Towards Automating the Generation of Mutation Tests

Mike Papadakis
Department of Informatics
Athens University of Economics and Business
mpapad@aueb.gr

Nicos Malevris
Department of Informatics
Athens University of Economics and Business
ngm@aueb.gr

Maria Kallia
Department of Informatics
Athens University of Economics and Business
kalliam@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 them aim at branch testing, while a quite limited number aim at a higher level of testing thoroughness such as mutation. In this paper an automated framework that makes a joint use of diverse techniques and tools is introduced in the context of automating mutation based test generation. The motivation behind this work is the use of existing techniques and tools such as symbolic execution and evolutionary testing towards automating the test input generation activity according to the weak mutation testing criterion. The proposed framework integrates existing automated tools for branch testing in order to effectively generate mutation test data. To fulfill this suggestion three automated tools are used for illustration purposes and preliminary results are obtained by applying the proposed framework to a set of java program units indicating the applicability and effectiveness of the proposed approach.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Testing tools

General Terms
Verification

Keywords
Mutation testing, automated test case generation, symbolic execution, concolic execution, genetic algorithms.

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 software testing cost, a lot of effort has been put towards automating the test data generation process, thus also reducing the overall software development cost. This activity is usually performed by utilizing an automating tool that produces the sought test data. In the absence of such tools this activity must be manually performed making the testing cost more unbearable. 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 for their effectiveness, ease of use and straightforward evaluation.

Mutation testing is a powerful fault-based yet highly expensive testing technique initially introduced by Hamlet [8] and DeMillo et al. [6]. 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 [2], 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" [10] 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 [28]. 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 [18] and [15] with promising results.

To find appropriate test data with relevance to a selected criterion can be a very tedious task [20]. 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. Most of the progress in the area has been reported by DeMillo and Offutt [7] in a technique called Constraint Based Testing (CBT). CBT uses paths and symbolic evaluation to construct sets of conditions

AST '10, May 3-4, 2010, Cape Town, South Africa

under which inputs should execute and infect the program state of the considered mutated programs. This approach although powerful has not been implemented or incorporated in an automated tool for modern programming languages such as java. In general, approaches employing mutation are scarce in the literature. Conversely, there appear not to exist any fundamental attempts that effectively utilise recent advances of symbolic execution [12], concolic execution [26] and search based optimization techniques [9]. Techniques that have succeeded in automating the 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 on focusing on specific mutants by selecting appropriate paths or tests in order to generate effective data capable of killing the specified mutants. The benefit of such an activity is that automated tools or techniques implemented for structural testing can be easily utilized with some modifications to perform mutation testing. Thus, mutation testing automation is reflected on the structural testing automation and efficiency, where known achievements have been recorded.

The suggestions made in this paper have been incorporated into an automated framework that uses a novel version of the mutant schemata technique for weak mutation. 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.
- 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.
- An approach that reduces the execution time used by dynamic approaches, such as concolic execution and search based optimization.

The rest of this paper is organised as follows: Section 2 introduces some background material. Section 3 presents some related to the present work. Section 4 details the proposed technique. Sections 5 and 6 report a conducted case study and discuss the practicality issues induced by the application of the proposed technique together with some future directions. Finally in section 7 conclusions 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:

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

The criteria requirements goal is both to guide and evaluate the quality of the test data. Testing based on fault based criteria, 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. 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 [7]. Based on these three conditions, DeMillo and Offutt developed a test data generation technique called Constraint-Based test data generation (CBT) [7] which forms the foundations for killing mutants. CBT answers in a general way, the question of how any approach should attempt to kill mutants.

The Reachability condition states that the mutant statement must be exercised with test data. It must be noted that mutation introduces one fault at a time and all the program's executable statements apart from the mutated one are the same to the original. If tests cannot execute the mutated statement, it is guaranteed that the tests have no chance to kill the seeded mutant [7]. The necessity condition states that the execution of the mutated statement must cause a departure form the original program state [7]. 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 program's output.

Current test data generation approaches [7], [21] try to utilise directly the reachability and necessity conditions based on constraint resolution and domain splitting. Because of its high complexity the sufficiency condition is indirectly satisfied through the satisfaction of the reachability and necessity ones. This is reinforced in [7] and [18] 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 [10]. 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 adapting existing methods for performing other forms of testing to perform mutation.

Modern test generation methods rely on either static or dynamic analysis techniques or on their combination. The peculiarities of mutation itself make difficult the straightforward application of either of the two approaches. To deal with these special

characteristics, an initial attempt was suggested [21] mainly based on static analysis. 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, 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 [7] 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 symbolic 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 benefits 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.

3. 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 when by its definition is a very expensive to use method. 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 [7]. The CBT technique has been implemented in a tool called Godzilla for the testing Fortran programs and has been integrated with the Mothra [5] 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 [17]. To overcome these difficulties, the Dynamic Domain Reduction (DDR) [17] 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 [21], 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 [21] a path based strategy that alleviates the effects of infeasible paths [30] was successfully used for producing mutation adequate test cases.

Dynamic approaches based on searching input domain sets have also been proposed. Bottaci [4] 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 [3] an evolutionary approach that generates mutation test data was proposed. In this technique, the generation process is mapped on to a minimization problem guided by an appropriate fitness function. In particular, the authors adopt the ant colony optimization algorithm [3] as a metaheuristic search engine and a partial implementation as they implement the reachability part only, of the fitness function proposed by Bottaci [4].

Many dynamic approaches have appeared in the literature for branch testing based on either concolic [26] or search based optimization techniques e.g. [9], [27]. Most of these techniques try to effectively utilize optimization algorithms and input domain control. Harman and Mcminn [9] conducted a comprehensive theoretical and empirical study of search based optimization approaches used in software testing. Their results suggest that simple hill climbing techniques as in [13] are the most effective for generating structural tests. Additional integrated approaches that attempt to effectively combine both search based optimizations and symbolic execution have also been suggested. For example in the context of object oriented applications, a framework [11] that attempts to improve the branch coverage by aiming at generating method sequences based on evolutionary testing and method internal structures relying on concolic execution was proposed. A similar hybrid approach [22] that integrates genetic algorithms as a search engine over the input domain and symbolic execution based on the Yates and Malevris method [30] has also been suggested.

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 to introduce equivalent ones on the other. Such an approach is proposed in [24] 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 [25]. According to the authors, dynamic invariants are introduced into the program under test. The mutants are assessed based on their impact with the invariants. By targeting to those with a higher level of impact, a good measure of the adequacy of the test suite is established, while limiting the number of considered mutants and the equivalent ones. However, both of these approaches rely on mutation analysis rather than on generating test data.

4. PROPOSED FRAMEWORK

The proposed framework attempts to automate the test data generation process according to the mutation testing criterion. The framework takes the test objective code written in java as input and automatically generates the required data. To achieve this, a test data generation engine must be employed. Here the presented framework can adopt 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 [23] and the second as etoc [27], an evolutionary based testing tool for java. The third one utilizes the "concolic" execution method [26] and it was implemented by the authors for the purpose of the present paper.

In [21] the foundations of producing certain constraints under which mutants are killed as proposed by DeMillo and Offutt [7], are used in order to construct a suitable model for test generation. This test model, called Enhanced Control Flow Graph (ECFG) [21], 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 [21] 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 in 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).

The proposed framework uses simple but quite effective modifications in order to make use of existing structural testing tools. This follows the spirit of avoiding making drastic alterations on existing tools for producing mutation tests. 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 [21] and thus reduce the mutants into branches. By interfacing this meta-program with an automated tool able to generate tests for structural testing, can effectively produce mutation testing tests for the original program.

The framework can be completely automated and its effectiveness depends on that of the underlying test data generation tool. At present, the framework automatically produces the meta-program that is passed to a test generation tool that produces the required tests. An overview of the proposed framework structure is presented in Figure 1. 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 that they implement. However, it is believed that they serve the general goals of the present study, which is to illustrate the applicability and effectiveness of the proposed framework.

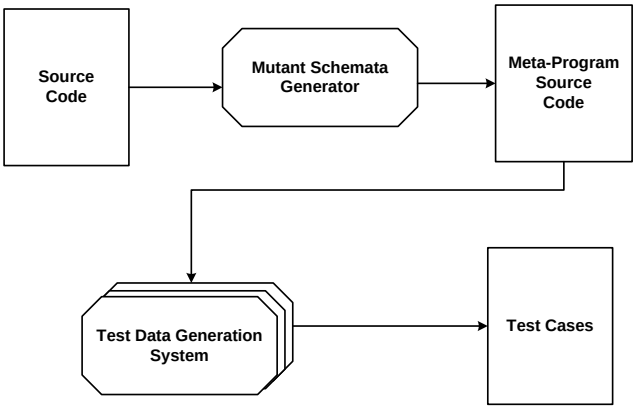

Figure 1. The proposed framework

The automated production of the meta-program uses an extension of the mutant schemata technique which is described in the following subsection (4.1). The tools used for generating tests and their use in the proposed framework are described in the succeeding subsections (4.2. and 4.3).

4.1 Mutant Schemata

Automating test case generation requires specific, to the technique used, information about the target test requirements. Mutation posses difficulties in producing this information (i.e. killing the mutants) as its requirements are spanned across different program versions (one mutant per version). Thus, there is a need for a unification of mutation requirements in a suitable way to be used by techniques, such as symbolic execution or a search based application, appropriate for test generation. By doing so, the candidate mutants are concentrated in a unique representation rather than being spread to one application per mutant. This approach is in a way similar to the one introduced by Untch et al. [28] who proposed the Mutant Schemata Generator (MSG) system. Each pair of operands participating to an operation is passed as parameters of the operator into a schematic function (e.g. a > b becomes FunctionGT(a, b)). Expanding the suggestions of the MSG approach, the evaluation of the mutants' execution is performed within the schematic function [16]. This implies an indirect reduction to a path - branch coverage problem of the mutated programs. By placing the mutant evaluation into the schematic function, the suitable conditions under which a considered mutant is killed is also embedded. These conditions are formed as decisions, into the schematic function, containing the following expression:

$$\textit{Original statement} \neq \textit{Mutated statement} \quad (2)$$

This expression has been used by DeMillo and Offutt [7] in order to produce mutant necessity constraints.

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) braches to represent the possible outcomes.

In order to make this possible, the code before being transferred to the considered test tool needs to be instrumented with the use of calls to statically or dynamically predefined schematic entities. These entities should be defined according to the considered mutant operators. This process is similar to the one presented by Untch et al. [28].

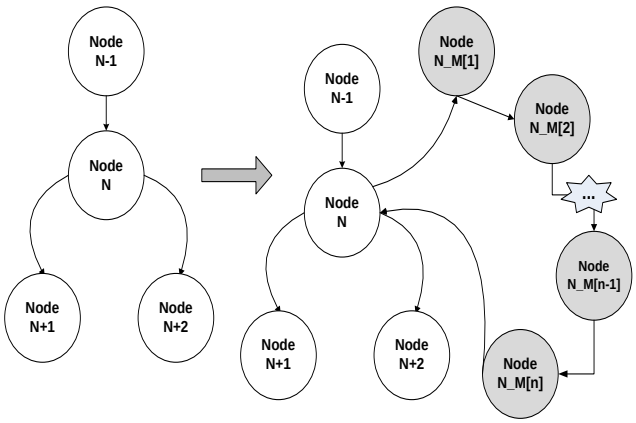

Figure 2. Augmented mutation graph

Comparing outputs of mutant statements with the respective original ones results in testing according to the weak mutation coverage criterion [16]. According to Howden [10], in weak mutation it is acceptable to execute all the mutants for one place when executing program code. Based on this idea the schemata were expanded with internal checks for the local results and were made responsible to execute all the selected mutants due to their position each time they were reached. This means that no mutants are considered if the selected test case can't reach them. This implies that an effective technique or a tool aiming at exercising program branches should now target on mutants effectively. When exiting the schemata, the result of the original code is maintained and returned in order to continue with the program execution along the original execution path, while having performed a quick evaluation of killed or not mutants for the decision node of the program's graph.

In Figure 2, an augmented mutation program graph that contains the eligible mutants is presented. Let us assume that node N of the left graph 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 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 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.

The transformation of the program's graph illustrates 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 should be able to reproduce as path conditions all the suitable conditions under which mutants are killed. Search based approaches should be able to guide the generation process through mutant branches and effectively kill them, by taking into account both the reachability and necessity conditions. Additionally, the use of internal evaluations into the mutant schemata results in a straightforward enumeration of the killed mutants by a selected test and hence there is no need for external driver or tool to calculate the ratio of the killed mutants.

4.2 Symbolic Execution

The symbolic evaluation process [12] 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 [14]. 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, this 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 [23], [29] (JPF-SE) was used. In JPF-SE symbolic execution is performed by initializing the input variables thus to 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, 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.

4.3 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 [27] 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 one of these targets-branches is considered in a row until it is covered or the search reaches a predefined upper bound limit (time or number of evolutions). 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 rest uncovered ones. If the produced tests fail to cover the targeted 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 on 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 [27]. 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 [9], [27]. Despite its limitation this tool has been used in our case study for illustration purposes only. In addition any other tool can be used.

4.4 Concolic execution

The concolic testing (concolic execution) method [26] 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.

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 power of the underlying decision solving procedures. In the present paper a prototype tool that implements the above procedure has been constructed and used in the above described framework for performing mutation. Currently the prototype has some limitations such as the handling of dynamic program inputs, method sequences and floating point arithmetic.

5. CASE STUDY

To perform an initial assessment of the applicability and feasibility of the framework, it was applied to a set of java programs. The selected programs were chosen from a) the testing textbook website by Ammann and Offutt [1], b) from the examples distributed together with the JPF-SE tool [23] and c) from the mutation benchmark programs used by Polo et al. [24]. Table 1 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 Offutt et al. [19]. 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 by any test that executes it. This is also argued in [7]. Thus, four operators were used (i.e. ABS, AOR, LCR and ROR details of these operators can be found in [5]) for the purposes of the case study. In addition, any other mutant operator can also be used by defining and embedding its mutant schemata into the produced schematic meta-program. The schemata were designed based on the initial specifications of the above mutation operators as set in [5]. 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 was described in section 4), aiming to cover program branches first and then mutants.

Table 1. Subject programs

Test Object	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

5.1 Experience

This section reports results from the application of the framework to the selected test objects. 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 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 2. Initial mutation score achieved by the employed tools

Test Object	JPF-SE	Concolic	Etoc
J1	87.69%	86.54%	85.38%
J2	0.00%	44.57%	73.71%
J3	50.31%	62.26%	63.52%
J4	91.89%	90.54%	7.43%
J5	91.67%	91.67%	87.50%
J6	0.00%	73.51%	80.22%
J7	87.74%	87.74%	73.58%
J8	98.67%	98.67%	94.67%
J9	68.25%	66.67%	74.60%
J10	43.96%	46.15%	62.64%

Table 2 presents the initially achieved mutation score by the utilized tools, when used for branch testing. It must be noted that branch coverage was performed for the purpose of obtaining test data for doing so 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 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 objects are purely due to the internal characteristics of the test objects and the methods themselves.

In the next phase of the case study the programs containing the mutant schemata were used as input instead of the original ones. Performing branch testing to these programs leads to the direct killing of the mutants. In Table 3 the results obtained by employing the selected tools are presented. These results record a high coverage level for all three employed tools. The comparison of the results obtained in Tables 2 and 3 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 objects. This should be regarded as an achievement of the proposed framework as it is flexible to adopt the characteristics of the mutants.

Table 3. Mutation score achieved by the employed tools based on the proposed framework

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

This study forms the first step towards automating the test data generation activity for mutation testing. The presented case study focuses on revealing the effectiveness of the proposed framework in guiding existing tools to produce high quality tests. The employed tools are not the most appropriate ones as they cannot focus on specific program branches. 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 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.

6. DISCUSSION AND FUTURE WORK

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 the branch testing in order to effectively apply existing automated methods for this criterion. This is the first attempt to the authors' knowledge of adopting concolic execution [26] for performing mutation testing. Concerning the symbolic evaluation and evolutionary techniques, very little has been done as discussed in the related work section 3. 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 [4] 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 reduced by at least 50% as stated in [18]. Moreover, the use of mutant schemata also results in additional time savings when compared to the traditional separate compilation approach [15] for each mutant. The proposed approach takes advantage of the execution path by executing only the reached mutants. Additionally, by executing all mutants in one execution run results in additional resource execution savings. Furthermore, the technique by combining all the above (weak mutation and mutant schemata) should result in further considerable savings.

Although the proposed method gives answer to the automation issues of mutation testing its optimal use and application requires some special treatment. This is a consequence of the introduced complexity of mutants and their necessity requirements. Thus, it may be mandatory to use effective heuristics to deal with the mutation complexity as exhaustive or full testing is prohibitive. The shortest path strategy [30] also used in [21] forms an answer to circumvent this problem. Conversely, other practical heuristics could be also considered for efficiency reasons. This is something that goes beyond the goals of the present study and is left for future research.

The results reported in this paper indicate the applicability of the proposed technique for mutation testing by using structural testing tools. It is the lack of tools that gave rise to this idea for generating test data to perform mutation in an effective way rather than in an efficient one. The results also suggest that mutation requires powerful and scalable tools, able to handle complex expressions. In future it is planed to expand the implemented concolic prototype in order to achieve a higher level of mutation coverage. Additionally, we plan to explore the application of the technique with other automated tools. Finally, it is planned to measure the efficiency of the proposed mutant schemata technique for killing mutants.

7. CONCLUSION

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 widely used as it 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 instead of developing a purpose built automated tool for generating mutation test data, suggests 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 also be used in a similar fashion. The innovation of the present work is that to make this argument possible, the automated tools were not internally modified.

However, all the necessary modifications were performed in the test objects source codes, by 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 for 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.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] P. Ammann and J. Offutt, "Introduction to Software Testing", Cambridge University Press http://ise.gmu.edu/~offutt/softwaretest/

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Exeperiments?", In Proceedings of the International Conference on Software Engineering, pages 402-411, 2005.

[3] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony", In Proceedings of the annual conference on Genetic and Evolutionary COmputation, pages 1074-1081, 2007.

[4] L. Bottaci, "A genetic algorithm fitness function for mutation testing", In Proceedings of the Software Engineering using Metaheuristic INovative Algortithms workshop, pages 3-7, 2001.

[5] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, K. N. King, "An extended overview of the Mothra software testing environment", In Proceedings of the 2nd workshop on Software Testing, Analysis and Verification, pages142-151, 1988.

[6] R. A. Demilo, R. J. Lipton, and F. D. Sayward, "Hints on test data selection: Help for the practicing programmer", IEEE Computer, 11(4):34-41, 1978.

[7] R. A. Demilo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Transactions on Software Engineering, 17(9):900-910, 1991.

[8] R. G. Hamlet, "Testing program with the aid of a compiler", IEEE Transactions on Software Engineering, 3(4):279-290, 1977.

[9] M. Harman and P. McMinn, "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation", In Proceedings of the International Symposium on Software Testing and Analysis, pages 73-83, 2007.

[10] W. E. Howden, "Weak mutation testing and completeness of test sets", IEEE Transactions on Software Engineering, 8(4):371-379, 1982.

[11] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution", In Proceedings of Automated Software Engineering Conference, pages 297-306, 2008.

[12] J.C. King, "Symbolic execution and program testing", Communications of the ACM, 19(7):38-94, 1976.

[13] B. Korel, "Automated software test data generation", IEEE Transactions on Software Engineering, 16(8):870-879, 1990.

[14] C. Koutsikas and N. Malevris, "A Unified Symbolic Execution System", In Proceedings of the International Conference on Computer Systems and Applications, pages 466-469, 2001.

[15] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: An automated class mutation system", Software Testing, Verification and Reliability, 15(2):97-133, 2005.

[16] M. Mastorantonakis and N. Malevris, "An Effective Metrod for Mutating JAVA Programs" In Proceedings of the International Conference on Software Engineering and Applications, 2003.

[17] A. J. Offutt, Z. Jin and J. Pan, "The dynamic domain reduction approach to test data generation", Software: Practice and Experience, 29(2):167-193, 1999.

[18] A. J. Offutt and D. S. Lee, "An Empirical Evaluation of Weak Mutation", IEEE Transactions on Software Engineering, 20(5):337-344, 1994.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf "An experimental Determination of Sufficient Mutation Operators", ACM Transactions on Software Engineering and Methodology, 5(2):99-118, 1996.

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

[21] M. Papadakis and N. Malevris, "An Effective Path Selection Strategy for Mutation Testing", In Proceedings of Asia-Pacific Software Engineering Conference, pages 422-429, 2009.

[22] M. Papadakis and N. Malevris, "Improving Evolutionary Test Data Generation with the Aid of Symbolic Execution", AIAI 2009 Artificial Intelligence Techniques in Software Engineering Workshop, pages 201-210, 2009.

[23] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software", In Proceedings of the International Symposium on Software Testing and Analysis, pages 15-25, 2008.

[24] M. Polo, M. Piattini, I.G. Rodriguez, "Decreasing the cost of mutation testing with second-order mutants", Software Testing, Verification and Reliability, 19(2):111-131, 2009.

[25] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations". In Proceedings of the International Symposium on Software Testing and Analysis, pages 69-80, 2009.

[26] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for C", In Proceedings of the International Symposium on Foundations of Software Engineering, pages 263-272, 2005.

[27] P. Tonella, "Evolutionary testing of classes", In Proceedings of the International Symposium on Software Testing and Analysis, pages 119-128, 2004.

[28] R. Untch, J. Offutt and M. J. Harrold, "Mutation analysis using mutant schemata", In Proceedings of the International Symposium on Software Testing and Analysis, pages 139-148, 1993.

[29] W. Visser, C. S. Pasareanu, and S. Khurshid. "Test input generation with Java PathFinder", In Proceedings of the International Symposium on Software Testing and Analysis, pages 97-107, 2004.

[30] D. F. Yates and N. Malevris, "Reducing the effects of infeasible paths in branch testing", ACM Software Engineering Notes, 14(8):48-54, 1989.