Various techniques and tools have been proposed in the literature considering this task [5], [6], [7]. Generally, test data generation approaches can be divided into two classes, the static and dynamic ones. Symbolic execution [8] is a typical example of the static class while random testing [6], search based optimization [9] and dynamic symbolic execution [5] are typical examples of the dynamic class.

One of the most commonly used techniques for producing test data is the random testing approach. This is because of the simplicity, full automation and direct implementation characteristics of this method. However, such an approach is ineffective to produce tests able to achieve a high level of coverage. This is due to the small probability of selecting specific values of interest from the program input domain. As a consequence random testing must be somehow directed in order to be more effective. This is achieved based on two approaches. The first utilizes search based optimization techniques while the second one is known as dynamic symbolic execution. This

paper considers the second approach. The propositions made in this paper can be applied to search based methods in order to produce mutation based test cases.

Dynamic symbolic execution (DSE) [10] also called directed random testing [11] or concolic execution [5] has been identified as a quite powerful technique for producing test data. This technique performs symbolic execution on programs guided by its actual execution. This is achieved by special program instrumentation that builds the symbolic path conditions dynamically i.e. driven by the program execution. By doing so, the process can be drastically simplified by approximating difficult to handle situations based on the actual values computed during program executions. Additionally, as it relies on program execution the process can always backtrack to random testing. Experiments show that tools utilizing this approach can achieve a remarkable coverage level [12], [10] and [13]. Thus, DSE forms an appropriate choice of test generation for performing mutation.

Generally, in order to expose a fault, the faulty program statement(s) must be exercised. The faulty statement(s) must cause an internal program infection which in order to be detected must be propagated to the program's output. The formulation of the above observation forms the basis of producing tests according to mutation. In the literature three conditions named reachability, necessity and sufficiency [7] are used for this task. The reachability condition states that the test execution must exercise the mutant statement. This can be achieved in a straightforward way by employing DSE. The necessity condition states that the execution of the mutant expression must produce a different internal state to the original one. To achieve this with DSE there is a need to embed into the program's code the mutant infected conditions. The sufficiency condition states that the internal infection must propagate to the program's output. In the present paper this is approximated based on a path space exploration performed by DSE. All the suggestions made in this paper have been incorporated into an automated framework for testing java programs according to mutation.

Conclusively the contributions made by the present paper can be summarized into the following points.

- *An automated technique named Mutation-DSE for producing test cases according to strong mutation testing by employing dynamic symbolic execution. To the best of our knowledge, it is the first technique that uses dynamic symbolic execution to effectively produce mutation based test cases.*
- *An efficient scheme to introduce, execute and integrate the required mutants with dynamic test data generation techniques.*
- *A case study indicating the feasibility, effectiveness and practicality of the proposed approach.*
- *An automated framework for testing java programs, able to produce test cases based on Mutation-DSE.*

The rest of the paper is organized as follows. Section II illustrates the application of the proposed technique on an example program. Sections III and IV detail the underlying concepts of mutation and describe the proposed technique respectively. In Section V, the experimental results of the conducted evaluation study on a set of benchmark programs are given. Section VI presents some related to the present work techniques. Finally, in Section VII conclusions along with a discussion on the proposed technique is given.

II. MOTIVATING EXAMPLE

This example section illustrates how the *Mutation-DSE* can be adopted to produce mutation based test cases. Generally to produce such tests based on a dynamic test data generation approach there is a need to embed all the suitable conditions under which mutants are killed into the program's source code [14]. This is due to the utilized test models that the dynamic approaches incorporate and determine during program execution. For example, DSE requires the inclusion of mutant necessity constraints into the dynamically built path conditions. Thus, the present approach embeds into the program's structure an expression of the following form according to each introduced mutant:

$$original\ expression \neq mutated\ expression \quad (1)$$

To inject these expressions efficiently, a special form of mutant schemata technique has to be employed. Details of this approach are given in the Sections III and IV. For the purposes of this example the mutanted programs can be seen as multiple separate ones with only one mutant introduced at a time. The mutanted program versions also embed expression (1) into their structure. Figure 1 presents an example method. The left part of this figure presents the control flow graph of the original program while the right one presents the control flow graph of a mutated version based on mutant schemata. The mutated program version incorporates all introduced mutants (every schematic function can introduce mutants) but for illustration purposes we may consider only the relational mutant say *m* $(> \rightarrow \geq)$ of vertex 2.

In order to kill a mutant the test data must reach the mutant statement, cause an internal infection and this infection must propagate and have an impact on the program output. To achieve this for the example mutant *m*, the test data must reach vertex 2, make true the expression $(x + y \geq 3) \neq (x + y > 3)$ and return different results, in this case x and x + y (the original program must return x : (vertex 5) and the mutated program $x + y$: (vertices 3, 6) or the opposite. Efficiently producing such test cases based on DSE, requires a guide towards the abovementioned program features [10], [12]. Because of the presence of infeasible paths this guidance can only be a heuristic one, see [15] for further details. The present approach adopts a shortest path heuristic [12], [13] that tries to explore only the relevant to the target mutant part of the program path space.

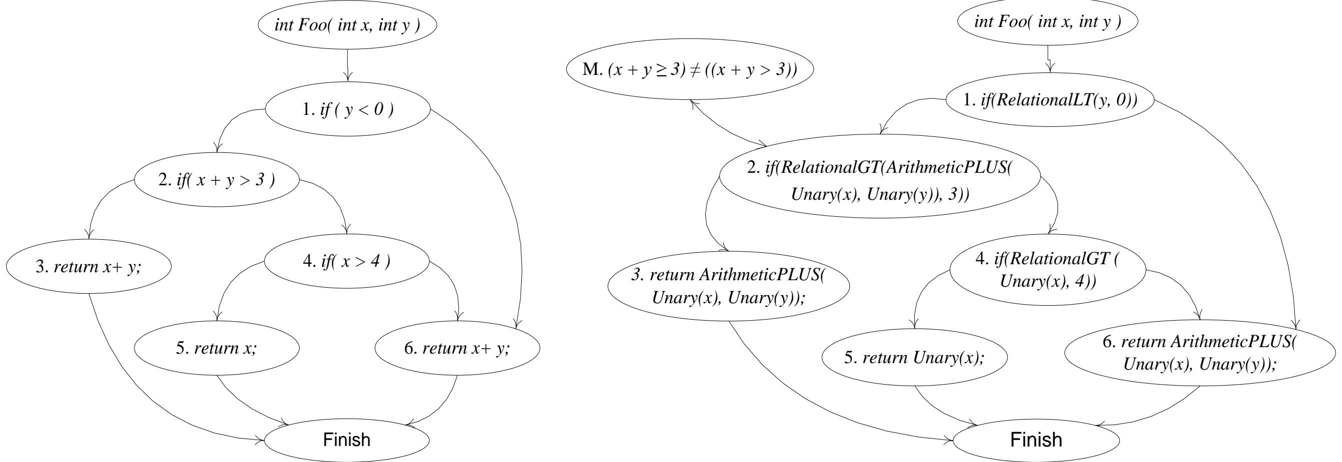

Figure 1. CFG of original and mutated program versions. (Node M repressents the embedded neccesity condition of the relational mutant m:$> \rightarrow \geq$)

This heuristic works by statically selecting the branching node to be flipped based on the minimum distance from the mutant node. This approach is followed for both reaching – fulfilling the mutant necessity constraint and propagating it to the program's output.

Assume that the target mutant *m* of the example in Figure 1 is to be killed. Additionally assume that the existing or randomly generated test data are x = 10 and y = 10. The proposed approach based on the DSE will proceed by executing the initial test set(s) on the mutanted program version (Figure 1 right part). This results in traversing the "1-6-Finish" path and producing the $(y \geq 0)$ constraint. By negating this condition a new input is obtained, let it be x=10, y =-15. This input traverses the path "1-2-3-Finish". Note that the mutant statement has been reached but the data were unable to infect the program state at the mutant point i.e. making the expression $(x + y \geq 3) \neq (x + y > 3)$ true. DSE now produces the path condition $(y < 0) \wedge (x + y \leq 3) \wedge (x + y < 3)$. In order to target on mutant's *m* necessity condition the proposed approach negates the third condition i.e. $(y < 0) \wedge (x + y \leq 3) \wedge (x + y \geq 3)$. This results in a new test case x=18, y =-15 that fulfils the necessity condition of mutant *m*. This test traverses the path "1-2-4-6-Finish" and returns 3 upon exit (x + y on node 6). Note that by executing the original program with the same test case results in traversing the "1-2-3-Finish" path and returning the same result 3 (x + y on node 3). Thus, leaving the mutant alive, despite reaching and forcing it to infect the program state.

The present approach tries to heuristically overcome this difficulty by exploring the path space after the mutant statement, or by taking a different path that reaches and infects the program state or both. In this case the next step towards fulfilling the sufficiency condition is performed as follows. DSE on the test case x=18, y =-15 and mutant *m* produces the path condition $(y < 0) \wedge (x + y \leq 3) \wedge (x + y \geq 3) \wedge (x > 4)$. By negating the last condition i.e. $(y < 0) \wedge (x + y \leq 3) \wedge (x + y \geq 3) \wedge (x \leq 4)$ a mutant killing test case can be produced. Thus, a test, say x=4, y =-1, that traverses the path "1-2-4-5-Finish" path and returns 4 as a result (x on node 5) can be obtained. Note that exercising with the same test cases the original program version results in traversing the path "1-2-3-Finish" path and returns 3 as a result (x + y on node 3). The process is then iteratively continued in a similar fashion for all the mutants remaining alive.

III. Mutation Testing and Framework

A. *Mutation Testing*

Fault based techniques introduce a number of faults into the program's source code and pose the requirement of exposing them. Thus, testing based on a fault based criterion such as mutation requires the production of mutant revealing test cases. DeMillo and Offutt developed a static test data generation technique called Constraint based test data generation (CBT) [7] for the generation of such tests. This approach tries to model with an algebraic set of constraints the suitable killable conditions for the introduced mutants. This can be generally achieved by selecting candidate program paths and specific to each mutant constraints [7] thus fulfilling the mutants Reachability and Necessity constraints. The sufficiency condition because of its high complexity is satisfied indirectly through the joint satisfaction of the reachability and necessity conditions. This approach has recently been extended [16] in order to be performed in an efficient way by reducing the effects of infeasible paths.

Both approaches [7] and [16] rely on a static path selection method and symbolic evaluation. The static form of symbolic execution suffers from several shortcomings that result on low effectiveness and applicability [17]. It is these drawbacks that the DSE method tries to handle [5], [10] and [11]. The major difference in philosophy of the above approaches with the present one, apart from the utilized technical details, is that the others try to statically select and produce the candidate paths and constraints, while the present approach produces them dynamically. To achieve this for mutation there is a need to embed the mutants into the program's under test code. This is performed based on the mutant schemata technique [18].

TABLE I. UTILIZED MUTATION OPERATORS

Description
Replaces a relational expression by altering the included operator with the other operators, and by the true and false expressions, e.g. a > b by a ≥ b, true etc.
Replaces an arithmetic expressioin by altering the included operator by the other operators, and by the first operand and the second one, e.g. a + b by a - b, a etc.
Inserts the absolute, -absolute expression and the requirement of 0 value (if 0 then the mutant is killed), e.g. a by abs(a), -abs(a) and a == 0

B. Parameterized Mutants and Adaptations

The mutant schemata technique [18] encodes all the introduced mutants into one meta-program by replacing each pair of operands participating in an operation with a call to a schematic function that represents this operation. The right part of Figure 1 presents a simplified example of this approach. The present approach expands the suggestions of the mutant schemata technique by incorporating the evaluation of the mutants' execution (formula 1) within the schematic function and after each mutated program statement. This is due to the need of forcing the mutant to infect both the mutant occurrence and the mutant statement. For example consider the application of an arithmetic mutant operator e.g. ($x + y \rightarrow x - y$), say m', on vertex 2 of Figure 1. The m' evaluation results in conditions $x + y \neq x - y$ at mutant occurrence place and $(x + y > 3) \neq (x - y > 3)$ at mutant statement. Note that fulfilling the first condition does not necessarily guarantee the fulfillment of the second one. A similar approach has been undertaken in [14] for reducing the killing of weak mutants to a path - branch coverage problem.

The utilized form of mutant schemata achieves to incorporate the mutant necessity condition into the program structure based on the mutant evaluation statements that it introduces. It must be noted that these evaluation statements do not affect the logic or the performed computations of the program under test and their placement is used only for the test generation process as presented in the example section. The unary increment and decrement operators require a more complicated treatment as they change the program state. Thus, they are replaced by an appropriate expression of plus or minus 1 on the evaluation statements. Following the original suggestions of the mutant schemata technique [18] the utilized meta-program introduces parameterized mutants. This is based on the use of global parameter values which control the application of mutant or original program expressions (by introducing controlling statements of the form: *if (mutantID == targetID)*. In the present approach the parameters additionally control the mutant's evaluation statements in order to avoid introducing considerable overheads. Additionally, as it is explained in the following section, the dynamic nature of the DSE approach effectively ignores the introduced complexity of these controlling statements based on the actual execution.

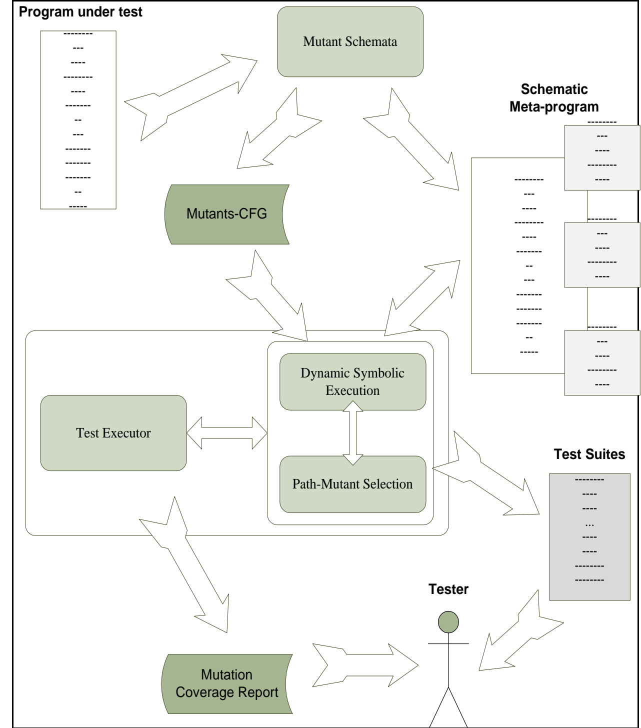

Figure 2. The proposed proccess

C. Proposed Framework

The present approach targets on traditional (method level) mutation operators. The chosen operators for the succeeding case study are those presented and briefly described in Table I. These were chosen based on the propositions made by Wong and Mathur [19] and Offutt et al. [20]. Similar mutation operators are also utilized in many empirical studies and tools such as [1], [2], [21] and [22]. An overview of the proposed approach is given in Figure 2. The process starts by generating the schematic meta-program version of the program under test. Additionally, the mutant schemata generation component produces a static structure of the call and control flow graphs and a list of the introduced mutants with their respective program statements. These two artifacts are then passed to the test generation module. This module iteratively selects a mutant as a target, performs DSE on the schematic meta-program and produces some test cases. These test cases are then passed to the test executor which determines their execution path, the mutants that the test cases can infect and the mutants that are killed. Then the process continues with the next iteration. Finally, after reaching a predefined number of iterations or time limit the process ends and reports the produced test cases and the achieved mutation score.

IV. MUTATION BASED DYNAMIC SYMBOLIC EXECUTION

A. Dynamic Symbolic Execution

Dynamic symbolic execution forms a dynamic adaptation of the traditional static application of symbolic execution [8]. Unlike the traditional approach of statically

selecting candidate paths, producing the path conditions and then trying to find the sough tests, the DSE is directed and approximated by the actual program execution. This is done by performing symbolic execution simultaneously to the actual execution (following symbolically the actual execution path and dynamically building the path condition). The path condition is maintained during the actual program execution based on a monitor mechanism. This mechanism updates appropriately the path condition whenever the course execution encounters a decision program point. Next, using the generated symbolic path constraints and by iteratively negating each one of them it generates new inputs that will traverse a different execution path. Gradually, the process continues until it explores all the feasible program paths. This approach is impractical and thus heuristics should be employed [10], [12] and [13] in order to guide the search towards exploring only a small, relevant and representative set of paths from the whole path space. More details on the utilization of DSE are given in [5], [11] and [13].

The benefits of employing DSE over the static symbolic execution is that the complex expressions or uncontrolled program fragments can always be approximated based on the actual program values encountered during the program execution [5]. This means that the path conditions do not necessarily express precisely all the performed operations or program decisions of a traversed path. Conversely, this leads to an under-approximation of the set of feasible paths which is appropriate for testing [10]. Thus, a trade-off between precision and efficiency has to be paid. One additional benefit identified during the present study was that by selecting candidate paths dynamically, one can effectively ignore some infeasible ones, caused by flag or internal variables. As the values of these variables depend solely on the traversed path, it is guaranteed that trying to negate them will result in an infeasible path. By abstracting the symbolic states and expressions towards the actual execution, these infeasible paths can be easily identified and hence can be ignored. It is this ability of the DSE technique that the present approach takes advantage of in order to efficiently prune away the parameterized mutants (irrelevant to the targeted one) and their evaluations.

B. Mutation Dynamic Symbolic Execution

Employing the DSE for producing mutation based test cases faces many challenges mainly due to the mutation testing special characteristics. First, the vast number of mutated program versions that need to be produced and executed. This forms a research issue for mutation [23] which falls outside the scope of the present research. Despite this, the additional overheads of DSE's requirements can result in a big burden because of: a) the required instrumentation of the monitor mechanism and b) the execution of the mutated program versions. These two facts should normally increase considerably the mutant production, compilation and execution costs. This should constitute a major overhead. Second, the effective handling of mutant necessity and sufficiency constraints

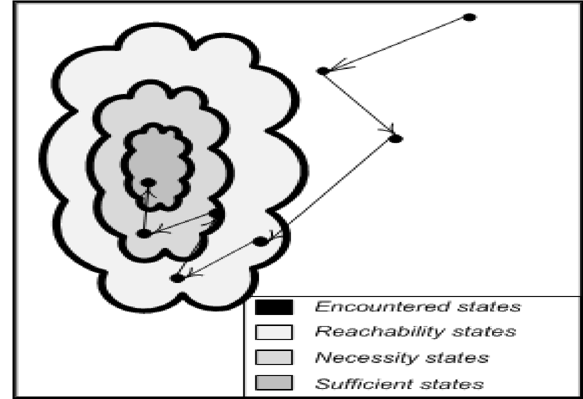

Figure 3. Incremental selection of tests-candidate program states

under the dynamic nature (the program conditions are built dynamically) of DSE must be addressed. Third, scalability issues of the approach must also be addressed. Recall DSE targets on finding all feasible program paths, a quite costly and often impossible task. Its application in combination with the vast number of mutanted program versions should result in prohibitive levels of effort or cost.

The proposed approach in order to be practical tries to handle all the above mentioned difficulties in an efficient way. It adopts a state of the art technique, mutant schemata, in order to produce and execute the mutated program versions under either the DSE mode or actual mode, efficiently. By utilizing the schemata DSE overheads are kept to a minimum, since only one meta-program is produced. Additionally, a considerable reduction on the required execution time can be achieved, when compared to the separate compilation process [22]. Furthermore, the special form of the utilized schemata, as described in the above section, embeds all the mutants' necessity conditions into the program's structure. The reason for doing so is twofold. First, it results in the incorporation of the mutants' necessity conditions into the dynamically built path ones, fact that makes possible the killing of mutants based on DSE. Second, by doing so it becomes possible to execute all mutants with one execution run [14] and thus record the infected mutants efficiently. Then, the process can continue by executing according to strong mutation only those mutants that have been left alive and are able to infect the program state (the rest can safely be ignored). This results in major execution savings. Additionally, the benefits of DSE, such as the simplifications based on the actual program values, also apply on mutant conditions (cases of arithmetic operators or method calls i.e. *, %, / power).

C. Proposed Approach

The practicality of the proposed approach is mainly based on the utilized heuristics for the efficient handling of the mutant's reachability, necessity and sufficiency conditions. The proposed approach utilizes in an incremental fashion the shortest path heuristic, which was found to be a quite effective one in the context of branch coverage [12], [13]. Informally the approach follows the

procedure of Figure 3. The approach tries to incrementally drive the search of the path space and program states towards those paths that reach, infect and expose the targeted mutants. In Figure 3 the outer box represents the whole program state population, the black point indicate the generated states while the neighborhoods (clouds) represent the space elimination to hit the targets.

The proposed approach works by selecting the next branching point to be negated based on its minimal distance from the targeted mutant statement. Specifically it operates on three conditions-modes: a) into trying to reach the mutant, b) trying to infect and c) trying to propagate. If the selected test does not reach the mutant statement (case a) the utilization of the minimum distance approach is straightforward. If the mutant is reached (case b) it selects and negates the mutant "simple" necessity constraint (at the mutant position level), the combined one (at the mutant statement level) and the original one. In case of non symbolic constraints (involve only local variables) a random flipping point is selected.

Mutation Based Dynamic Symbolic Execution

```
Required: Set T (initial test cases)
          Meta-CFG (the Control flow graph of the meta-program)
          MutantSet (set of introduced mutants)
          k,z –value limits

1. Prodedure main
2. Order MutantSet in an ascending order of distances from the
     program's input to the mutant statements
3. TargetSet = MutantSet
4. for repeated times //algorithm repeatitions
5.   for each mutant m ∈ TargetSet
6.     index = 0
7.     while m is live AND index < k // k tries
8.       t = NextTest(m)
9.       f = Flip(t, m)
10.      run and produce path condition pc //DSE
11.      negate the flipping point(s) f and produce path condition(s) pc'
12.      if pc' is solvable
13.             produce a new test case t
14.             add t on T set
15.             execute mutants against t, record execution path
       and infected mutants
16.             remove killed mutant from TargetSet
17.      end if
18.    index ++
19.    end while
20.  end for
21. end for

22. Procedure NextTest(m)
//Selects the next test t to proceed with DSE
23. Select all tests T' from set T that traverse different program paths
24. put T' on a priority queue that prioritizes tests based on the following
     criteria a) infect m (highest priority) and b) according to their
     minimum distance from the m's vertex (higher priority on the
     closest to the mutant vertex traversed paths).
25. return the next test from the priority queue

26. Procedure Flip(t, m)
//Selects the flipping point f of test t in order to kill mutant m
27. if m is infected
28.    select the z closest points after the mutated vertex
29. if m is reached
30.    select the mutant statement(s)
31. if m is not reached
32.    select the closest point to the mutated vertex
```

To make this more understandable consider the case of $(a + b > k \rightarrow a + abs(b) > k)$. Suppose that this mutant has been exercised with the data $a=15,\ b = 2,\ k = 0$. The process at first tries the condition $(b < 0)$ (produced by the mutant position level condition) and produces the test $a = 15,\ b = -10,\ k = 0$. This test infects the mutant at this position level but not at the statement level. Then, it tries to infect at statement level, *i.e.* $(b < 0) \wedge (a + b > k) \wedge (a + abs(b) \leq k)$. As this is found to be infeasible, the process continues with the original statement *i.e.* $(b < 0) \wedge (a + b \leq k)$ and produces the test $a = -15,\ b = -10,\ k = 0$. Finally, it repeats the negation at statement level, *i.e.* $(b < 0) \wedge (a + b \leq k) \wedge (a + abs(b) > k)$ and produces the sought test $a=5,\ b = -10,\ k = 0$.

Finally if the targeted mutant has been reached and infected but not exposed (case c) the process selects and negates z branching points after the mutant statement. The whole process is performed *k* times before moving to the next target mutant. This is an essential cost effective limit enforced by the presence of equivalent mutants, infeasible paths and incompleteness of the technique.

This approach, despite being an approximation technique, in practice can be quite effective even by employing small *k* and *z* values. This is attributed to the collaterally killed mutants, i.e., mutants that are killed without being targeted. Thus, if the technique fails to kill a mutant, it still has many chances to kill it by targeting on the remaining alive ones. Furthermore, as the process produces new tests by using existing or random ones, it becomes possible for the tester to assist the process by adding some manually produced test cases. These can then be utilized for the purposes of killing additional mutants.

Finally, a complete view of the proposed approach is given by the Mutation Based Symbolic Execution algorithm which utilizes the concepts and techniques described in this and the previous sections.

V. Evaluation Study

This section describes an empirical evaluation of the proposed approach on a selected set of widely used in the literature programs. The following case study has been performed utilizing an automated framework for the test of java programs. This framework has been implemented on top of the jFuzz tool [24], by incorporating the suggestions made in the present paper. Although the utilized programming language is java, the primary examined aim is the method level mutation operators [22], described in Table I. Thus, the results should also hold for procedural languages, such as C, too. The following experimental study provides some evidence indicating the applicability, the effectiveness and efficiency of the proposed approach.

A. Experimental Regime

The experiment described in this section uses a prototype implementation of the proposed approach on a set of five extensively used programs in experimental studies in the context of mutation. Details of these programs are given in Table II. The first of the selected

programs (*Tritype*) has been widely used in test data generation approaches of mutation [7], [16], [25] (see related work section). The second one (*remainder*) forms an implementation that calculates the remainder of a division procedure. The last four of the selected programs are part of the well known Siemens program suite [26]. These programs (*Replace, Tcas and Schedule*) have been used in most of the recent experimental studies of mutation i.e. [1], [2] and [23]. For the purposes of the present experiment these programs were rewritten in java (the programming language handled by the prototype tool) taking special care on keeping unaffected the control flow structure and the program computations. This is a common practice in testing experiments also undertaken in [19] and [27]. Using the selected programs an experimental study was undertaken in order to determine the benefits of employing the proposed approach. To this extend, two measures were employed, one representing the method's effectiveness i.e., the ability of the produced tests to kill most of the mutants and one representing the method's cost, i.e., the process required amount of effort to achieve specific levels of coverage. The experiment and the obtained results were based on the following parameters. A *k*-value (line 7 of the algorithm) equal to five, two repeated times (line 4 of the algorithm) and *z*-value equal to all the unique propagation points of each case (line 28 of the algorithm).

To investigate the ability of the proposed approach to produce mutation based test data the number of the killed mutants was measured per chosen program. This number was then compared with an estimate to the number of all killable (non equivalent) mutants. The identification of equivalent mutants form a well known undecidable problem [28] and their manual evaluation require extensive analysis [21] resulting in huge amount of effort. Thus, in order to complete the present experiment with reasonable resources the number of equivalent mutants was estimated based on the number of the killed mutants by the accompanied with the programs test suite (the Siemens programs have a comprehensive accompanied test suite, see the [29] for details about the construction of these tests). This approach is common on mutation testing experiments, also undertaken on most of the recent experimental studies of mutation testing, see [23]. It is noted that for the Tritype and Remainder programs the equivalent mutants were detected based on manual analysis, thus, measuring precisely their number.

The method's required cost on achieving specific levels of coverage is measured based on the number of the algorithm's iteration cycles. This number reflects the application cost of the approach as it is proportional to the number of calls to the utilized constraint solver. The number of constraint solver calls forms a representative effort estimate [10], as the time taken by the various constraint solvers dominates the entire process [10] (the time spend by the strategies on selecting nodes or mutants constraints to be solved is negligible) and also this time may vary significantly between different utilized solvers. An additional cost factor, introduced by the mutation testing process, is related to the required number of the mutant executions in order to determine the killed mutants. A similar cost factor has been considered in [30]. The present approach tries to reduce this cost factor efficiently as described on Section IV.*B*.

B. Experimental Results

This section reports results on performing mutation testing based on the selected program set. Figure 4 presents the cost and effectiveness measures of the proposed approach for the six employed programs. The graphs are plots of the number of killed mutants against the required number of iterations. The horizontal lines at the top of the graphs represent an estimate of the killable mutants' number (killed mutants from the accompanied test suites). From these plots it can be observed that a significant number of the killable mutants can be killed by requiring only a relatively small number of the algorithm iteration cycles. In all the examined cases the percentage of killed mutants over the estimated killable ones is above the 85% level (*Replace* 86.5%, *Tcas* 100%, *Schedule* 91%, *Remainder* 97.5 and *Tritype* 96.8%). Additionally, it can be observed that the increase of the killed mutants converges significantly at the beginning of the process and gradually falling thereafter. The slow convergence of the process after a number of iteration cycles should be attributed to the existence of equivalent and hard to kill mutants. Aiming at those hard to kill mutants requires more iteration cycles, resulting in a considerable amount of additional effort.

The existence of equivalent mutants imposes the need for additional attempts of the utilized approach in order to be killed. As this cannot be done, these attempts fall astray. Moreover, as the equivalent mutants' number remains constant for all the process iteration cycles and the number of killable mutants decreases, the percentage of equivalent to killable ones gradually increases [21]. Thus, the effect of the equivalent mutants on the required process effort gradually increases fact that is evident on the plots of Figure 4. Considering the killable mutants, it can be observed that some mutants remain alive after the utilized number of iterations. An inspection analysis on the live killable mutants revealed that the process achieved to infect them, while it failed to make these changes observable to the program's output (failed to propagate).

TABLE II. SELECTED PROGRAMS

Program	**Description**	**Lines of Code**	**Number of mutants**
Tritype	*Triangle classification*	*40*	*314*
Remainder	*Remainder evaluation*	*50*	*324*
Replace	*Pattern matching and substitution*	*500*	*937*
Tcas	*Trafic collision avoidance system*	*120*	*213*
Schedule	*Process scheduler*	*200*	*165*

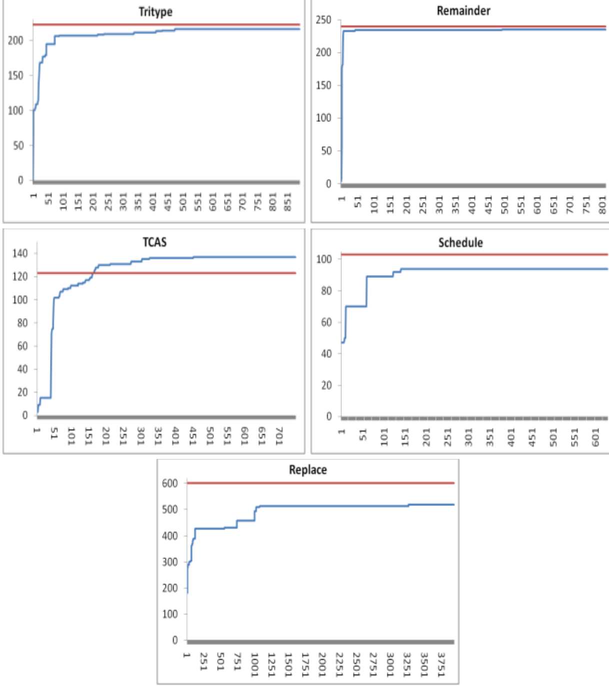

Figure 4. Mutants killed VS iterations number by the Mutation-DSE

TABLE III. APPLICATION RESULTS

Program	½ Iterations			All Iterations		
	Killed Mutants	*Solver Calls*	*No. Executions*	*Killed Mutants*	*Solver Calls*	*No. Executions*
Tritype	*214*	*1012*	*500*	*216*	*1588*	*514*
Remainder	*234*	*2749*	*435*	*235*	*5498*	*741*
Replace	*514*	*14740*	*19400*	*520*	*30238*	*80471*
Tcas	*136*	*1733*	*3729*	*137*	*3705*	*4720*
Schedule	*94*	*714*	*293*	*94*	*949*	*303*

C. *Mutant Execution Cost*

A major part of the mutation testing overheads is due to the execution of the candidate set of mutants. As the dynamic approaches rely on many program executions their use on mutation is escalated due to the required high number of actual executions (vast number of mutants). Table II measures the overheads encountered by the proposed approach. Additionally, Table II summarizes the obtained results, recording the number of killed mutants, the number of calls to the underlying solver and the number of mutant executions for the undertaken iterations (½ of all undertaken iterations and the total as recorded in the graphs of Figure 4), for the selected set of programs.

From these results it can be observed that only a small number of mutants can be killed in the second half part of the employed iterations, while the number of mutant executions and the number of calls to the constraint solver (cost factors) increase significantly. It is noted that the number of mutant executions in the cases of *Replace* and *Tcas* increase faster than the number of constraint solver calls, due to the existence of mutants that are infected but not killed by the produced test cases.

VI. RELATED WORK

Over three decades have gone by since the initial suggestions of mutation [3] and [4] During this period, a considerable amount of research has been carried out as stated in the survey conducted in [23]. Despite this, particular aspects of mutation, such as the automated generation of test cases, have barely been researched. This fact is common to all the structural testing methods, where, only the last years substantial progress is recorded. Thus, the application of these advances to mutation testing form an active research area that also motivated the suggestions made in the present paper.

Most of the progress in the area of generating test cases according to mutation has been carried out by the work of DeMillo and Offutt [7] in a method named Constraint Based Testing (CBT). As discussed in the previous sections the CBT approach tries to model the reachability, necessity and sufficiency conditions into algebraic constraint systems. Then these algebraic systems are solved in order to produce the sought test data. This is achieved based on the use of techniques such as symbolic evaluation, constraint based testing and domain reduction. In this approach a static form of symbolic execution is employed. This results into problems mainly due to the static nature of the symbolic evaluation such as the handling of arrays, loops, non linear expressions and the path explosion problem. To overcome these difficulties a dynamic approach needs to be employed [17], such an approach is the one proposed in the present paper.

Another approach based on symbolic execution and the selection of paths is the work presented in [16]. In this work an augmented graph called enhanced control flow graph is constructed. This graph embeds all the mutant conditions into its arcs providing a suitable test model for producing the required test cases. By employing an efficient path selection heuristic on this augmented graph, one can achieve high mutation coverage with limited resources [16]. Additionally, cost effective strategies can also be employed thus providing decisive advantages compared with the CBT method.

Attempts on adopting dynamic test data generation approaches such as evolutionary testing have also appeared in the literature. Ayari et al. [25] proposed an evolutionary approach for the generation of mutation test data. In this work a search based minimization technique was employed in order to generate test data. The chosen techniques were some metaheuristic search methods guided by a fitness function that measured the closeness of reaching the targeted mutants. Another evolutionary based approach is that of Braudry et al. [30]. In this work genetic algorithms were employed in order to augment an existing test suite according to mutation. The fitness calculations were performed based on the achieved mutation score. The drawback of these two methods is that it do not guide the search method by quantitatively measuring the closeness of killing specific mutants. This fact makes the search not only inefficient but also ineffective for killing many mutants. Nevertheless, these two approaches employ a

different philosophy to the one presented here. However, such a comparison falls outside the scope of the present research and forms a matter of future research.

Approaches using dynamic test generation methods for mutation are quite limited. Perhaps the only, to the authors knowledge, approach is the work in [14] on weak mutation. In this attempt, a special form of mutant schemata is employed in order to reduce the mutant killing problem (weak mutants) into a covering braches one. Then, by employing existing automated tools for structural testing, these mutants-branches can be effectively killed-covered. The present approach is based on these ideas and extends them in order to target on a more demanding criterion such as strong mutation. Additionally, the present research specifically deals with some of the special characteristics of mutation as described in Section IV. Further, the present research also considers the method's efficiency.

Considering the test data generation approaches in general, applied on different testing contexts e.g. branch testing, many more attempts appear in the literature. The most relevant to the present one are those that utilize DSE [5] and [11] and have already been discussed in the previous sections. In [12] and [13], a shortest path heuristic has been employed in order to efficiently perform branch testing, with promising results. In [10] some additional heuristics inspired by the fitness functions used in evolutionary testing [9] are effectively adopted for guiding the DSE process.

Finally, various tools and techniques have been proposed by researchers based on either pure symbolic execution or pure dynamic approaches such as search based testing [9]. In [31] a symbolic execution system for testing java programs is proposed. This approach uses symbolic execution only at specific program features selected by the tester. Then by using actual program executions it is achieved to both effectively "set-up" the symbolic execution environment and "proceed" with the irrelevant to the interest features program parts. Additionally, an automated and quite powerful tool that incorporates evolutionary testing was suggested by Tonella [32]. Further, in the work of Lakhotia et al. [33] a technique that adapts search based optimization for the test of programs containing pointers and dynamic data structures has been proposed.

VII. Conclusion And Future Work

This paper introduces a novel technique for automating the generation of mutation based test cases. The technique proposed here makes use of state of the art techniques such as mutant schemata and dynamic symbolic execution in order to produce high quality test data. The philosophy behind the proposed approach is to use mutation analysis for producing tests instead of assessing them only. Thus, mutation acts as a yardstick towards producing the sought tests. Additionally, the present approach automates efficiently the mutant evaluation process in order to minimize the mutation testing overheads. This is achieved by combining two special forms of mutant schemata, one for weak and one for strong mutation, resulting in major execution savings as the experimental results show.

The suggestions made in this paper have been implemented into an automated framework for java. The performed experimental study constitutes one of the few ones (see related work section) and also the largest one in the context of mutation based test data generation. The experimental results obtained in this paper suggest that the proposed approach can produce test cases of high quality as they are able to kill a remarkable number of the introduced mutants. Evidence, is also provided to support the argument that these tests can be produced at a relatively low cost and that the proposed approach can be also applicable to larger cases. Further, by utilizing all the benefits of the DSE technique (handling of pointers, arrays, loops, non linear expressions and the path explosion problem), the present approach comprises a major improvement over the existing techniques.

The conducted study has also revealed some interesting points–observations concerning test data generation and mutation testing. The first observation is that the existence of collaterally killed mutants (mutants that are killed when targeting to others) helps to overcome the faced difficulties of killing specific mutants. Thus, if the process fails to kill a non equivalent mutant (when targeting to it), it still has a chance of killing it by targeting to others. The second observation concerns some benefits of aiming at equivalent mutants. Generally, it is believed that attempts to kill equivalent mutants result in a waste of effort. While this being true, the attempts to kill them may result in killing, reaching or infecting other mutants collaterally. Additionally, the incremental nature of the proposed approach (targeting first at mutant infection) may result in producing test cases capable of infecting equivalent mutants. These tests are redundant in respect of strong mutation but not in respect of weak and for this reason they are of additional value. Furthermore, if the utilized approach fails to kill some non-equivalent strong mutants it may still be able to kill the respective weak ones (finds tests able to infect the mutant, but fail to propagate to the program's output) resulting in some additional test cases (possibly valuable). Moreover, if the produced tests are able to kill all non equivalent weak mutants, these tests, under the use of specific mutant operators [34] can subsume various structural criteria such as the multiple condition coverage criterion [34]. It is noted that this argument does not hold for strong mutation because of the existence of strongly equivalent mutants. The last observation obtained from this experiment suggests that by employing a mixed approach of symbolic execution and random testing (find data up to certain points and then continue with random or actual execution) when targeting on mutants, results in finding many feasible paths collaterally. Thus, most of the mutants have been reached from the previously examined mutants reducing the required effort to reach them.

In future some extensions are also being planned. These are the effective incorporation of additional heuristics dealing with the path explosion and the

equivalent mutant problems. Specifically, the incorporation of appropriate fitness evaluations [10] and search based testing [9] is currently under research. Additionally, series of experiments are also scheduled in order to efficiently incorporate heuristics for the identification of equivalent mutants such as [28] and potential strategies to avoid them along the lines suggested by Schuler and Zeller [21].

ACKNOWLEDGMENT

This work is supported by the Basic Research Funding (PEVE 2010) program of the Athens University of Economics and Business.

REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005, pp. 402-411.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," IEEE Trans. Softw. Eng., vol. 32, pp. 608-624, 2006.

[3] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," IEEE Trans. Softw. Eng., vol. 3, pp. 279-290, 1977.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol. 11, pp. 34-41, 1978.

[5] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal, 2005, pp. 263-272.

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in Proceedings of the 29th international conference on Software Engineering, 2007, pp. 75-84.

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

[8] J. C. King, "Symbolic execution and program testing," Commun. ACM, vol. 19, pp. 385-394, 1976.

[9] P. McMinn, "Search-based software test data generation: a survey: Research Articles," Softw. Test. Verif. Reliab., vol. 14, pp. 105-156, 2004.

[10] T. Xie, N. Tillmann, P. d. Halleux, and W. Schulte, "Fitness-Guided Path Exploration in Dynamic Symbolic Execution," in the 39th International Conference on Dependable Systems and Networks, 2008.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago, IL, USA, 2005, pp. 213-223.

[12] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 443-446.

[13] C. Cadar, D. Dunbar, and D. Engler, "KLEE Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in OSDI, 2008.

[14] M. Papadakis, N. Malevris, and M. Kallia, "Towards Automating the Generation of Mutation Tests," in AST 2010, Cape Town, 2010.

[15] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On Two Problems in the Generation of Program Test Paths," IEEE Trans. Softw. Eng., vol. 2, pp. 227-231, 1976.

[16] M. Papadakis and N. Malevris, "An Effective Path Selection Strategy for Mutation Testing," in Proceedings of the 16th Asia-Pacific Software Engineering Conference, 2009.

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

[18] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, Cambridge, Massachusetts, United States, 1993, pp. 139-148.

[19] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: an empirical study," J. Syst. Softw., vol. 31, pp. 185-196, 1995.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," ACM Trans. Softw. Eng. Methodol., vol. 5, pp. 99-118, 1996.

[21] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," Paris, France, 2010.

[22] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research Articles," Softw. Test. Verif. Reliab., vol. 15, pp. 97-133, 2005.

[23] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions of Software Engineering, vol. To appear, 2010.

[24] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for Java," in NFM, 2009.

[25] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in Proceedings of the 9th annual conference on Genetic and evolutionary computation, London, England, 2007, pp. 1074-1081.

[26] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," Empirical Softw. Engg., vol. 10, pp. 405-435, 2005.

[27] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold, "MATRIX: Maintenance-Oriented Testing Requirements Identifier and Examiner," in Proceedings of the Testing: Academic \& Industrial Conference on Practice And Research Techniques, 2006, pp. 137-146.

[28] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," Software Testing, Veri and Reliability, vol. 7, pp. 165-192, 1997.

[29] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003, pp. 60-71.

[30] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment," in Proceedings of the 13th International Symposium on Software Reliability Engineering, 2002, p. 195.

[31] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, WA, USA, 2008, pp. 15-26.

[32] P. Tonella, "Evolutionary testing of classes," SIGSOFT Softw. Eng. Notes, vol. 29, pp. 119-128, 2004.

[33] K. Lakhotia, M. Harman, and P. McMinn, "Handling dynamic data structures in search based testing," in Proceedings of the 10th annual conference on Genetic and evolutionary computation, Atlanta, GA, USA, 2008, pp. 1759-1766.

[34] J. Offutt and J. Voas, "Subsumption of Condition Coverage Techniques by Mutation Testing," Dept. of Information and Software Systems Engineering, George Mason Univ., Fairfax, Va., 1996.