Software Quality Journal, vol. 19(4), pp. 691-723, 2011, (The final publication is available at www.springerlink.com, http://dx.doi.org/10.1007/s11219-011-9142-y)

Automatically Performing Weak Mutation with the Aid of: Symbolic Execution, Concolic and Search Based Testing[1]

Mike Papadakis and Nicos Malevris

Department of Informatics
Athens University of Economics and Business
{ mpapad, ngm } @ aueb.gr

ABSTRACT

Automating software testing activities can increase the quality and drastically decrease the cost of software development. Towards this direction various automated test data generation tools have been developed. The majority of existing tools aim at structural testing, while a quite limited number aim at a higher level of testing thoroughness such as mutation. In this paper an attempt towards automating the generation of mutation based test cases by utilizing existing automated tools is proposed. This is achieved by reducing the killing mutants' problem into a covering branches one. To this extent this paper is motivated by the use of state of the art techniques and tools suitable for covering program branches when performing mutation. Tools and techniques, such as symbolic execution, concolic execution and evolutionary testing can be easily adopted towards automating the test input generation activity for the weak mutation testing criterion by simply utilizing a special form of the mutant schemata technique. The propositions made in this paper integrate three automated tools in order to illustrate and examine the method's feasibility and effectiveness. The obtained results, based on a set of Java program units, indicate the applicability and effectiveness of the suggested technique. The results advocate that the proposed approach is able to guide existing automating tools in producing test cases according to the weak mutation testing criterion. Additionally, experimental results with the proposed mutation testing regime show that weak mutation is able to speedup the mutant execution time by at least 4.79 times when compared with strong mutation.

Keywords

Mutation testing, weak mutation, automated test case generation, mutant schemata, symbolic execution, concolic execution, search based testing.

1. INTRODUCTION

Software testing is a very expensive activity as it can consume 50% or even 60% of the total cost of the software life cycle. To reduce the software testing cost, an excessive amount of effort has been put towards automating the test data

[1] This paper is an extended version of the (Papadakis et al. 2010) paper.

generation process, thus also reducing the overall software development cost. This activity is usually performed by utilizing an automated tool that produces the sought test data. In the absence of such tools this activity must be manually performed making the testing cost exceedingly expensive. Therefore, the need for automating this process is imperative, especially when employing expensive testing techniques. Usually, to evaluate that a piece of software has been thoroughly tested, a collection of requirements are selected and checked whether they have been successfully executed with test cases. Requirements that have received considerable popularity are the structural test coverage criteria basically because of their effectiveness, ease of use and straightforward evaluation.

Mutation testing is a powerful fault-based yet highly expensive testing technique initially introduced by Hamlet (Hamlet 1977) and DeMillo et al.(R. A. DeMillo et al. 1978). This technique is the basis of the present work and an attempt to automate the test data generation process for its effective use is investigated. The successful automation leads to a successful cost reduction, thus allowing mutation testing to be more usable and manipulable. Mutation analysis is based on the production of syntactical alterations of the code under test aiming at producing semantically different program versions. The different program versions are called mutated versions as each one contains a simple syntactic change of the original code. The role of the test cases is to unveil these purposely syntactic alterations by distinguishing the mutated programs from the original one. A mutant is termed “killed” if there is a test that distinguishes its output from that of the original program whereas; in the absence of such test cases it is termed “equivalent”. The percentage of the mutants killed is used as a measure of the testing thoroughness of the method. Although mutation has been shown to be quite powerful (J. H. Andrews et al. 2005), (James H. Andrews et al. 2006) it has unfortunately proved to be highly demanding in order to generate and execute the mutated versions. In view of this and in order to reduce the resulting cost, various mutation techniques have been proposed. One such technique namely “weak mutation” (Howden 1982) targets on reducing the process execution cost. It suggests stopping the program execution of the mutated programs immediately after the mutated statements are executed with data. One other additional technique, called mutant schemata, targets on reducing the generation and compilation cost of the produced program versions (Untch et al. 1993). This technique produces one meta-program that embeds in its structure all mutated versions. Both these techniques have been assessed empirically and details can be found in(Ma et al. 2005) and (A. J. Offutt and Lee 1994) with promising results.

To find appropriate test data with relevance to a selected criterion can be a very tedious task (A. Jefferson Offutt and Untch 2001). This constitutes a major problem for full or partial automation. Unfortunately, this is the case for mutation and its variants i.e. weak mutation too. In general, approaches employing mutation based test case generation are quite scarce in the literature. Most of the progress in the area has been reported by (Richard A. DeMillo and Offutt 1991) in a technique called Constraint Based Testing. This approach although powerful has not been implemented or incorporated in an automated tool for modern programming languages such as Java. Additionally, it suffers from many shortcomings due to the employed static test generation engine. Conversely, there appear not to exist any fundamental attempts that effectively utilize recent advances of symbolic execution (J. C. King 1976), concolic execution (Sen et al. 2005) and search based optimization techniques (Harman and McMinn 2007); state of the art techniques that have succeeded in automating at an acceptable level the test generation activity for structural testing.

The approach in the present paper automatically reduces the killing mutant’s problem to a covering branches problem. This constitutes the basic achievement of the present work. Treating each mutant as a branch, helps focusing on specific mutants by selecting appropriate paths or tests in order to generate effective data capable of killing them. The benefit of such an activity is that automated tools or techniques implemented for structural testing can be directly utilized to perform mutation testing. Thus, mutation testing automation is reflected on the structural testing automation and efficiency, where known achievements have been recorded. Additionally, by utilizing the suggested special form of the mutant schemata, a considerable amount of the mutant execution cost can be gained. Thus, the proposed approach appears to be suitable for dynamic test data generation approaches which form the current state of the art on test generation.

Conclusively, this paper presents a mutation testing framework that supports the automation of the testing process under the use of mutation testing criterion. The originality of the framework's components, the supporting technologies and tools, is not completely new, although their integration and use under the mutation testing activity form a novel approach. Also, the automatic generation of mutation based test cases via state of the art techniques such as concolic execution and search based testing form an additional contribution by the present research. Specifically, the framework utilizes a state of the art approach due to Amman and Offutt (Ammann and Offutt 2008) and the mutant schemata method (Untch et al. 1993), in order to support the mutant generation and execution. It also utilizes a novel implementation of weak mutation (Howden 1982) with mutant schemata to support major savings with respect to mutant execution required time. The conducted experiment shows that a speedup even more than 10,000 times can be obtained in mutant execution speed compared to the required one by strong mutation. Finally, by utilizing the proposed special regime of mutant schemata, existing test data generation approaches can be straightforwardly used to perform mutation.

The suggestions made in this paper have been incorporated into an automated framework. A case study indicating the applicability of the proposed advances has been undertaken, revealing their strengths. The contribution of the present work can be summarized into the following proposed points:

- An automated technique for reducing mutants to branches.
- An automated framework for performing weak mutation based on mutant schemata.
- A practical approach on using existing automated test data generation tools that rely on either static (symbolic execution) or dynamic (concolic and search based) techniques to produce mutation based test cases. This follows the spirit of avoiding to make any modifications to these tools. Thus, using them unaltered are for producing mutation based test cases.
- An approach that reduces significantly the mutants' execution time. Thus, mutation can be efficiently performed by dynamic approaches, such as concolic execution and search based optimization.

The rest of this paper is organized as follows: Section 2 introduces some background material. Section 3 discusses the foundations utilized by the present research on how to kill the introduced mutants. Sections 4 and 5 detail the proposed technique and its implementation under the proposed framework. Sections 6 and 7 report a conducted case study and discuss the practicality issues induced by the application of the proposed technique. Section 8 presents work related to the present one. Finally in section 9 conclusions with some future directions are discussed.

2. BACKGROUND

The goal of testing criteria is to select a subset of all possible test cases that have a high ability of detecting errors. There are many types of testing requirements e.g. functional, structural and fault-based that examine different program characteristics. In general, structural testing criteria require the examination of the internal composition of the program's source code. Tests are derived to exercise certain program elements such as basic blocks, branches, paths etc., of the program under test. Typically, tests are produced until a predefined level of coverage is reached. The level of coverage according to a selected criterion is defined according to the following ratio:

$$\text{Coverage} = \frac{Test\ Elements\ Covered}{Total\ Elements - Infeasible\ Elements} \qquad (1)$$

The criteria requirements goal is both to guide and evaluate the quality of the test data. Thus, a typical testing scenario on using the above ratio (equation (1)) is the following. First the tester tests the system under test and determines the level of testing thoroughness that has been achieved (evaluation goal). If this level is not satisfactory, according to the undertaken policy, then the tester needs to improve it. In doing so some new test cases must be produced that cover some of the yet uncovered elements. If full (100%) coverage level is required then the tester has to produce test cases able to cover all feasible test elements. In practice this scenario is quite difficult due to the instance of infeasible and hard to cover elements. Additionally, the test generation process is a very laborious task and thus, its automation, even if not entirely, should be quite beneficial.

Based on the criteria hierarchy a proportional to the utilized criterion level of confidence can be established. Although, the research community has not yet resulted in a specific model of the expected effectiveness of the various criteria, it has been shown empirically that utilizing a higher criterion in the criteria hierarchy will result in a higher level of testing thoroughness. Unfortunately, this can only be achieved at an extra cost, a fact indicating that there is a tradeoff between cost and effectiveness of the utilized criteria. Thus, it can be argued that by automating at the same level the generation of test cases of a higher level criterion one can benefit more than by automating the generation of a lower level criterion. Nevertheless, this paper is concentrated on generating mutation based test cases and on how this process can be effectively automated using current state of the art techniques.

2.1 Mutation Testing as a Coverage Criterion

Practically, applying the testing criteria serves as a rule of thumb for selecting a subset of all possible inputs. The main objective of these rules is to guide the selection of test cases to those able to expose the majority of the program's faults. Mutation testing may establish a considerably high level of confidence due to its effectiveness (J. H. Andrews et al. 2005). Thus, such a criterion forms an ideal choice for achieving a high level of testing thoroughness.

Testing based on fault based criteria (such as mutation), requires the exposition of some introduced faults. According to these criteria, a number of faults are seeded into the program's code and fault based requirements are utilized for their exposition. The test coverage is defined along the same lines as for the coverage defined in (1), by calculating the percentage of the faults revealed. As already discussed, mutation testing is a fault based testing technique that introduces faults by making simple syntactic changes to the source code under test. The introduction of the syntactic changes is based on a set of rules called mutant operators. The introduced mutants form either common programming mistakes or typical testing requirements such as the execution of all program branches, conditions etc. Based on the coupling effect (R. A. DeMillo et al. 1978), which states that complex faults are coupled to simple ones in such a way that a test data set that detects all simple faults in a program will detect most complex faults (A. Jefferson Offutt 1992), all mutants have been limited to simple ones. In practice, mutation testing has been empirically found to be one of the most effective software testing methods, as it produces test cases capable of subsuming most of the rest of software testing criteria in the criteria hierarchy (Ammann and Offutt 2008), (Jia and Harman 2010) on the one hand while manifesting the ability to detect more faults than its rivals on the other (Li et al. 2009), (Jia and Harman 2010).

Mutation testing has been applied at various testing levels i.e. unit, integration, system level. These levels differ according to the underlying programming model. Thus, in Object Oriented languages 4 levels have been defined (Ma et al. 2005): intra-method, inter-method, intra-class, inter-class. The intra-method level (Ma et al. 2005), which also forms the focus of the present paper, is the traditional unit testing level of the procedural programming model, of testing the function of a specific method of a class under test. The inter-method level (Ma et al. 2005) aims at faults between method calls, corresponding to integration testing of the procedural programming model. The intra-class and inter-class levels target on faults caused by methods interaction due to a sequence of calls inside a class and due to class integration respectively.

Mutation analysis provides a test criterion, rather a test process (A. Jefferson Offutt and Untch 2001). Thus, its purpose is to provide a stopping rule of the testing process on the one hand and to guide the tester on selecting a specific set of test cases on the other. Following these lines, the present research utilizes mutation in order to guide existing structural testing tools to automatically produce test cases capable of killing mutants. Additionally, it automatically measures the quality of the produced test cases according to the weak mutation testing criterion.

2.2 Mutation testing challenges

Applying mutation requires three main stages that must be performed. These are a) mutant generation, b) test case generation and mutant execution and c) mutation score evaluation. These three steps should be performed together with the coverage criteria and thus, steps b) and c) should be performed iteratively until a predefined coverage goal has been reached. Unfortunately, these stages require a vast amount of resources and thus, researchers have suggested various techniques in order to reduce them (A. Jefferson Offutt and Untch 2001), (Jia and Harman 2010).The present paper concentrates on automating effectively steps a) and b).

The main problems that arise due to mutation testing are: for step a) mutant generation requires the production of a huge number of mutant programs, that should be produced and compiled. To reduce this expense a technique called mutant schemata (Untch et al. 1993) has been proposed. This technique is utilized by the present approach and its details are given in section 4. Test generation and mutant execution in b) require the generation of test cases able to kill the remaining live mutants. Test case generation forms a special issue for all the testing criteria, a task that is also a well known undecidable problem (A. Jefferson Offutt and Pan 1997). Various approaches have been proposed in the literature mainly in the structural testing context; in section 5 such approaches are discussed. To evaluate if this task has been achieved, all mutants (live mutants per iteration stage) should be executed against the produced test cases. The execution of the introduced mutants results in a prohibitive situation due to their vast number (A. J. Offutt and Lee 1994). A possible answer to this situation is the use of weak mutation (Howden 1982) (a mutation variant) which is also utilized by the present approach and its details are given in the following section. Finally, regarding the results of mutant executions an analysis of the achieved mutation score c) is required. Unfortunately, in performing mutation there are many mutants that cannot be killed (infeasible elements) by any test case and thus, they must be removed from the candidate mutant set in order to assess the quality of the produced test cases. This problem has been proven to be an undecidable problem in its general form (A. Jefferson Offutt and Pan 1997) and issues concerning these mutants fall outside the scope of the present paper. Further details can be found in (A. Jefferson Offutt and Pan 1997), (A. Jefferson Offutt and Untch 2001) and (Jia and Harman 2010).

2.3 Weak mutation

Strong mutation, requires considerable amounts of computational resources for mutant execution. To overcome this difficulty, weak mutation (Howden 1982) was proposed as a mutation testing variant. Weak mutation tries to efficiently reduce the required test execution cost by stopping the program execution at the mutant place. Thus, a test execution reduction of approximately 50% can be gained (A. J. Offutt and Lee 1994). The key issue of weak mutation is the mutant evaluation point where actually the program execution should be stopped. Determining if the mutant is dead or alive requires the evaluation of the runtime program states on both the original and the mutated programs. Thus, an implementation of a weak mutation system does not seem to be straightforward. Additionally, such a system should incorporate all the applied to strong mutation advances in such a way that it drastically reduces the mutant generation and execution required time. This forms an issue dealt by the present paper.

Weak mutation coverage criterion requires comparing the outputs of mutant expressions with the respective original ones. According to (Howden 1982), in weak mutation it is acceptable to execute all the mutants for one place when executing program code. Based on this idea the present approach implements an efficient weak mutation testing tool capable of executing all the selected mutants due to their position in the program code. This approach results in major execution savings as the undertaken experiments show (see section 6). Such an approach is along the same lines with the split-stream execution approach suggested but not implemented by King and Offutt (K. N. King and Offutt 1991) for strong mutation and (Mastorantonakis and Malevris 2003) for weak. Additionally, as pointed out before, such an approach helps reducing mutants to branches and thus, assists existing test generation tools to perform mutation.

3. How to Kill Mutants

Generating test cases according to mutation requires seeking ways for revealing all or most of the introduced mutants. If such tests are found then a high level of confidence for the software system that is tested can be established. This is due to the ability of those tests to expose the introduced faults-mutants which have been found to be similar to real faults (J. H. Andrews et al. 2005). Thus, producing tests according to mutation testing will result in a high level of testing thoroughness. In order to do so there is a need for finding the suitable test cases capable of revealing the introduced mutants. Finding such tests forms one of the most difficult technical tasks (A. Jefferson Offutt and Untch 2001) in the software testing activity. Thus, the benefits of fully or partially automating such an activity are evident.

As it has appeared in the literature, the exposition of a seeded fault such as a mutant, should adhere to three conditions known as Reachability, Necessity and Sufficiency (Richard A. DeMillo and Offutt 1991). Based on these three conditions, DeMillo and Offutt developed a test data generation technique called Constraint-Based test data generation (CBT) (Richard A. DeMillo and Offutt 1991) which forms the foundations for killing mutants. These three conditions answer in a general way, the question of how any approach should attempt to kill mutants. Therefore, the proposed approach transforms these conditions (Reachability and Necessity conditions) into program branches, resulting in suitable forms for the use of any existing structural test generation technique.

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 with the original. If tests cannot execute the mutated statement, it is guaranteed that the tests have no chance of killing the seeded mutant (Richard A. DeMillo and Offutt 1991). The necessity condition states that the execution of the mutated statement must cause a departure from the original program state (Richard A. DeMillo and Offutt 1991). This is substantiated by the fact that the execution outcome of the original and the mutated statements must be different. In the opposite situation the syntactical equality of the rest of the two program versions suggests that they will never form different computations and will therefore never result in observable output differences. The sufficiency condition states that the infected program state must propagate up to the last program statement. The execution path and its computations must use the mutated statement and its internal different value (necessity condition) and create a different observable formulation from the mutated statement up to the program's output.

Current test data generation approaches try to tackle these three conditions by transforming them into a suitable form according to the utilized test generation technique. Thus, the CBT approach (Richard A. DeMillo and Offutt 1991), transforms these conditions into mathematical systems of constraints, the (Papadakis and Malevris 2009) approach transforms them into path conditions and the (Ayari et al. 2007) into a search based optimization problem. The (Richard A. DeMillo and Offutt 1991) and (Papadakis and Malevris 2009) studies, try to utilize directly the reachability and necessity conditions based on constraint resolution and domain splitting, while the (Ayari et al. 2007) approach utilizes the reachability condition only. Because of its high complexity the sufficiency condition is indirectly satisfied through the joint satisfaction of the reachability and necessity ones. This is reinforced in (Richard A. DeMillo and Offutt 1991) and (A. J. Offutt and Lee 1994) where it is shown that tests meeting the reachability and necessity conditions have a high chance of meeting the sufficiency condition as well, although fulfilling the sufficiency condition may be highly desirable in order to meet strong mutation requirements. However, by fulfilling the reachability and necessity conditions only, this results in meeting the weak mutation criterion requirements (A. J. Offutt and Lee 1994).

Automated tools targeting on mutation testing are scant due to technical issues concerning mutation analysis and the corresponding test data generation, which is difficult and resource-consuming. It is these difficulties that the present research tries to overcome by utilizing existing state of art methods for performing other forms of testing to perform mutation too. To achieve this, the present approach focuses on weak mutation. To this extent, the necessity conditions are described based on the following formation: Let the original expression be *e* and the mutated one *e'*. Then, the necessity condition is formed by the necessity constraint " *e != e'* " denoted as *N*. This constraint has been formed directly by the definition of the necessity condition and represents a suitable decision statement on whether the mutant is weakly killed or not. By fulfilling the necessity conditions at runtime (at the mutant points) the test cases meet the weak mutation requirements. The introduction of these conditions into the program structure and their fulfillment forms the underlying objective of the present paper in order to guide the test generation process to effectively perform mutation.

An initial attempt towards this direction was suggested (Papadakis and Malevris 2009) based on symbolic execution. According to this method a suitable program representation model called enhanced control flow graph is used. This type of model is constructed by augmenting the program's control flow graph with mutant constraints (such as *N*), by representing each mutant with a special type of vertex. Every added mutant vertex is connected with its original corresponding node and represents the necessity constraint related to this mutant. The augmented graph is then used to select paths that include each mutant in turn in a static manner and then derive appropriate test data by symbolically executing them. The strength of this method is attributed to the unification of all mutant conditions in one appropriate test model containing both path and mutant conditions.

The benefit of the above consideration is that each mutant and its representation on the graph i.e. the original node connected to each mutant node, and vice versa, allows to convert the problem of generating test data to kill each mutant into that of generating data to cover all the branches that connect the original with the mutant nodes. This can be tackled by the well researched problem of generating test data that will cover all the branches in the respective graph. The proposed approach embodies this important characteristic and tries to utilize automated tools for covering program branches in order to kill the mutants. However the drawback of the described approach is that it requires special treatment (the construction of the enhanced control flow graph) and method adaptations according to every test generation method used. This results in being impractical in cases of state of the art test generation methods such as concolic and search based testing. This is due to their dynamic nature that relies on actual program execution and not an underlying test model. As modern test generation methods rely on either static or dynamic analysis techniques or on their combination, the proposed approach extends the suggestions made in (Papadakis and Malevris 2009) by introducing all the necessary modifications not in the underlying test model (enhanced control flow graph) but straight in the source code. Doing so, results in a unified representation (source code) of the mutant necessity conditions which is common for all the test generation methods. Thus, by employing a source to source compiler that implements the proposed source code modifications, any test generation method, either static or dynamic, can straightforwardly perform mutation.

4. Approach Description

The mutation testing criterion requires the introduction and execution with test cases of a candidate set of mutants. Naturally, the set of mutants compose different program versions that are compiled and executed from the candidate set of test cases. To automate the mutant generation and execution processes, various approaches have been proposed; refer to (Untch et al. 1993) and (Jia and Harman 2010) for further details. The present approach achieves to automate these phases in an efficient way using an innovative integration of mutant schemata and weak mutation approaches. Experiments with the proposed scheme suggest that considerable execution savings (see performance results, section 6.1) can be gained, making mutation testing feasible for practical use. Additionally, the propositions made in this paper go a step further by effectively automating the generation of mutation based test cases in a generic way. That is, the ability to assist any test generation method for structural testing to produce tests for mutation. This constitutes a major achievement as mutation based test case generation approaches are quite limited; see section 8 for relevant approaches.

4.1 Introducing and Executing Mutants

Automating the mutation testing process requires the generation and execution of the mutant programs. A naïve approach is to produce those mutant versions, compile them and run them against the test cases. To reduce these overheads, mutant's compilation, the mutant schemata approach was proposed by Untch et al. (Untch et al. 1993). In their work the Mutant Schemata Generator (MSG) system was proposed in the context of strong mutation. This approach was later applied to the MuJava (Ma et al. 2005) tool for performing strong mutation in the Object Oriented context. It is this approach that was extended by the present research in order to automate the mutation analysis process.

4.1.1 Mutant generation

Traditionally, in the strong mutation context, the MSG approach encodes all the candidate mutants of a program into one meta-program. This is a special parameterized program that applies one mutation at runtime based on a global parameter. This is achieved by placing static or dynamic schematic entities, according to the considered mutant operators. The meta-program is produced based on a special source to source compiler and compiled from the same compiler as the original program. Thus, only one compilation (of the one meta-program) is needed and also the meta-program execution can be performed at compile-speeds (Untch et al. 1993). The global parameter acts as a switching on and off parameter for each mutant. Mutant execution is performed by executing the meta-program with the candidate tests by setting appropriately this global parameter. This requires a special test driver able to handle the test execution process and mutation score evaluation.

To make clear how the approach works consider the example program "Foo" of figure 1. On the left hand side of this figure the original program is presented while on the right hand side the produced schematic-program is given. For example the statement
if (y < 0) can be mutated by the following mutants:

if (y ≤ 0)

if (y > 0)

if (y ≥ 0)

if (y == 0)

if (y != 0)

if (true)

if (false)

All these mutants are embedded inside the RelationalLT schematic function. The "Mutants" groups of Figure 1 represent the introduced mutants of each statement which are embedded inside the schematic functions. Dynamically calling the schematic functions results in a dynamic introduction of a specified mutant or its original function. The global parameter defines which one of the mutants, related to the called function, will be applied. Conclusively, all the reachable "Mutant" groups form a test execution path that will be called and all original expressions will be returned except the one mutant application point which will result the mutant expression.

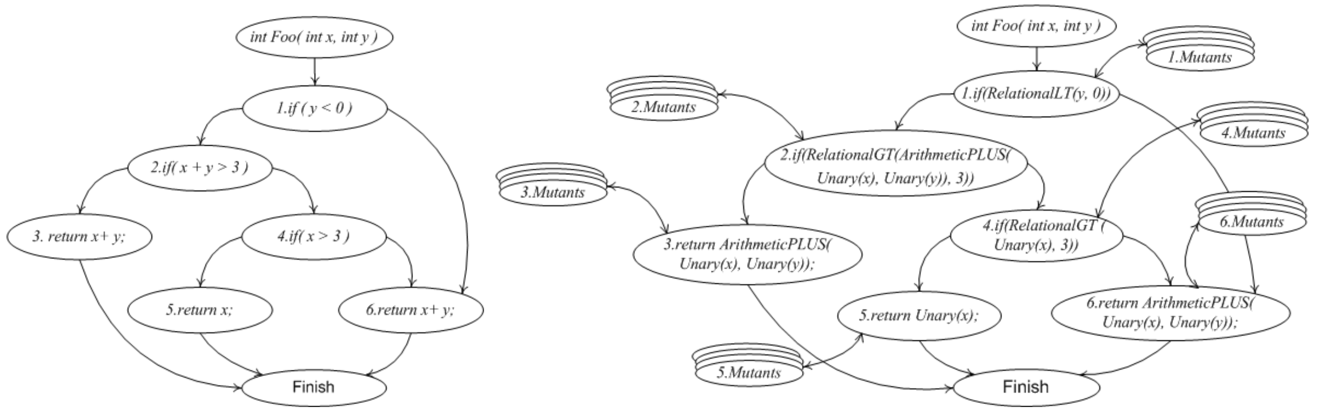

Figure 1. Mutant schemata example

The proposed approach expands the suggestions of the MSG method in order to take advantage of both the compile-speeds and mutant generation advances. Moreover, by employing weak mutation further execution savings can be gained. Integration of these two techniques has been mentioned by (A. Jefferson Offutt and Untch 2001) but never implemented or studied before. Performing weak mutation based on mutant schemata was originally suggested in (Mastorantonakis and Malevris 2003), which is the approach that was extended by the present research.

The difference between the MSG and the present approach is that weak mutation places the need for mutant evaluations at various program places. In strong mutation there is not such a need as mutant evaluation is performed at the program output action taken by the mutant execution driver. Weak mutation requires evaluating both the original and mutant states at the mutation points. To achieve this, there is a need for placing many internal program evaluations that will decide whether the mutants have been killed (weakly) or not. These evaluations could be placed at various program places (A. J. Offutt and Lee 1994), in the proposed approach it was decided to place them inside the schematic functions. Thus, consider the previous example of the statement *if (y < 0),* (program "Foo" of Figure 1). The produced mutant evaluations are presented in Figure 2.

```
boolean orig = y < 0;
if ( orig != ( y ≤ 0 ) )
if ( orig != ( y > 0 ) )
if ( orig != ( y ≥ 0 ) )
if ( orig != ( y == 0 ) )
if ( orig != ( y != 0 ) )
if ( orig != ( true ) )
if ( orig != ( false ) )
```

Figure 2. Mutants' evaluation example

In Figure 3, an example program graph that contains the eligible mutants is presented. Let us assume that node N of the graph on the left is to be tested with mutation. The proposed approach suggests injecting all the possible mutants (n new nodes (N_M[1] to N_M[n]), where n is the number of all the candidate mutants after node N. Then the control flow of the whole program must be restructured to follow the nodes N_M[1] to N_M[n] and from there to return to node N, in essence at the end of node N, to continue with the initial program flow. The data in memory when entering nodes (N+1) or (N+2), should be identical irrespective of which graph (original or mutated) is being executed. In practice the code of node N will be replaced by an entity executing all the nodes from N_M[1] to N_M[n] and at the end will return to the main program flow, the result from the execution of the code in node N. This will assist in continuing the execution of the mutated program as if no added nodes from the mutation process were present. All checks will occur internally, comparing the result in the program's memory between the original code and the mutated one. Thus, it encapsulates the mutation testing exercise in each schematic function while making it transparent for the succeeding nodes.

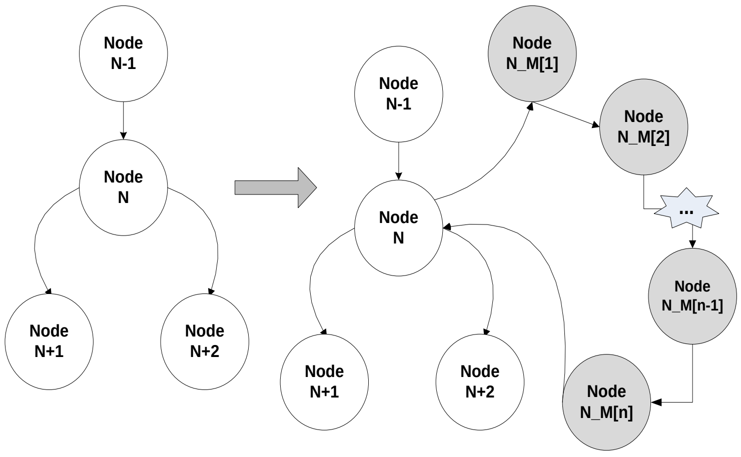

Figure 3. Augmented mutation graph

4.1.2 Mutant execution

Mutant execution according to strong mutation requires a specialized test driver capable of executing, the original and mutant program versions, comparing the program outputs, calculating the mutation score and handling unexpected program behavior caused by mutants. In the MSG approach the test driver executes only the meta-program with the candidate test cases by specifying the mutant parameter. Thus, for each test case the original and mutant programs are executed, their results are recorded and compared in order to calculate the mutation score. Additionally, there is a need to conform to unexpected program behavior caused by mutants, such as the handling of exceptions, the handling of endless loops etc. Endless loops pose the need of monitoring the test execution process in order to decide whether such a situation has occurred or not. A usually undertaken approach, also undertaken in the Proteum (Maldonado et al. 2001) and MuJava (Ma et al. 2005) tools, to bypass this problem is to place an execution time limit beyond which it is assumed that an endless loop has been encountered.

The proposed scheme does not require a specialized driver. This is due to the utilization of weak mutation and the introduction of the internal program checks that record the killed and live mutants at runtime. To make this clear, consider the introduced mutant evaluations of Figure 2. Executing the schematic program with test cases, these decisions (if statements) are executed, a fact that results in the evaluation of the killed mutants. Thus, if the true branches are executed the specific introduced mutants are recorded as killed by the then part of the mutant evaluation decision. Here it must be noted that the introduction of these decision statements does not affect the program state. This suspends the test driver from monitoring and handling undesirable situations caused by mutants.

4.2 Reducing Mutant Execution Cost Based on Weak Mutation

Strong mutation requires the execution of the original and mutant programs and a comparison of the programs' obtained results. Thus, there is a need to execute all live mutants with all available test cases. The execution time is approximately proportional to the number of the introduced mutants. In practice this number turns to be a big burden since it has been quantified as *O*(*Vals* x *Refs*), (A. Jefferson Offutt et al. 1996). Although, mutant schemata have been shown to be beneficial for reducing the test execution overheads, the execution cost of mutation testing still forms one of the most time consuming activities. If mutation is used as a testing criterion, then there is a need for finding suitable test cases. Seeking these tests naturally requires a considerable amount of executions in order to find the sought data.

The proposed scheme requires only one execution per test case in order to execute and evaluate all the introduced mutants. Additionally, as mutants are evaluated at their positions, required by weak mutation, and program execution continues based on the original's program state, mutants do not cause endless loops. Consider the example in Figure 1 and mutant execution with a candidate test case. According to strong mutation, mutant execution requires the execution of the original program and all the candidate mutants against this test. Mutant executions will result in traversing the same with the original program path until the mutant application point is reached, at this point the specified mutant will be applied. From this point, every mutant will probably traverse the same or a different to the original program path due to the mutants' introduction. Traversing a different to the original program path experiences different program behavior which may result in endless loops or other discrepancies. These discrepancies can considerably influence the mutants execution cost as the conducted experiment shows. On the contrary, weak mutation stops at the mutation points so this different program behavior is avoided resulting in reducing test execution time. The proposed scheme goes one step further. This is the dynamic application of the mutants driven by the runtime program execution. Thus, if the candidate test case traverses the "1-6-Finish" execution path, in the original program, then the mutants that are applied are only those belonging to the Mutant groups 1, 6. The rest have no chance of being killed as they will not be executed. Additionally, the proposed scheme does not stop the program execution at the mutation points but instead it continues the program execution by maintaining the original's program state. This gives the advantage of executing all the introduced mutants with only one execution. In the above case, executing the meta-program with the candidate test it will call the respective Mutant groups 1, when the execution reaches node 1, it will apply all the mutant evaluations of this mutant group and continue the execution to the original path by going to node 6. At node 6 it will apply all mutant evaluations of the 6 Mutant group and will continue to the Finish node. Finishing the program execution results in traversing the path traversed also by the original, evaluating all the introduced mutants, with respect to this path, and by experiencing normal behavior of the program under test.

The mutant evaluations should naturally introduce some overheads because of the decisions introduced into the schematic functions. These overheads should be negligible compared to the vast number of executions of the introduced mutants. Aside, this issue also holds for the strong mutant schemata approach (when compared with the separate compilation approach (Ma et al. 2005)), where it was found that executing repeatedly only one program, the meta-program, results in execution savings as there is no need to load and unload different program versions on the computer's memory[2]. Thus, the performance of the proposed approach should result in major execution savings, proportional to the number of mutants and test cases. The experimental evaluation of the required execution time is performed in section 6.1, where speedups up to 10,000 times can be recorded, as the conducted experiment shows.

4.3 Generating Mutation Based Test Cases

The proposed approach attempts to automate the test data generation process according to the mutation testing criterion based on existing test case generation techniques and tools. To achieve this, the source code of the program under test must be transformed appropriately. The main idea was to use the program transformations of mutant schemata in order to introduce the mutant necessity conditions in the program under test (see section 3). These conditions should then guide the test generation process accordingly. Mutant necessity conditions can be easily formed as branch conditions and so, if they are embedded into the suitable locations of the program under test, can be easily tackled with structural testing tools. Such an approach was originally introduced in (Papadakis and Malevris 2009) in the context of constructing a suitable test model for use with symbolic execution and it was later generalized in (Papadakis et al. 2010) in the context of using existing structural testing tools and techniques to perform mutation.

In (Papadakis and Malevris 2009) the foundations of producing certain constraints under which mutants are killed as proposed by DeMillo and Offutt (Richard A. DeMillo and Offutt 1991), are used in order to construct a suitable model for test generation. This test model, called Enhanced Control Flow Graph (ECFG) (Papadakis and Malevris 2009), forms a graph embedding into its arcs all the considered mutant constraints. By doing so, covering the ECFG branches results in covering - killing all the considered mutants. Thus, following this approach an automated tool that interfaces with a suitable ECFG (Papadakis and Malevris 2009) can produce mutation tests. Although this approach can reduce the mutant killing problem to a covering branches problem, existing automated tools constructed for structural testing, cannot be used straightforwardly. Hence using existing automated tools requires complex adaptations to their embodied generation engines in order to produce mutation tests. Such adaptations are based on the bilateral embodiment of the actual program execution graph and the enhanced model (ECFG).

Overall, in order to produce mutation based test cases there is a need for representing and satisfying all the mutant killable conditions according to the special characteristics of each utilized method. Dynamic approaches gain the required information through actual program execution and thus, mutant killable conditions representation must follow these lines. Additionally, the representation of the mutant killable conditions is an application-technique dependent task. The proposed approach answers the above problem in a generic way, by altering the program source code using the mutant schemata technique. Thus, by representing the mutant killable conditions at the code level a common basis for all the test generation methods is established.

[2] This conclusion was based on the Java programming language

4.3.1 Mutant necessity condition representation

The proposed approach represents mutant necessity conditions in a generic way in order to make use of the existing structural test data generation approaches. The innovative idea behind this research is the production of one meta-program that includes all candidate mutants into its structure. The structure of the meta-program is along the same lines as the ECFG and thus reduces the mutants into branches. Interfacing this meta-program with an automated tool or technique able to generate tests for structural testing, can effectively produce mutation testing tests for the original program. It is the weak mutation system described in previous sections that is suitable for the test generation purpose too. Thus, by introducing weak mutation mutants, these can formulate the basis for test generation methods.

Automating the test case generation requires specific, to the technique used, information about the target test requirements. Mutation poses difficulties in producing this information (i.e. killing the mutants) as its requirements are spanned across different program versions (one mutant per version). Hence, the proposed scheme unifies the mutation requirements in a suitable way to be used by test generation techniques, such as symbolic execution or a search based application. By doing so, the candidate mutants are concentrated in a unique representation rather than being spread to one application per mutant. Expanding the suggestions of the MSG approach, the evaluation of the mutants' execution is performed within the schematic function, as described before. This implies an indirect reduction to a path - branch coverage problem of the mutated programs. Figure 2 presents an example of the reductions into branches made by the proposed approach on the relational mutants through the mutant's evaluations. By placing the mutant evaluations into the schematic functions, the suitable conditions under which the considered mutants are killed are also embedded. These conditions are formed as decisions, into the schematic function, containing the following expression:

$$\text{Original expression} \neq \text{Mutated expression} \qquad (2)$$

This expression has been used by (Richard A. DeMillo and Offutt 1991) in order to produce mutant necessity constraints. The joint representation and satisfaction of both the reachability (represent the conditions from program input to the schematic function) and necessity conditions (expression (2)) is sufficient for producing tests capable of killing the introduced mutants.

The above decision expression (2) has two possible outcomes (the original is either equal to the mutated or not). Thus implying the introduction of true (mutant is killed) and false (mutant is alive) branches to represent the possible outcomes. Further, by unifying both the reachability and necessity conditions, by representing both of them as program decisions, at the code level any technique or tool can effectively represent them. This suggests that an effective technique or a tool aiming at exercising program branches should now target on mutants effectively.

4.3.2 Mutant necessity condition satisfaction

Generating tests that cover specific program paths or branches requires the satisfaction of specific requirements related to the utilized test generation method. For example symbolic or concolic execution requires the selection, representation and solving of the path condition of a feasible program path, while the search based approaches require the selection of an appropriate fitness function etc. Satisfying the mutant necessity requirements can be effectively achieved based on the covering branches approaches by reducing mutants to branches as presented in the above section. Figure 3, illustrates in the program's graph the actual schematic modifications of the program's source code when applying the mutant schemata method. It is noted that every mutant node (N_M[1], ..., N_M[n]) contains inside its structure the evaluation of killable mutants according to expression (2). Any tool that uses symbolic execution or concolic execution can reproduce and solve the meta-program's path conditions that embed all the suitable conditions under which mutants are killed. Search based approaches are capable to guide the generation process through mutant branches and effectively kill them, by taking into account both the reachability and necessity conditions. Besides theory, the undertaken experiment of section 6 reveals the feasibility of this approach based on three automated tools utilizing, symbolic, concolic and search based testing approaches.

5. Mutation based Test Data Generation utilizing existing Structural based Tools and Approaches

This section introduces three different in philosophy test data generation approaches and their implementation for the structural testing of Java programs. An automated framework that performs mutation testing based on the propositions made in the previous section and by utilizing these three automated tools is also proposed. Here it must be noted that any technique or tool capable of producing structural tests could be also employed without requiring any special modification or treatment.

5.1 Symbolic Execution

The symbolic evaluation process (J. C. King 1976) of a program consists of assigning symbolic values to variables in order to deduce an abstract algebraic representation of the program's computations and representation. This technique is based on the selection of paths from its control flow graph and the computation of symbolic states. The symbolic state of a path forms a mapping from input variables to symbolic values and a set of constraints called path conditions over those symbolic values (J. C. King 1976). Path conditions represent a set of constraints called symbolic expressions that form the computations performed over the selected path. Solving the path conditions results in test data which if input to the selected path, the path will be executed. If the path condition has no solution the path is termed infeasible.

In the present paper a symbolic evaluation system known as symbolic execution extension of the Java PathFinder (Visser et al. 2004), (Păsăreanu et al. 2008) (JPF-SE) was used. In JPF-SE symbolic execution is performed by initializing the input variables, thus supporting complex data structures. The basic function of JPF-SE is to direct JPF to validate the various paths contained in the symbolic execution tree. This is done in an exhaustive way using a depth first or breadth first strategy. Whenever a new branching point is reached, the path condition is updated by checking it for satisfiability using an appropriate decision procedure. If the path condition is unsatisfiable, the system backtracks to a previous satisfiable point according to the strategy taken. By doing so, all feasible paths are thus explored. In the present work, JPF-SE was assisted to produce test cases according to branch testing by selecting a subset of the produced test cases able to cover all (covered) program branches. The default decision procedure of the JPF-SE, namely Choco which is a constraint solver for Java was used in combination with the default exhaustive exploration of the symbolic execution tree.

5.2 Evolutionary Testing

Testing techniques based on genetic algorithms try to mimic the natural evolution and use it as a search engine in seeking for suitable tests. The present framework integrates the technique proposed by (Tonella 2004) for evolutionary testing of Java classes. According to this technique tests are encoded into chromosomes as method sequences and their respective parameter values for a class object of the class under test. Test evolution starts by setting as objective targets the program branches. Each of these targets-branches is considered in turn until it is covered or the search reaches a predefined upper bound limit (time or number of evolutions). The initial population of tests is produced at random. These tests are executed in order to determine if the targeted branches have been covered, if so, the tests are saved and the search continues to the remaining uncovered ones. If the produced tests fail to cover the target branches a fitness value is calculated according to each test. The fitness value is computed as the ratio of the covered control and call dependence edges over those of the target branch. New tests are produced considering previous ones with higher fitness values by transforming them based on crossover and mutation operations. These operations (crossover and mutation) form a set of predefined modifications to the chosen tests, such as insert, delete and alter method invocations and method parameters.

The above technique has been implemented into an automated tool called Etoc (Tonella 2004). Etoc’s primary target is the production of test cases according to the branch testing criterion. Although this tool has been shown to be quite powerful for testing Java classes, its main purpose is to generate program method sequences able to test state related behavior encapsulated by objects under test. Thus it fails to produce tests aiming at complex non state dependent branch conditions inside methods. Targeting on these conditions should employ techniques such as the ones reported in (Lakhotia et al. 2010), (Harman and McMinn 2007). Despite its limitation this tool has been used in our case study for illustration purposes only. Consequently, any other similar tool that serves the purposes of evolutionary testing can be used.

5.3 Concolic Execution

The Concolic testing (Concolic execution) method (Sen et al. 2005) forms a combination of actual and symbolic execution. According to this method, when actual execution takes place, symbolic constraints are collected, constructing the path condition of the executed path. It then uses this path condition in order to drive the execution towards different program paths. This is achieved by negating one condition of the predicates in the path condition. The advantage of this approach is that complex and unhandled expressions can be resolved by the actual execution by replacing or simplifying them with the actual values encountered during the execution. Additionally, the test generation process can always backtrack to random testing when facing problematic situations.

The process starts with random or user defined inputs. These produce program traces that form both the execution path and its respective path condition in a simplified form (simplifying unhandled expressions). The process then iteratively negates and solves all path condition’s predicate expressions, each one in turn starting from the ultimate one to the first one. New inputs are produced which hopefully follow different execution paths. Ideally, if all expressions can be handled, the process can continue until all program feasible paths have been executed. In practice this is limited by the underlying decision solving procedures, the number of the encountered paths and the programming constructs encountered. Further, the same process may restart many times in order to reach a required coverage goal. Conclusively, the process performs structural testing targeting on all feasible program paths (Sen et al. 2005). In the present paper a prototype tool that implements the above process has been constructed and used in the proposed framework for performing mutation. The application of the prototype was assisted to produce test cases according to branch testing by selecting a subset of the produced test cases able to cover all (covered) program branches. Currently the prototype has some limitations such as the handling of dynamic program inputs, method sequences and floating point arithmetic. The handling of method sequences is performed based on either user defined sequences or on a random generation approach.

5.4 Proposed Framework that Assists Existing Automated Tools to Perform Mutation

The suggestions made in this paper resulted in developing an automated framework that performs mutation by utilizing existing structural automated tools. The framework's effectiveness depends on the effectiveness of the underlying test data generation tools. At present, the framework produces the meta-program automatically and this is forwarded to a test generation tool that produces the required tests. An overview of the proposed framework structure is presented in Figure 4. For the paper's purposes the present study uses the symbolic execution extension of the Java PathFinder tool, the Etoc tool and a Concolic execution tool. Although these tools may not be the most appropriate and effective ones, they were chosen because of their availability and the different underlying techniques they implement. However, it is believed that they serve the general goals of the present study, which is to provide a general way of automating the mutation test data generation activity. Additionally, the chosen tools help to study the applicability and effectiveness of the proposed approach. A secondary purpose also served, was to allow practitioners to use their own tools for performing mutation.

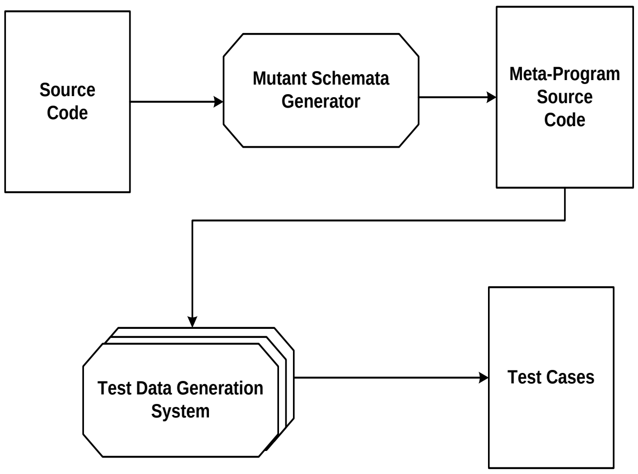

Figure 4. The proposed framework

The proposed framework (figure 4) initially takes the test subject code written in Java as input and automatically generates the sought test data. To achieve this, the programs' source code is transformed and a schematic meta-program is produced. This schematic program contains all the killable conditions required by the introduced mutants into its structure formed as branches (see section 4). The schematic functions were designed based on the method's description (Untch et al. 1993) and by placing the mutant evaluations inside the schematic functions. Thus, mutant evaluations were forced to be applied at the mutants execution points, similar to the mutant necessity conditions applied by (Richard A. DeMillo and Offutt 1991). Additionally, the framework replaces all logical (short circuit[3]) operators with their respective Boolean logical operators (non circuit) in order to conform to the mutant schemata behavior[4]. If such a replacement causes any discrepancies to the program behavior then the logical operators should be omitted or tested separately. Then, by utilizing a test data generation engine able to cover these branches can also cover-kill the mutants. This activity can be performed by any automated tool aiming at structurally testing Java programs. In the present study three automated tools were used. These tools were chosen because of their availability and the differences in their philosophy for generating data. Two of them are publicly available. The first is known as the symbolic execution extension of the Java PathFinder tool (Păsăreanu et al. 2008) and the second as Etoc (Tonella 2004), an evolutionary based testing tool for Java. The third one utilizes the "Concolic" execution method (Sen et al. 2005) and it was implemented by the authors for the purpose of the present paper. Finally, these tools were assisted to produce test cases according to the proposed framework. The framework's application succeeded in assisting these tools to kill the majority of the introduced mutants, as the following experiments show.

6. CASE STUDY

The aim of this research is to provide practical solutions upon automating the mutation testing activity. To this extent, the propositions made in the previous sections are assessed on a set of programs, extensively used in mutation testing research. This section examines the following issues: a) the performance of the proposed scheme with respect to the required test execution time and b) the ability of the proposed scheme to guide effectively existing test data generation tools to perform mutation. Weak mutation was proposed as an alternative method to strong mutation in order to reduce the required execution time. Thus, the performance of the proposed scheme was compared with the "traditional" MSG approach for strong mutation in section 6.1. Additionally, the ability of the proposed scheme to assist test data generation tools to produce mutation based test cases is also considered by comparing the number of the killed mutants with and without the use of the proposed framework in section 6.2.

6.1 Performance Results

Weak mutation was originally proposed as a mutation variant technique able to reduce mutation testing execution cost. As pointed out in the previous sections, mutant execution time consumes the major part of the whole time spent during the mutation analysis process. Thus, reducing these expenses, mutation performance can be significantly improved. The question that is examined in the following experiment is how well the proposed mutant schemata approach performs in comparison to the "traditional" MSG approach for strong mutation. Previous studies (A. J. Offutt and Lee 1994) show a reduction of 50% of the mutant execution required time. The proposed scheme, as discussed in section 4, employs weak

[3] Short circuit operators (logical) execute or evaluate the second argument of a Boolean expression if the first argument does not suffice to determine the value of the expression.

[4] Schematic functions take both expression operands as arguments and thus, they execute or evaluate the second argument of the expression at the schematic call point.

mutation in order to efficiently execute the introduced mutants taking advantage of the program structure. By taking this approach only one execution run per test case is needed in order to execute all the introduced mutants, while strong mutation requires one execution of the original program and one additional per live mutant for each examined test case. Thus, major execution time savings can be gained.

To answer the above question experimentally, a set of mutation benchmark programs were chosen and a performance comparison of the two methods was made. The comparison was based on the same schematic programs and test execution mechanisms for both of the approaches. By using two different sets of schematic functions (one for Strong and the other for Weak mutants) a straightforward comparison can be made. This is due to the use of the same mutant sets and test cases for both systems. Here, it must be noted that a specialized driver using the same reflection mechanism was used for both of the examined methods. In order to perform a fair evaluation between the two methods, the comparison was undertaken on a typical case of testing where, whenever a mutant is killed it is discarded from the mutant set and thus it is not executed against the rest of the available test cases. In this case the mutant execution cost per test is decreased as more tests are executed while killing additional mutants. In the opposite case weak mutation will achieve considerably more speedups. Here it must be noted that the equivalent mutants were not discarded from the mutant set in order to provide a typical testing situation. It is these mutants that influence most the mutant execution time, as by their definition they cannot be killed and thus, they will be executed against all used test cases.

Tests were created to kill almost all[5] introduced weak and strong mutants for each of the subject programs. Table 1 records details about the selected programs and the chosen set of test cases. The selected programs were chosen from the mutation benchmark programs used by Polo et al. (Polo et al. 2009) and from the Siemens program suite[6] (Harrold and Rothermel). The column “Test Subject” records the names of the selected programs while the “Number of Mutants” and “Number of test cases” record the number of the produced mutants and the number of the chosen test cases of each chosen program respectively.

Table 1. Programs for comparing Weak to Strong mutation execution times

Test Subject	**Number of Mutants**	**Number of test cases**
Trityp	352	33
FourBalls	214	11
Mid	163	19
Find	201	11
Tcas	298	51
Replace	1158	52
Schedule	248	9

[5] For the Siemens programs the selected tests were chosen among those that kill all the killable mutants by the available test suite. Thus, they might not kill all the killable mutants.

[6] Siemens programs were rewritten in Java for the purposes of the experiments

Test execution times were measured from the beginning of the mutant executions until the end of executions including the mutation score evaluation. To avoid any side effects from the utilized machine and operating system the experiments were independently repeated 10 times and record the average required times among the 10 repetitions. Test execution results are recorded in Table 2. In Table 2 the execution times were measured in milliseconds and the speedups based on the following relation that represents the amount of how many times weak mutation runs faster than strong mutation.

$$Speedup = \frac{Strong\ Mutation\ Time}{Weak\ Mutation\ Time}$$

From these results it can be observed that major execution savings can be recorded using the proposed schemata approach. The achieved speedup ranges differ considerably from objective to objective with a least achieved speedup for weak mutation to be 4.79 times faster than the reciprocal for strong. In the best case (Replace program) the weak mutation approach runs 10294.10 times faster than strong mutation.

Table 2. Programs for comparing Weak to Strong mutation execution times measured in milliseconds.

Test Subject	**Weak Mutation**	**Strong Mutation**	**Speedup**
Trityp	28.73	2673.56	93.04
FourBalls	24.97	2130.58	85.33
Mid	19.65	1616.65	82.26
Find	21.65	1603.71	74.08
Tcas	27.78	133.01	4.79
Replace	110.42	1136639.62	10294.10
Schedule	1476.58	13828.74	9.37

The variation in the required test execution times is due to the utilized program structures, introduced mutants and program execution behavior. An interesting point observed in the Replace program was the unexpected behavior of the various introduced mutant programs with respect to strong mutation. Recall that with the proposed approach (incorporating weak mutation) the test execution of mutants experiences the same behavior with the original program avoiding unexpected behavior. In the case of Replace, many mutants were killed due to timeouts[7], as mutants caused endless loops thus hitting the timeout mechanism. Further, the Replace program contains a high number of mutants, which is approximately three times bigger (see Table 1) than the number of mutants contained in the other programs. Additionally, many mutant programs required much more time to finish than the original program. Whence, the huge discrepancies occurred between the execution times of Strong and Weak mutants in the Replace program.

[7] In the experiments a timeout of 3 seconds was employed in order to overcome the problems caused by the endless loops

6.2 Test Generation Results

To perform an initial assessment of the applicability and feasibility of the framework, it was applied to a set of Java program units. The selected programs were chosen from a) the testing textbook website by (Ammann and Offutt 2008), b) from the examples distributed together with the JPF-SE tool (Păsăreanu et al. 2008) and c) from the mutation benchmark programs used by (Polo et al. 2009). Table 3 presents the number of the candidate mutants and the number of produced equivalent mutants per each program.

Mutant schemata were generated based on the mutation operators set proposed by (A. Jefferson Offutt et al. 1996). As the current approach targets on weak mutation, the unary mutant operator, which inserts unary language operators (i.e. decision negation, unary increment and decrement), was excluded as by definition it will always be weakly killed at the execution point by any test that executes it. This is also pointed out in (Richard A. DeMillo and Offutt 1991) for the construction of the mutant necessity conditions templates. Thus, four operators were used (i.e. ABS, AOR, LCR and ROR; details of these operators can be found in (K. N. King and Offutt 1991)) for the purposes of the present case study. In addition, any other mutant operator can be also used by defining and embedding its mutant schemata into the produced schematic meta-program. All equivalent mutants were detected by manual analysis in order to accurately calculate the mutation score achieved. Test cases were then derived based on the three employed tools (i.e. JPF-SE, Etoc and Concolic prototype as these were described in section 5), aiming at producing test cases according to branch testing, utilizing the original programs first and then the schematic ones.

Table 3. Programs for the test data generation process

Test Subject	**Number of Mutants**	**Number of Equivalent Mutants**
Trityp : J1	352	92
FourBalls : J2	214	39
Mid : J3	163	4
Find : J4	201	53
Bubble : J5	93	21
Cal : J6	330	62
TrashAndTakeOut : J7	117	11
PrintPrimes : J8	103	28
BankAccount : J9	69	6
BST : J10	94	3

Experience

This section reports results from the application of the framework to the selected test subjects. For the needs of the case study, the framework first generates three meta-programs, embodying the peculiarities of each of the utilized tools. Then the incorporated tools together with the meta-programs were used aiming at program branches, according to their normal functionality. The automation level depends solely on the test data generation tool and it is independent of the mutation evaluation process. As all the incorporated tools use different approaches for automating the test generation process, the case study highlights the general character and the simplicity of the proposed schemata technique utilized by the framework. In order to reinforce the power of the schemata technique (for generating tests for mutation) three different automated tools were used. This also shows the ability of the framework to host any other automated tool in a similar fashion. Additionally, the framework can be used as a yardstick towards the use and development of more powerful and specialized tools for mutation testing.

Table 4 records the initially achieved branch coverage and mutation score by the utilized tools, when used for branch testing. It must be noted that the tools were used for the purpose of obtaining test data for the original program version (ordinary use) without including the mutant schemata. Then, these test data were driven to the mutated programs containing the mutant schemata for killing the mutants. The results indicate that all three tools are effective for covering program branches (the majority of the branch coverage scores are 100%) but not very effective for killing the mutants. This was somehow expected as the tools were used for branch testing in a crude way. The variations of the scores among different test subjects are purely due to the internal characteristics of the test subjects, the methods themselves and their respective implementations.

Table 4. Initial mutation score achieved by the employed tools

Test Subject	**JPF-SE**		**Concolic**		**Etoc**	
	Branch	Mutation	Branch	Mutation	Branch	Mutation
J1	100%	87.69%	100%	86.54%	75%	85.38%
J2	0%	0.00%	60%	44.57%	100%	73.71%
J3	100%	50.31%	100%	62.26%	100%	63.52%
J4	100%	91.89%	100%	90.54%	7%	7.43%
J5	100%	91.67%	100%	91.67%	70%	87.50%
J6	0%	0.00%	100%	73.51%	100%	80.22%
J7	100%	87.74%	100%	87.74%	100%	73.58%
J8	100%	98.67%	100%	98.67%	100%	94.67%
J9	100%	68.25%	100%	66.67%	100%	74.60%
J10	80%	43.96%	80%	46.15%	100%	62.64%

In the next phase of the case study the programs containing the mutant schemata were used as input instead of the original ones. Performing structural testing to these programs directs testing on killing the introduced mutants. In Table 5 the results obtained by employing the selected tools with the assistance of the schematic program are presented. These results record a similar, with branch testing, branch coverage level and a high mutation coverage level, for all three employed tools. The comparison of the results obtained in Tables 4 and 5 shows that the application of the suggested approach within the proposed framework produces much better results when the mutants' schemata are embodied in the test subjects. This should be regarded as an achievement of the proposed framework as it is flexible to adopt the characteristics of the mutants. A pictorial representation of the values of Tables 4 and 5 is given in Figure 5, where the lower part (dark grey) of the bar charts displays the percentage of mutants killed per test subject per method used when applying the three tools aiming at covering the branches. The values obtained are in essence the collateral weak mutation coverage achieved when testing based on structural testing as performed by each utilized tool. On the contrary, the upper additional parts (light grey) indicate the maximum achieved coverage when the mutants are represented as branches and this makes the differences significant since using the three tools for branch testing, killing the mutants is a direct activity of covering their mutant-branches. In fact, for one of the programs, J4, an almost 59% increase is recorded with Etoc, while for 6 of them full coverage was achieved (killing all mutants) involving all three methods, whereas this is not true for the straightforward initial tool usage. Hence, the discrepancies are in favor of the proposed approach indicating its strength in killing the introduced mutants.

The obtained results, clearly suggest the differences in the testing quality, measured by the achieved mutation scores, when performing branch testing on the original and the schematic program versions. Additionally, the automation level of the mutation testing can be improved by utilizing branch testing tools. Another interesting matter that can be observed in Figure 5 is that the proposed technique almost always delivers an improved mutation score compared to what achieved by the original program versions. This fact is also true for all the utilized tools and reveals the strength of the proposed technique. Besides its effectiveness, the proposed approach can be used incrementally in order to establish confidence on specific program units. That way, a testing strategy that aims at program branches can first be used in order to fulfill the minimum testing requirements and then directly target on mutants using the same tools and techniques in order to lead the testing process to a higher level of thoroughness.

Table 5. Mutation score achieved by the employed tools based on the proposed framework

Test Subject	**JPF-SE**		**Concolic**		**Etoc**	
	Branch	Mutation	Branch	Mutation	Branch	Mutation
J1	100%	98.85%	100%	99.23%	75%	97.69%
J2	0%	0.00%	80%	68.00%	100%	77.14%
J3	100%	100.00%	100%	100.00%	100%	73.58%
J4	100%	96.62%	100%	97.97%	60%	66.22%
J5	100%	94.44%	100%	94.44%	80%	90.28%
J6	0%	0.00%	100%	80.97%	100%	92.54%
J7	100%	95.28%	100%	99.06%	100%	83.02%
J8	100%	98.67%	100%	100.00%	100%	100.00%
J9	100%	85.71%	100%	100.00%	100%	84.13%
J10	80%	71.43%	100%	100.00%	100%	80.22%

Finally, it must be noted that this study forms the first step towards automating the test data generation activity for mutation testing. The present case study focuses on revealing the effectiveness of the proposed framework in guiding existing tools to produce high quality tests. Despite the fact that the employed tools may not be the most appropriate ones, as they cannot focus on specific program branches, handle specific program constructs and characteristics, the proposed framework effectively assists them to achieve high mutation scores. The employment of only these three tools was imposed by the limited availability of similar automated tools. In any case, the scope of the present research was not to compare and record the performance of the automated tools. The purpose of the present study was to show that mutation testing can be effectively performed by employing a category of tools that perform another type of testing such as branch testing.

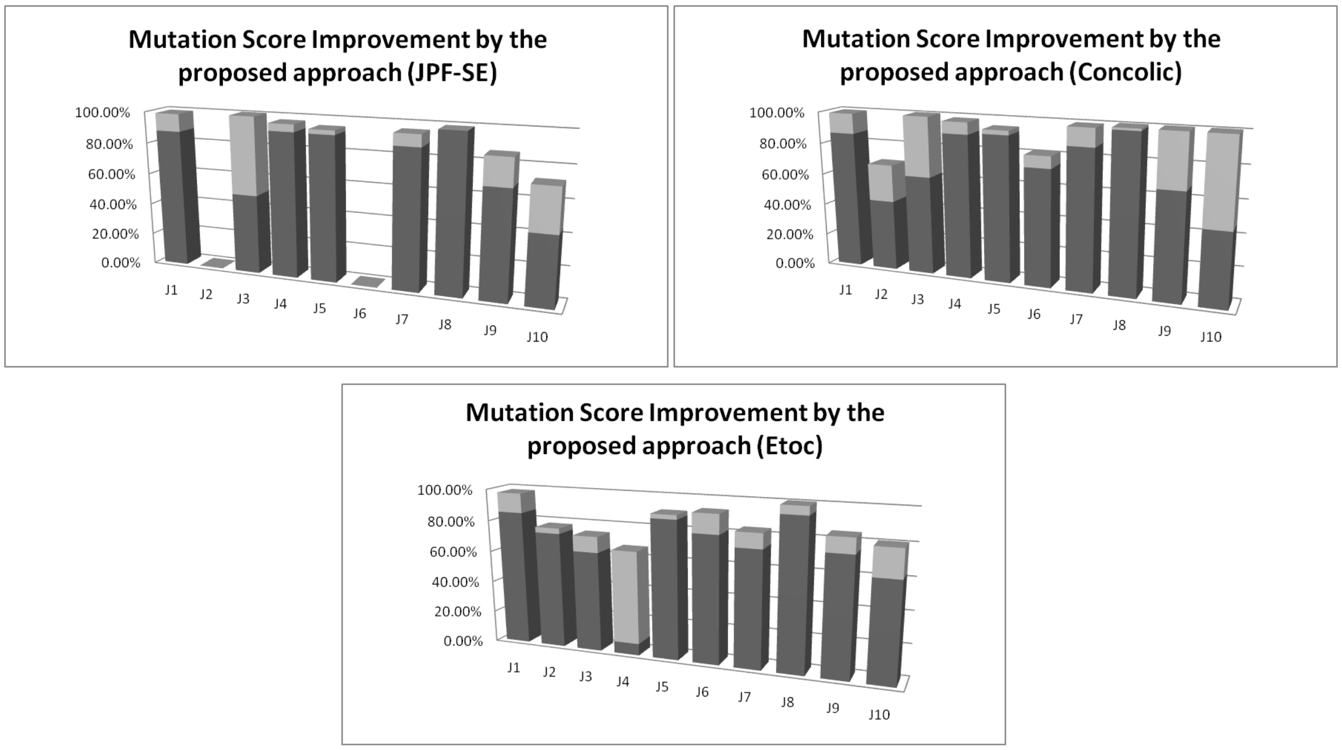

Figure 5. Improved mutation score by the proposed approach. The dark grey color indicates the achieved mutation score when performing branch testing with the employed tools. The light grey color indicates the improvement on the mutation score achieved when performing branch testing on the schematic programs.

6.3 Threats to Validity

The present paper focuses on automating mutation testing using existing techniques and tools. One possible threat to the validity of the results reported here may be related to the generalization of the obtained results. Thus, the framework's effectiveness may vary in other cases. However, the main objective was to present the system's practicality, feasibility and application to a set of programs with respect to its ability to assist existing automated tools to perform mutation rather than comparing the effectiveness of the various test generation approaches. In view of this, the results obtained may serve as a yardstick towards the employment of mutation testing in an automated fashion and in order to indicate that it is possible to adopt mutation for the testing activity, resulting in the production of high quality tests. The last threat of internal validity maybe based on the use of software systems. Thus, possible bugs, time measures, time limits, program conversion and differences of the test objectives from C to Java may have also affected the obtained results. Additionally, the employment of the utilized tools in order to perform branch testing (for details see section 5) may have also influenced the obtained results.

7. LESSONS LEARNED

7.1 Discussion

The proposed technique forms the first step towards automating the generation of test cases according to higher level criteria such as mutation. The technique suggests a novel and practical way of reducing the mutation testing problem to a well studied one such as branch testing, in order to effectively apply existing automated methods for this criterion. This also constitutes one of the very few attempts of adopting recent advances of the test data generation such as symbolic execution , concolic execution (Sen et al. 2005) and search based testing techniques (Harman and McMinn 2007) for performing mutation testing. As far as the search based techniques are concerned, various improvements can be made to improve their effectiveness. For example, the implementation of the fitness function suggested by (Bottaci 2001) can be straightforwardly implemented by measuring the branch distance of mutant constraints.

Generally, dynamic approaches rely on the actual execution of the program under test. In order to be effective they often require a dramatically huge number of execution cycles. This problem is intensified under mutation testing which produces a vast number of mutants. The attempt to kill the mutants by executing them, in combination with the required excessive execution cycles results in exhaustive computational resources while being time consuming. An advantage of the proposed technique is that by employing weak mutation in comparison to strong mutation the amount of time and therefore the overall effort can be significantly reduced. In previous studies (A. J. Offutt and Lee 1994) a reduction of approximately 50% was recorded with the price of losing approximately 10% of mutation score. Here, the use of mutant schemata integration with weak mutation results in significantly more savings than the ones reported in (A. J. Offutt and Lee 1994) (see the conducted experiment section 6). These results suggest that it is possible to efficiently automate the whole mutation testing process. Further, based on the conducted experiments, the feasibility and applicability of the proposed technique, by using structural testing tools, on mutation based test data generation has been shown. It is the lack of mutation tools that gave rise to this idea for generating test data to perform mutation based on the existing ones. Further, as the current trend in software testing research is to build more powerful test data generation techniques and tools i.e. (Baresi et al. 2010), (Harman and McMinn 2007), based on the proposed approach, these techniques can be straightforwardly utilized to perform mutation.

Mutation based test case generation approaches are quite limited (see related work, section 8). This is due to the mutation testing inherent difficulties. These are the difficulty of representing and satisfying the mutant’s killable conditions, the general difficulty of automatically producing test cases even for the low level testing criteria in the criteria hierarchy and the handling of the mutation analysis process. The proposed approach in this paper gives a possible answer to all these factors. It achieves the representation of all the mutants’ killable conditions, which are spanned across the various mutant program versions, in a generic and unified way, the schematic program. This unification helps to facilitate any existing or possible future tool aiming at structural testing to perform mutation. Thus, solutions to the general problem of test generation can be effectively applied to mutation. Finally, the proposed scheme achieves the reduction of the mutation analysis overheads related to the mutant production and execution. As the present experiment shows, a speedup of even 10,000 times can be achieved giving rise to the argument that mutation testing can be applied in practice.

The proposed framework uses a variant of weak mutation for guiding and evaluating its test production process. This does not necessarily mean that the results here can be extrapolated to include strong mutation or any other mutation variant too. In any case, this was not the intention of the present framework, as it aims at a weaker testing level than strong mutation. It must be noted that the effectiveness of mutation and therefore its variants too, as testing methods are not challenged in this paper. Considering existing mutation based test case generation approaches it can be argued that all of them, see section 8 for a discussion on existing approaches and the present work, are based on the foundation established by DeMillo and Offutt (Richard A. DeMillo and Offutt 1991). Therefore, they all try to reach and infect the program state; in fact they try to satisfy the reachability and necessity conditions, according to the introduced mutants based on techniques originally proposed for branch testing. The effectiveness of these approaches depends on the utilized test generation engine and its underlying method. Thus, the present approach forms a generalization of them as it targets on the same conditions (reachability and necessity conditions) and can be also applied based on all branch testing approaches. In order to target on strong mutation there is a need for handling the mutant sufficiency condition. This condition can be approximated by exploring the mutant program path space (Papadakis and Malevris 2009), (Papadakis and Malevris 2010a); a similar approach was also suggested in the (Richard A. DeMillo and Offutt 1991) work. Such an approach can be implemented extending the present paper's framework by employing a special form of mutant schemata for strong mutation that also embeds the mutant evaluations inside the schematic functions (Papadakis and Malevris 2010a) and a specialized driver able to assist the test generation tool to effectively search the mutant program path space.

Finally, it must be pointed out that the reported results serve the goal of presenting the improved effectiveness of the tools when they aim at the program branches on the original and the meta-program versions. The results must not be seen as a comparison between the utilized test data generation techniques. Such a comparison may not be valid due to the following reasons. First, these tools form different independent implementations of the utilized techniques and thus, their effectiveness reflects more implementation issues and not the methods themselves. Second, the tools were not utilized or assisted equally, using optimal time or exploration limits. Their use was based on an intuitive way to maximize the effectiveness with respect to branch testing. Recall that the mutation scores measured are the collateral coverage achieved when aiming at program branches on both the original and the schematic programs. Third, the utilized tools do not target directly at program branches but indirectly making them effective or ineffective in specific occasions. The JPF-SE and the Concolic prototype target on all feasible program paths based on which they select the candidate branch tests. The Etoc tool targets on program branches by aiming at method sequences, fact that makes it more effective in state dependent methods and classes, while it is less effective in covering internal method structures. The Concolic tool operates based on random method sequences and inputs, while the symbolic tool operates based on an exhaustive sequence exploration. For all the above reasons, the authors believe that the performed comparison should be seen as a comparison of the achieved mutation scores of each tool when aiming at program branches on the original and the schematic programs.

Comparison between mutation testing and other testing criteria such as the structural ones has been extensively studied in the literature. Specifically, it has been shown that weak mutation subsumes most of the structural testing criteria such as the statement, decision, modified condition / decision and multiple condition coverages (J. Offutt and Voas 1996), (Ammann and Offutt 2008). Empirically, comparing the statement, decision coverage and mutation testing criteria have been also attempted by (James H. Andrews et al. 2006), (Smith and Williams 2009), (Li et al. 2009). According to the (Li et al.

2009) work, mutation has been found to be more cost-effective than both the statement and decision coverage criteria. Besides the application cost effectiveness, there is always the need of achieving a higher level of confidence on specific programs or units. In such a situation, mutation testing could be employed in order to construct or to augment an existing test suite. In the study of (Smith and Williams 2009) this question is addressed. Their results suggest that mutation forms a good choice for providing guidance on augmenting a test suite on the one hand and also that increasing mutation leads on an improved structural coverage in many cases. These results also give rise to the employment of the proposed framework in order to automate the construction or augmentation of existing test suites for those specific cases. Additionally, in the studies of (James H. Andrews et al. 2006) mutants were showed to be representative of real faults and thus, were used in order to assess the effectiveness of the structural testing criteria. This argument reinforces the suggestions made in this paper as it successfully guides existing test data generation tools to achieve higher mutation coverage and thus, probably detecting a higher rate of faults. Finally, it should be noted that the present paper deals with the automation issues of the test generation process according to mutation and not its cost-effective application.

7.2 Scalability Issues

Mutation testing research primarily considers the automation issues of the testing process. According to the literature, a quite limited number of such approaches exist, while also leaving aside the efficiency and scalability issues. The proposed method, despite giving an answer to the automation issues of mutation testing, its optimal use and application require some special treatment. This is a consequence of the introduced complexity of mutants and their necessity requirements. The introduced complexity is due to the introduction of the mutant-branches that enable two possible targets, the true and false branch outputs, for tools aiming at branch testing. Although, the aim is to kill mutants, in this case cover the true mutant branches, it should not be so trivial to achieve the opposite situation, i.e. to cover the false mutant branches too. Thus, it may be necessary to aim only at the specific mutant branches and not to all meta-program branches. Further, as the mutants-necessity conditions introduction extends the candidate path space, it may be mandatory to use effective heuristics to deal with the mutation extended path space as exhaustive or full testing is prohibitive. The shortest path strategy (Yates and Malevris 1989) also used in (Papadakis and Malevris 2009) forms an answer to circumvent this problem by limiting this path space with the appropriate use of a weighted graph. That way, all the introduced complexity is alleviated by selecting the least weighted paths (shortest paths) that ignore the non targeted mutants. Here, it must be noted that goal oriented approaches can efficiently prune away the introduced complexity as they are guided by the mutant-targets and not the candidate program paths. Conversely, other practical heuristics can also be considered for further efficiency reasons. This is in favor of the proposed approach as it reduces mutants to branches making practical heuristics of branch testing also applicable to mutation testing.

7.3 Limitations

The proposed framework in this paper has several limitations which are currently under further research. First, it targets on weak mutation and thus it is not straightforward in dealing with strong. This forms a current practice in mutation based test generation approaches also undertaken by the (Richard A. DeMillo and Offutt 1991), (Ayari et al. 2007), (Papadakis and Malevris 2009) imposed by the complexity of formalizing the sufficiency condition (Richard A. DeMillo and

Offutt 1991). Besides this, dealing with strong mutation requires a specialized driver in order to evaluate the test thoroughness and direct the test generation tools to produce additional tests. These additional tests are those that cover-kill the mutant branches while failing to strongly kill them. Second, it does not incorporate any of the most recent and powerful specialized testing tools such as the Austin tool (Lakhotia et al. 2010) or the Crest tool (Burnim and Sen 2008) able to effectively target at individual program branches which are ideal for the present framework. This was forced by the unavailability of such tools for the Java language. Nevertheless, incorporating more powerful and specialized tools will achieve the killing of more mutants resulting in a higher level of testing thoroughness. Finally, the inability of the mutant schemata technique to generalize on certain "Structural" Object Oriented mutants, as identified in (Ma et al. 2005), raises the need for their special consideration. Nevertheless, as these mutants target on inter-class testing level their handling falls outside the scope of the present research which is related to the "behavioral", method level mutants (Ma et al. 2005). The present research concentrates on testing at the unit level (method level mutants) while the intra and inter-class testing (Object Oriented mutants) is concentrated on the method and class integration testing (Ma et al. 2005).

8. RELATED WORK

The automatic generation of test data has been regarded as the main issue in software testing for a long period. This is true for all methods developed for assessing the quality of software. In view of this, mutation being a very powerful testing method, could not be left aside especially since, by its definition, it is a very expensive method to use. Despite the need for tools that will alleviate the problems induced by mutation very little has been done towards developing automated tools for this purpose. The most important work can be attributed to DeMillo and Offutt in a method known as the Constraint Based Testing Technique (Richard A. DeMillo and Offutt 1991). The CBT technique has been implemented in a tool called Godzilla for the testing Fortran programs and has been integrated with the Mothra mutation testing environment. Godzilla embodies 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 a specific, to each mutant 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. CBT has empirically been shown to be an effective technique however, it has certain drawbacks with respect to the symbolic evaluation when dealing with the handling of arrays, loops, non linear expressions and the path explosion problem as this is reported in (A. Jefferson Offutt et al. 1999). To overcome these difficulties, the Dynamic Domain Reduction (DDR) (A. Jefferson Offutt et al. 1999) method was proposed. With this method tests are produced based on the reduction of the input spaces of the variables involved. The DDR approach treats the test generation problem as a dynamic path based problem. Its basic characteristic is the generation of test data for a chosen path using a search heuristic over the input domain guided by the program's control flow graph and a backtracking mechanism. In (Papadakis et al. 2010) a transformation of the problem of killing mutants to a covering branches alternative, was suggested. Thus, effective heuristics applied for branch testing can be extended to mutants too. The most popular methods for branch testing are those that select specific path sets to generate the sought test data. As with all path generation methods their major deficiency is the generation of infeasible paths, this problem is also inherited when employing path generation for performing mutation testing too. In (Papadakis and Malevris 2009) a path based strategy that alleviates the effects of infeasible paths (Yates and Malevris 1989) was successfully used for producing mutation adequate test cases.

Dynamic approaches based on searching input domain sets have also been proposed. (Bottaci 2001) proposed a fitness function composed of two parts, one that measures the reachability distance (measures how close the data are to reach the mutant statement) of the produced tests and the other for measuring their necessity distance (measures how close the data are to killing the mutant statement). In (Ayari et al. 2007) an evolutionary approach that generates mutation test data was proposed. In this technique, the generation process is mapped onto a minimization problem guided by an appropriate fitness function. In particular, the authors adopt the ant colony optimization algorithm as a metaheuristic search engine and a partial implementation as they implement the reachability part only, of the fitness function proposed by (Bottaci 2001). In (Papadakis et al. 2010) the use of mutant schemata resulted in an effective treatment of both the reachability and the necessity conditions. Overall, it is noted that all the existing approaches presented so far try to deal with strong mutation indirectly (by targeting on weak mutation), according to the utilized test generation method (constraint based testing, concolic testing and search based optimization). Thus, the proposed approach gives a practical unification of their application based on the mutant schemata.

Approaches dealing directly with strong mutation i.e. handling the sufficiency condition are under research. The mutant schemata approach has been extended to strong mutation utilizing dynamic symbolic execution (Papadakis and Malevris 2010a). This extension utilizes a shortest path heuristic and a special form of mutant schemata in order to prune away the introduced mutant’s complexity and efficiently reach, infect and reveal the introduced mutants. Additionally, the mutant’s sufficiency condition is approximated by exploring the mutant program path space. Along the same lines is also the work of Fraser and Zeller (Fraser and Zeller 2010) who implement a search based optimization approach based on the mutants’ impact measure (Schuler and Zeller 2010). Thus, according to (Fraser and Zeller 2010) an approximation of the sufficiency condition can be established.

Many dynamic approaches and tools have appeared in the literature for branch testing based on either concolic (Sen et al. 2005) or search based optimization techniques e.g. (Lakhotia et al. 2010), (Harman and McMinn 2007). Most of these techniques try to effectively utilize optimization algorithms and input domain control. Harman and McMinn (Harman and McMinn 2007) conducted a comprehensive theoretical and empirical study of search based optimization approaches used in software testing. Their results suggest that simple hill climbing techniques are the most effective for generating structural tests. Based on these results, the Austin tool (Lakhotia et al. 2010) that employs hill climbing technique was built for testing C programs with dynamic data structure inputs. Considering various heuristics for the concolic approach the Crest tool (Burnim and Sen 2008) was implemented. As the proposed schemata approach can assist any structural testing tool to perform mutation, all these approaches and tools can be also employed to produce mutation based test data.

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, while avoiding the introduction of equivalent ones on the other. Such an approach is proposed in (Papadakis and Malevris 2010b) 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 (Schuler and Zeller 2010). According to the authors, dynamic invariants and coverage measures are introduced into the program under test. The mutants are assessed based on their impact on the invariants and coverage. By targeting 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.

9. Conclusion and Future Work

Test data generation is a tedious and expensive task. Its automation helps the effective application of software testing techniques. However, such automated tools do not exist for all techniques. Mutation testing is a well researched, highly powerful and promising technique. Despite this it has not been as widely used as should be expected. One of the possible reasons behind this is probably its high complexity and the lack of automated tools to facilitate this problem. It is this deficiency that the present research tried to cover. The approach proposed here, *sets the foundations of how to automatically produce mutation based test cases according to established structural testing test generation techniques*. Additionally, existing automated tools can be straightforwardly utilized for this purpose. Thus, instead of developing a purpose built automated tool for generating mutation test data, it is suggested using existing ones for other well established techniques such as branch testing whose successful performance is well known. To evaluate this argument a number of Java programs was used with three different test data generation tools. Other languages or tools can be also used in a similar fashion. The innovation of the present work is that it has managed to make this argument possible. *The practicality of the proposed approach comes from its ability to utilize any automated tools (for structural testing) as it is i.e. without requiring any internal modification.* This is achieved by modifying the test subjects' source codes, using the mutant schemata technique. The automated tools used were based on symbolic evaluation, concolic execution and search based optimization, demonstrating that these techniques can be effectively employed to generate mutation test cases.

From the conducted study it can be concluded that a *high level of automation of the generation of test cases for killing the mutants can be achieved*. This effort provides the foundations for exploring the capabilities of symbolic execution, concolic execution and search based optimization techniques for fault based testing. Additionally, it can be concluded that the *proposed approach on weak mutation, achieves a significant speedup on the mutant execution process* by making it appear as an ideal choice for such a task. Finally, the results also suggest that mutation requires powerful and scalable tools, able to handle complex expressions. In future, it is planned to implement a concolic and an evolutionary based prototype in order to achieve a higher level of mutation coverage capable of handling equivalent mutants. By reducing mutants to branches, equivalent mutants are transformed into infeasible branches. The utilization of an automated tool such as (Baluda et al. 2010), which is able to both produce test cases and detect infeasible program branches, used with the proposed approach, could improve the automation of mutation testing. This is a direct consequence of aiming at both the test case generation and equivalent mutant detection. Additionally, we plan to extend the proposed approach in order to handle second order mutants (Papadakis and Malevris 2010b) so as to perform mutation at a greater scale.

REFERENCES

[1] Ammann, P., & Offutt, J. (2008). *Introduction to software testing*: Cambridge University Press.

[2] Andrews, J. H., Briand, L. C., & Labiche, Y. 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). 1062530: ACM. doi:http://doi.acm.org/10.1145/1062455.1062530.

[3] Andrews, J. H., Briand, L. C., Labiche, Y., & Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng., 32*(8), 608-624, doi:http://dx.doi.org/10.1109/TSE.2006.83.

[4] Ayari, K., Bouktif, S., & Antoniol, G. 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). 1277172: ACM. doi:http://doi.acm.org/10.1145/1276958.1277172.

[5] Baluda, M., Braione, P., Denaro, G., & Pezzè, M. Structural coverage of feasible code. In *Proceedings of the 5th Workshop on Automation of Software Test, Cape Town, South Africa, 2010* (pp. 59-66). 1808275: ACM. doi:http://doi.acm.org/10.1145/1808266.1808275.

[6] Baresi, L., Lanzi, P. L., & Miraz, M. Testful: An evolutionary test approach for java. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, 6-10 April 2010 2010* (pp. 185-194). doi:http://dx.doi.org/10.1109/ICST.2010.54.

[7] Bottaci, L. A genetic algorithm fitness function for mutation testing. In *SEMINAL: Software Engineering using Metaheuristic INovative Algorithms, Workshop, 2001* (pp. 3-7)

[8] Burnim, J., & Sen, K. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008* (pp. 443-446). 1642995: IEEE Computer Society. doi:http://dx.doi.org/10.1109/ASE.2008.69.

[9] DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer, 11*(4), 34-41, doi:http://dx.doi.org/10.1109/C-M.1978.218136.

[10] DeMillo, R. A., & Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng., 17*(9), 900-910, doi:http://dx.doi.org/10.1109/32.92910.

[11] Fraser, G., & Zeller, A. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis, Trento, Italy, 2010* (pp. 147-158). 1831728: ACM. doi:http://doi.acm.org/10.1145/1831708.1831728.

[12] Hamlet, R. G. (1977). Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng., 3*(4), 279-290, doi:http://dx.doi.org/10.1109/TSE.1977.231145.

[13] Harman, M., & McMinn, P. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 international symposium on Software testing and analysis, London, United Kingdom, 2007* (pp. 73-83). 1273475: ACM. doi:http://doi.acm.org/10.1145/1273463.1273475.

[14] Harrold, J., & Rothermel, G. Siemens programs, hr variants. http://www.cc.gatech.edu/aristotle/Tools/subjectsMay 2010.

[15] Howden, W. E. (1982). Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng., 8*(4), 371-379, doi:http://dx.doi.org/10.1109/TSE.1982.235571.

[16] Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 99*(PrePrints), doi:http://dx.doi.org/10.1109/TSE.2010.62.

[17] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM, 19*(7), 385-394, doi:http://doi.acm.org/10.1145/360248.360252.

[18] King, K. N., & Offutt, A. J. (1991). A fortran language system for mutation-based software testing. *Softw. Pract. Exper., 21*(7), 685-718, doi:http://dx.doi.org/10.1002/spe.4380210704.

[19] Lakhotia, K., Harman, M., & Gross, H. Austin: A tool for search based software testing for the c language and its evaluation on deployed automotive systems. In *2nd International Symposium on Search Based Software Engineering, 2010*. doi:http://dx.doi.org/10.1109/SSBSE.2010.21.

[20] Li, N., Praphamontripong, U., & Offutt, A. J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09), Denver, Colorado, 1-4 April 2009* (pp. 220-229). doi:http://dx.doi.org/10.1109/ICSTW.2009.30.

[21] Ma, Y.-S., Offutt, J., & Kwon, Y. R. (2005). Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab., 15*(2), 97-133, doi:http://dx.doi.org/10.1002/stvr.v15:2.

[22] Maldonado, J. C., Delamaro, M. E., Fabbri, S. C. P. F., Simão, A. d. S., Sugeta, T., Vincenzi, A. M. R., et al. (2001). Proteum: A family of tools to support specification and program testing based on mutation. In W. E. Wong (Ed.), *Mutation testing for the new century* (pp. 113-116): Kluwer Academic Publishers.

[23] Mastorantonakis, M., & Malevris, N. An effective metrod for mutating java programs. In *Proceedings of the IASTED International Conference on Software Engineering and Applications, 2003* (Vol. 7, pp. 252-257)

[24] Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol., 1*(1), 5-20, doi:http://doi.acm.org/10.1145/125489.125473.

[25] Offutt, A. J., Jin, Z., & Pan, J. (1999). The dynamic domain reduction procedure for test data generation. *Softw. Pract. Exper., 29*(2), 167-193, doi:http://dx.doi.org/10.1002/(SICI)1097-024X(199902)29:2<167::AID-SPE225>3.3.CO;2-M.

[26] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol., 5*(2), 99-118, doi:http://doi.acm.org/10.1145/227607.227610.

[27] Offutt, A. J., & Lee, S. D. (1994). An empirical evaluation of weak mutation. *IEEE Trans. Softw. Eng., 20*(5), 337-344, doi:http://dx.doi.org/10.1109/32.286422.

[28] Offutt, A. J., & Pan, J. (1997). Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability, 7*, 165-192, doi:http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U.

[29] Offutt, A. J., & Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century* (pp. 34-44): Kluwer Academic Publishers.

[30] Offutt, J., & Voas, J. (1996). Subsumption of condition coverage techniques by mutation testing. Dept. of Information and Software Systems Engineering, George Mason Univ., Fairfax, Va.

[31] Papadakis, M., & Malevris, N. An effective path selection strategy for mutation testing. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference, 2009* (pp. 422-429): IEEE Computer Society. doi:http://dx.doi.org/10.1109/APSEC.2009.68.

[32] Papadakis, M., & Malevris, N. Automatic mutation test case generation via dynamic symbolic execution. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, 1-4 Nov. 2010 2010a* (pp. 121-130). doi:http://dx.doi.org/10.1109/ISSRE.2010.38.

[33] Papadakis, M., & Malevris, N. An empirical evaluation of the first and second order mutation testing strategies. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, 6-10 April 2010 2010b* (pp. 90-99). doi:http://dx.doi.org/10.1109/ICSTW.2010.50.

[34] Papadakis, M., Malevris, N., & Kallia, M. Towards automating the generation of mutation tests. In *Proceedings of the 5th Workshop on Automation of Software Test, Cape Town, South Africa, 2010* (pp. 111-118). 1808283: ACM. doi:http://doi.acm.org/10.1145/1808266.1808283.

[35] Păsăreanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S., et al. 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). 1390635: ACM. doi:http://doi.acm.org/10.1145/1390630.1390635.

[36] Polo, M., Piattini, M., & García-Rodríguez, I. (2009). Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability, 19*(2), 111-131, doi:http://dx.doi.org/10.1002/stvr.392.

[37] Schuler, D., & Zeller, A. (un-)covering equivalent mutants. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, 6-10 April 2010 2010* (pp. 45-54). doi:http://dx.doi.org/10.1109/ICST.2010.30.

[38] Sen, K., Marinov, D., & Agha, G. 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). 1081750: ACM. doi:http://doi.acm.org/10.1145/1081706.1081750.

[39] Smith, B. H., & Williams, L. (2009). On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Softw. Engg., 14*(3), 341-369, doi:http://dx.doi.org/10.1007/s10664-008-9083-7.

[40] Tonella, P. (2004). Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes, 29*(4), 119-128, doi:http://doi.acm.org/10.1145/1013886.1007528.

[41] Untch, R. H., Offutt, A. J., & Harrold, M. J. 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). 154265: ACM. doi:http://doi.acm.org/10.1145/154183.154265.

[42] Visser, W., Păsăreanu, C. S., & Khurshid, S. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA, 2004* (pp. 97-107). 1007526: ACM. doi:http://doi.acm.org/10.1145/1007512.1007526.

[43] Yates, D., & Malevris, N. (1989). Reducing the effects of infeasible paths in branch testing. *SIGSOFT Softw. Eng. Notes, 14*(8), 48-54, doi:http://doi.acm.org/10.1145/75309.75315.