Metallaxis: An Automated Framework for Weak Mutation

Mike Papadakis
University of Luxembourg
72 PUBLICATIONS 1,278 CITATIONS
SEE PROFILE

Nicos Malevris
Athens University of Economics and Business
51 PUBLICATIONS 611 CITATIONS
SEE PROFILE

Metallaxis: An Automated Framework for Weak Mutation

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

Abstract— **Mutation testing is highly regarded as a key method towards fault revealing. Despite this advantage, it has proved to be rather impractical for industrial use because of the expenses involved. To this extend, automated techniques have proved suitable for applying and reducing the method's demands. Whilst there is much evidence that automated test data generation techniques can effectively automate the testing process, there has been little work on applying them in the context of mutation testing. In this paper an automatic mutation based test data generation framework called "Metallaxis" is proposed. The framework uses a novel dynamic execution scheme in order to both introduce the required mutants and to effectively generate test cases able to kill those mutants. A conducted case study employing the proposed approach reveals the feasibility, practicality and effectiveness of the proposed approach.**

Keywords: Automated test case generation, search based testing, mutation testing, mutant schemata

I. Introduction

Software testing can account more than half of the cost of the software under development. As the main purpose is to reduce such an excessive cost, the testing activity should incorporate effective and efficient methods experiencing the highest possible level of automation. The test data generation process plays a crucial role in both the effectiveness and efficiency of the software testing phase. Unfortunately, as it is evident from current practice, the level of automation achieved to date is not as high as it ought to be, thus resulting in a rather low quality testing activity due to the unavoidably high cost of the imperative laborious manual activity. Hence, the need for automatically producing the required test data is essential in order to increase the test thoroughness and to reduce the testing expenses at the same time.

Testing quality is usually measured by the test adequacy criteria. Adequacy criteria often referred to as coverage criteria, involve certain requirements that should be fulfilled by the test cases. Mutation testing or mutation analysis, is a fault-based coverage criterion initially introduced by Hamlet [1] and DeMillo et al. [2]. Mutation analysis introduces purposely built errors into programs under test by making alterations (mutants) to the code under test based on a set of simple syntactic rules called mutant operators. The purpose of injecting faults into programs is to both guide the generation of test cases to reveal them on the one hand and to assess the test data quality on the other. To this extend testing seeks to reveal the mutants, which when detected are called "killed" and "live" in the opposite case. The strength of the method relies on the hypothesis - ability of the introduced syntactic faults-mutants to produce realistic, semantic faults. In the study by Andrews et al. [3], this hypothesis is reinforced. Additionally, in [3] mutants were used instead of real faults in order to simulate and compare the effectiveness of various structural testing criteria. Thus, from this study it is evident that developers can benefit from employing mutation analysis.

Automating mutation analysis requires the automatic production of the candidate mutant programs as well as execution with the candidate test cases. This can be efficiently automated with the mutant schemata approach [4] which embeds all the candidate mutants into one schematic meta-program and thus, all tests are executed against this schematic program. Test execution poses an additional barrier to mutation analysis as it requires test cases to be executed against all live mutants. To effectively reduce the execution time required for mutation, alternative methods called weak and firm mutations [5] have been proposed. According to these methods the program execution may stop after the mutated or a succeeding program expression. Evaluation of the mutant can be performed by checking the program state at the stopping execution statement. Thus, execution savings of more than 50% can be achieved [5] with only a small loss of the method 's effectiveness.

The practical use of an adequacy criterion requires the automated generation of test cases according to its requirements. This can prove to be a very tedious task [6] for any selected criterion including mutation. Recently, search based optimization techniques and tools [7] and [8] have succeeded to automate the test case generation activity quite effectively. Conversely, these techniques have not been effectively incorporated into automated tools in the context of mutation. This paper introduces an automated framework named Metallaxis that performs mutation analysis and generation of test cases. Metallaxis is the Greek word for mutation, in English it is used in the context of describing moth species basically when they change from worm to moth.

In Metallaxis the mutants are automatically generated based on a novel version of the mutant schemata technique for performing weak mutation and search based testing. It incorporates a hill climbing search based optimization approach for producing the sought tests. The contribution of the present work can be summarized into the following proposed points:

- An automated technique for producing mutation based test cases employing a search based approach.
- A novel integration of mutant schemata and test data generation techniques to perform mutation testing.
- An automated test data generation prototype using the alternating variable method (hill climbing).

The rest of this paper is organised as follows: Section II presents the Metallaxis system by detailing its components. In section III the obtained results from the application of the framework are reported and analysed. Sections IV and V discuss the relevance and the benefits of the application of the Metallaxis system with previously proposed systems and approaches. Finally in section VI conclusions and future directions are given.

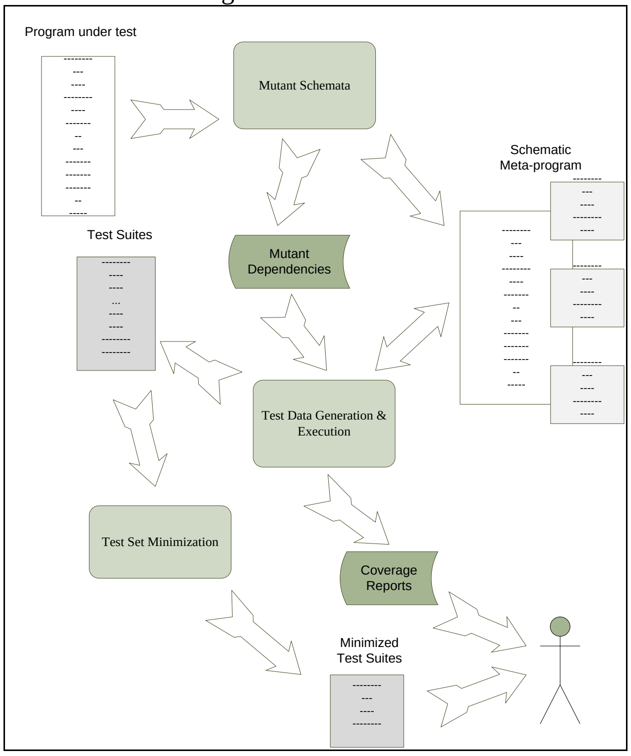

Figure 1. The Metallaxis system.

II. METALLAXIS: SYSTEM DESCRIPTION

The proposed framework system makes a joint use of diverse techniques in order to automatically produce and evaluate mutation based test data. The framework is composed of three basic components A: the "Mutant Schemata" component, B: the "Test Data Generation & Execution" component and C: the "Test Set Minimization" component. Figure 1 presents and summarizes the Metallaxis framework practice, which first by utilizing A embeds all the candidate mutants into one schematic meta-program and produces control dependence details. Then component B produces test cases according to the alternating variable method guided by the schematic program. Finally, component C minimizes the produced tests and presents them to the user along with a report of the achieved coverage. In the succeeding sections details of the framework and the underlying components are given.

A. *Generating Mutants*

Dynamic approaches are based on the information gained through dynamic program runtime execution. In the context of structural testing the programs under test host all the needed information in their structure and thus, it is straightforward to implement a monitoring mechanism for the data evolution purpose. In the context of mutation, there is a special need for unifying both the original's and the mutants' runtime information. The difficulties originate from the mutations' nature, as the needed information is spanned across the original and the various mutants versions [9], [10] programs. To efficiently overcome this difficulty a special form of the mutant schemata technique was employed in order to unify all the mutation analysis requirements into a unique version suitable for the test evolution representation. This approach was initially introduced in [10] in the context of using existing test data generation tools for performing mutation. Here the technique has been extended in order to effectively guide the test data generation process. This is achieved by embedding a fitness guide and evaluation inside the schematic functions.

The Mutant Schemata Generation (MSG) [4] technique encodes into one meta-program all the introduced mutants. This is achieved by appropriately replacing each pair of operands participating in an operation with a call to a schematic function, with this pair of operands as parameters (e.g. a > b becomes RelationalGT(a, b)). Expanding the suggestions of the MSG approach, the evaluation of the mutants' execution and all the required fitness calculations are performed within the schematic function. This as it is shown in [10] reflects the killing mutants problem to a path - branch coverage problem. By incorporating the mutant evaluation into the schematic function, the necessary conditions for killing each considered mutant are also embedded. These conditions, which take the following expression, are formed as decisions in the schematic function.

$$\textit{Original expression} \neq \textit{Mutated expression} \qquad (1)$$

The above expression was also used by DeMillo and Offutt [6] in order to produce mutant necessity constraints, based on which mutation based tests are derived.

The above decision expression (1) is consisted of two possible outcomes (i.e. the original is either equal to the mutated or not). Thus, entailing the introduction of true (mutant is killed) and false (mutant is alive) cases to represent the possible mutant evaluation outcomes. Measuring the closeness of making the above decision (1) true, results in an effective measure of the test case fitness. This measure forms the main guide of the test data generation engine see section II.D. Evaluating the mutants depending on the mutated statements, results in testing according to the weak mutation testing criterion [11]. As Howden suggests [11] when performing weak mutation, all mutants can be executed for a specific program location when exercising the program code. Based on this idea the enhancement of mutant schemata included internal evaluations for the results obtained for the specific program expression. These schemata were also made responsible for executing all the selected mutants due to their position each time they were reached. This means that ***no mutants are to be considered if the selected test cases cannot reach them***. Upon the termination of a schema, the result of the original code is maintained and returned in order to proceed with the program execution along the original execution path, while also recording the killed or not mutants. This implies that ***a single program execution run per test case is sufficient to detect all the resulting killed mutants***. This leads to major execution savings at the test set minimization stage, see section II.D.

In Metallaxis the schemata component is responsible for injecting the schematic function calls into the produced meta-program and to generate the mutant related dependencies. Such mutant dependencies are the program control ones that reach a respective mutant statement. This information is used by the test case evolution engine (Test Data Generation & Execution component) in order to guide the search towards a targeted mutant.

B. *Executing Mutants with Tests*

The present approach takes advantage of the unified representation of all mutants and their killing conditions into one meta-program. Based on the use of the introduced schemata technique, mutant execution can be performed straightforwardly without the need of a specialized mutant execution driver. This is a direct consequence of embedding the mutant evaluation inside the schematic functions. Additionally, mutation score and fitness function calculations have also been embedded into the schematic class. Thus, test execution requires only an initialization of the mutant schemata class at the beginning of the process and an additional call to the calculation function at the end.

The Metallaxis system employs the reflection mechanism of the java language in order to execute the meta-program with the produced test cases and extracts the fitness function calculations. The test execution driver is tighten to the test data generation component as it is responsible for both executing the produce tests and acquiring the fitness calculations.

C. *Evolutionary Testing*

Search based testing [7], [12] techniques formulate the test data generation problem as a search problem and employ search based optimization techniques in order to effectively generate the sought test data. The search is guided by an appropriate fitness function which indicates how close the tests are in covering a specific program element – target. This paper concentrates on mutation testing at the intra-method level [13]. To achieve this, a separate search is performed according to each live mutant.

The approach described here aims at optimizing a fixed number, and range, of input variables according to the employed fitness function (see section II.D.1). This constitutes a common practice in evolutionary testing. To this extend, the framework assumes that a special driver that handles the program input (with varying input size) and the fixed size input pre-exists. A program that takes an array as input is restricted to a predefined fixed length say 10 for example. This results in test cases including only arrays of 10 elements. This approach can be also used in order to produce tests of arrays with less than 10 elements by using an additional variable (representing array size say SIZE) with a range of values 1-10, where the driver ignores the 10 - SIZE elements.

D. *Search Based Testing*

Search based testing [7], [12] techniques formulate the test data generation problem as a search problem and employ search based optimization techniques in order to effectively generate the sought test data. The search is guided by an appropriate fitness function which indicates how close the tests are in covering a specific program element – target. This paper concentrates on mutation testing at the intra-method level. To achieve this, a separate search is performed according to each live mutant. The approach described here aims at optimizing a fixed number, and range, of input variables according to the employed fitness function. This constitutes a common practice in evolutionary testing. To this extend, the framework assumes that a special driver that handles the program input (with varying input size) and the fixed size input pre-exists. It should be noted that this approach can be also used in order to produce input tests with less size than the fixed one.

TABLE I. BRANCH FITNESS

Expression	**True Branch**	**False Branch**
a == b	*abs(a - b)*	*a == b?k : 0*
a != b	*a != b? 0 : k*	*abs (a != b?a - b : 0)*
a < b	*abs (a < b?0 : a - b + k)*	*abs (a < b?a - b + k : 0)*
a <= b	*abs (a <= b?0 : a - b)*	*abs (a <= b?a - b : 0)*
a > b	*abs (a > b?0 : a - b + k)*	*abs (a > b?a - b + k : 0)*
a >= b	*abs (a >= b?0 : a - b)*	*abs (a >= b?a - b : 0)*
a || b	*min[fit(a), fit(b)]*	*fit(a) + fit(b)*
a && b	*fit(a) + fit(b)*	*min[fit(a), fit(b)]*

```
Mutatest(int i, int j, int k){
  int ret = 0;
    if ( i > j )
      if ( j < k )
        if ( i < k )
}
```
Original program

```
Mutatest (int i, int j, int k){
  int ret = 0;
    if (RelationalGT(i, j, 1))
      if (RelationalLT(j, k, 8))
        if(RelationalL(i, k,15))
}
```
Mutated program

Figure 2. Demonstrating Example

1) *Fitness function*

Search based testing has been proved to be an effective method for generating test input data. In order to be effective an appropriate fitness function needs to be employed. The present framework utilizes a fitness function composed of three parts. The first two are known as the “approach level” and the “branch distance” introduced by Wegener et al. [14] in the context of structural testing, and the third one is named “mutation distance”.

The approach level measures the closeness, of a candidate test case, for executing a target mutant statement. It is calculated by counting the number of the target mutant’s control dependent nodes missed by the candidate input test. It should be noted that compound predicates have been split into multiple simple ones (including both true and false outcomes). This is done in order to include those nodes into the approach level measurement, along the lines suggested in [15]. The branch distance quantifies the distance from flipping a branch i.e. making it from true to false or the opposite. It is computed using the runtime values of the branch expression of interest. The expression of interest is the topmost of the missed ones from the mutant control dependencies. This measure is calculated based on the expression formulas used by Awedikian et al. [15] also recorded at table I. Mutation Distance as introduced in this paper reflects the branch distance measure on mutants. It should be noted that these three measures guide the search towards fulfilling the reachability and mutant necessity constraints proposed by Demillo and Offutt [6]. Table II presents the expression formulas based on which Mutation Distance fitness calculations were made. These formulas were obtained by simplifying and reducing the necessity constraints and provide useful information for killing the considered mutants. In Table II the Ffit(x) and Tfit(x) signify the True and False branch fitness of clause x respectively.

Computing the overall fitness of the test cases requires a unification of the three used measures. This is done based on the following equation:

$$\begin{aligned} fitness = 2 * approach\ level \\ + normilized(Branch\ Distance) \\ + normilized(Mutation\ Distance) \end{aligned}$$

2) Hill Climbing

Metallaxis uses the alternating variable method, proposed by Korel [16]. This method forms a Hill climbing algorithm which has been shown to be more effective than other search algorithms in the context of structural testing [7] and has also been incorporated to automated test data generation tools such as the Austin tool [8], [17] for structurally testing C programs. Hence, it forms an ideal choice as it is a quite simple to implement method and should outperform other search alternatives in the current context. Here, it should be noted that mutation testing, in particular weak mutation, can be transformed to branch testing [10] and since hill climbing outperforms its rivals, in the structural testing context [7], there is no reason why this should not hold for mutants too. Nevertheless, this is beyond the scope of the present paper and is left open for future research.

The method starts by randomly initializing the input program variable values. Then it repeatedly selects and adjusts one of those values by altering it. This is performed until no further fitness improvement can be obtained i.e. no further alternations are fruitful. In this case the method switches to the next input variable. The algorithm stops when no further fitness improvement can be recorded by selecting and alternating any of the input variables. Consider the example of Figure 2. In the left part of Figure 2 the original sample program is presented. In its right part the mutated meta-program is detailed. The introduction of the mutants is recorded in the alterations made to the original program e.g. the statement *if (i < k)* has become *if (RelationalGT(i, k, 15))*. The variables *i* and *j* are the two original operand variables while 15 signifies that this expression contains the mutants identified by the relational operator (7 mutants) with identification numbers from 15 to 21.

TABLE II. MUTATION FITNESS

Operator	**Original expression**	**Mutant Fitness**	
Relational	*a > b*	*a >= b: abs(a-b)* *a < b: k* *a <= b: 0*	*a != b: abs(a-b+k)* *a == b: abs(a-b)* *true: abs(a-b)* *false: abs(a-b+k)*
	a >= b	*a > b: abs(a-b)* *a < b: 0* *a <= b: k*	*a != b: abs(a-b)* *a == b: abs(a-b+k)* *true: abs(a-b+k)* *false: abs(a-b)*
	a < b	*a > b: k* *a >= b: 0* *a <= b: abs(a-b)*	*a != b: abs(a-b+k)* *a == b: abs(a-b)* *true: abs(a-b)* *false: abs(a-b+k)*
	a <= b	*a > b: 0* *a >= b: k* *a < b: abs(a-b)*	*a != b: abs(a-b)* *a == b: abs(a-b+k)* *true: abs(a-b+k)* *false: abs(a-b)*
	a != b	*a > b: abs(a-b+k)* *a >= b: abs(a-b)* *a < b: abs(a-b+k)* *a <= b: abs(a-b)*	*a == b: 0* *true: abs(a-b)* *false: k*
	a == b	*a > b: abs(a - b)* *a >= b:abs(a-b+k)* *a < b: abs(a - b)* *a <= b:abs(a-b+k)*	*a != b: 0* *true: k* *false: abs(a-b)*
Arithmetic	*a + b*	*a - b:k* *a * b:k* *a / b:k*	*a % b:k* *a:k* *b:k*
	a – b	*a + b:k* *a * b:k* *a / b:k*	*a % b:k* *a:k* *b:k*
	*a * b*	*a + b:k* *a - b:k* *a / b:k*	*a % b:k* *a:k* *b:k*
	a / b	*a + b:k* *a – b:k* *a * b:k*	*a % b:k* *a:k* *b:k*
	a % b	*a + b:k* *a – b:k* *a * b:k*	*a / b:k* *a:k* *b:k*
Absolute	*a*	*abs(a):abs(a+k)*	*-abs (a):abs(a)* *0:abs(a)*

Logical	*a && b*	*a||b:min[Tfit(a)+Ffit(b), Ffit(a)+Tfit(b)]*
		a:Tfit(a)+Ffit(b)
			b:Ffit(a)+Tfit(b)
			true:min [Ffit(a), Ffit(b)]
			false:Tfit(a)+Tfit(b)

	a || b	*a&&b:min[Tfit(a)+Ffit(b), Ffit(a)+Tfit(b)]*
		a:Ffit(a)+Tfit(b)
			b:Tfit(a)+Ffit(b)
			true:Ffit(a)+Ffit(b)
			false:min[Tfit(a), Tfit(b)]

Let the initial random inputs be: *i = 150*, *j = 400*, *k = 300* and the target mutant the 15th one i.e. (*i < k* to *i <= k* with mutant fitness *abs*(*i – k*)). The process at first selects the *i* input variable and performs exploratory steps (small increases and decreases say *p* - here 1 for integer and 0.1 for real variables - of the input variable). These steps indicate the search direction. In the example here, *i* should be increased as it results in better fitness values. After the determination of the search direction the process continues with pattern steps (these steps are computed based on the formula: $2^n * direction * p$, where direction is 1 for increase or -1 for decrease). Thus, in the above example the next obtained input values (pattern steps) will be for the *i* variable *152*, *154*, *158*, *166*, *182*, *214*, *278*, *406*. At this point the fitness function can not be improved by further altering the *i* input variable as the fitness is also relies on the second branch point (*j < k*). The process continues with input variable *j*, it performs exploratory steps and starts to decrease the *j* value as follows: *398*, *396*, *392*, *384*, *368*, *336*, *272*. At that point the process chooses the *k* input variable and starts increasing its value accordingly to *302*, *304*, *308*, *316*, *332*, *364*, *428*. After value *428* it performs exploratory steps again and starts to decrease its value to *426*, *424*, *420*, *412*, *396*. Here it changes direction again and continues to *398*, *400*, *404*, *412* where it decreases to *410*, *408 404* and finally it finds the required value *406* that kills the mutant. The process has effectively achieved to produce the test case (*i = 406*, *j = 272*, *k = 406*) that kills the mutant (*i < k to i <= k*). In case this procedure fails to kill the required mutant the process starts on new randomly selected inputs for *i*, *j* and *k*. Of course, this could be a consequence of hitting a local minimum or a consequence of an equivalent mutant.

From the above example it should be clear that the undertaken approach is not affected by the presence of the non target mutants. This is a direct consequence of the searching activity based solely on the mutant control dependencies and not on the individual paths that reach them. In doing so the path explosion complexity problem of [10] is avoided.

E. Test case Minimization

Metallaxis tries to reduce the overall effort imposed by mutation testing. To this extend it employs a test set minimization process before returning the produced tests to the tester. This is done by taking advantage of the employed weak schemata technique, which efficiently executes and evaluates all the introduced mutants at every test execution run. Thus, all needed coverage (killed mutants) details about each of the generated tests have already been accumulated by the execution runs of the test data generation phase. Hence, existing test set minimization algorithms can be directly applied here, without requiring any additional execution effort. This fact can be seen as an advantage of the proposed approach in using weak mutation as opposed to strong. In the later case a huge number of additional execution runs is required, as a result of executing all mutants against all test cases.

The Metallaxis framework employs a greedy heuristic for minimizing the produced test sets. This heuristic forms a classical approximation algorithm [18] for the minimum set cover problem. The algorithm begins by initializing a set S of all the covered-killed by the test cases mutants. At each iteration, it selects the test case that covers-kills the highest number of mutants contained in S and removes them from S. The algorithm is repeated until S becomes empty. In section IV results of the achieved reductions are given.

III. CASE STUDY

In this section a preliminary evaluation of the Metallaxis framework was performed based on four program units. The conducted case study illustrates the ability and performance of the Metallaxis system to produce high quality (mutation based) test cases. Additionally, a comparison with random testing was performed in order to illustrate the framework's strengths. Table III presents some details about the selected program units. The selected programs have been widely used in mutation and search based testing studies i.e. [6], [9], [19] and [20].

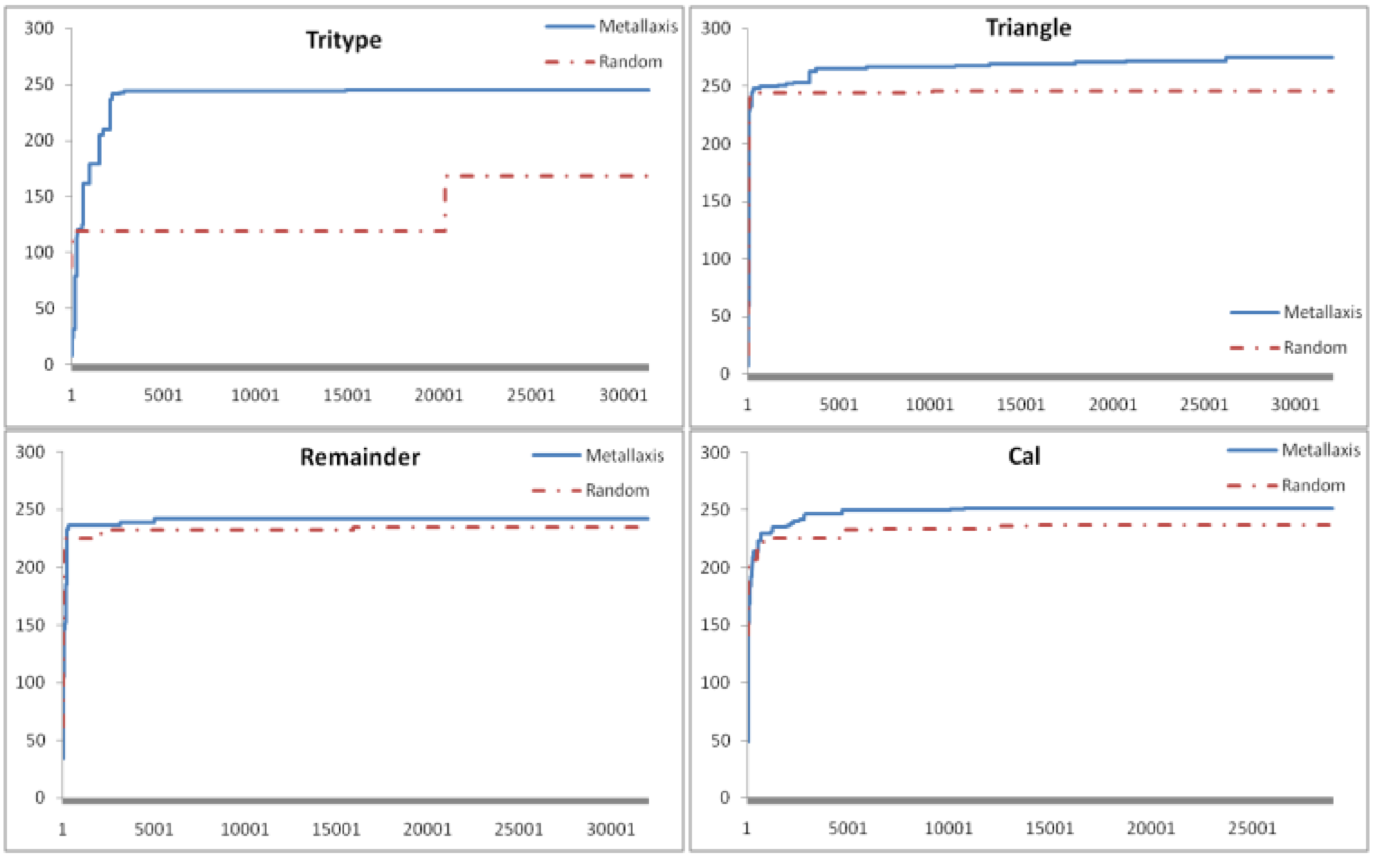

Figure 3. Mutants killed vs no. of iterations by Metallaxis framework and random testing

TABLE III. TEST PROGRAMS DETAILS

Test Objects	Lines of Code	Input settings	No. of Mutants
Tritype	40	3 integers: (range 16-bit)	349
Triangle	90	3 integers: (range 16-bit)	421
Remainder	50	2 integers: (range 16-bit)	324
Cal	75	5 integers: 2x[0, 12], 2x[0, 365], [-3,000, 3,000]	327

TABLE IV. CASE STUDY RESULTS

Test Objects	No. of Killed Mutants		Mutation Score		Time (Sec)
	Random	Metallaxis	Random	Metallaxis	
Tritype	168	245	66.4%	96.8%	42
Triangle	246	275	89.5%	100%	70
Remainder	238	242	98.3%	100%	402
Cal	237	252	93.3%	99.2%	44

The Metallaxis mutation testing framework system works on java programs at the intra-method level utilizing the mutation operators proposed by Offutt et al. [21] presented in Table II along with their fitness evaluations. The case study results along with a comparison to the random testing approach are presented in Figure 3 and Table IV. The conducted experiment was performed by employing Metallaxis to kill every introduced mutant with a maximum of 10 attempts per mutant. It must be noted that Metallaxis does not attempt to kill any of the previously killed by the produced test cases mutants at any stage. The Random test generation approach was performed on equal number of iterations to the required by Metallaxis framework execution number. The obtained results signify the ability of the Metallaxis framework to produce high quality test cases (as it is able to produce nearly mutation adequate test cases) in a small amount of time. Additionally, as it can outperform random testing its strengths are evident. Further, the obtain results of the test minimization are recorded in table V. These results indicate that it is possible to reduce the number of the produced test cases without loss on the achieved mutation score in with a minimum cost.

TABLE V. MINIMIZATION RESULTS

Test Objects	Produced test cases	Minimized test cases
Tritype	26	19
Triangle	23	20
Remainder	16	8
Cal	33	22

The above results indicate that the Metallaxis framework can be quite powerful at performing mutation. This is achieved by both producing and evaluating candidate test cases along with their fitness calculations, efficiently. From the above results one can observe that Metallaxis can achieve a considerable fast convergence at the early stages of the test generation process. This fact indicates that it may be possible for such a system to perform mutation on large industrial programs. This is a matter of future research.

IV. RELATED WORK

The Automatic generation of test cases has been an open research question since the early days of software engineering. Although being a well studied problem it has not yet been automated to an acceptable level for practical use. Recently, search based optimization techniques have succeeded in automating satisfactorily the test generation process. To this extend search based optimization techniques were used in constructing automated tools [17], [22] for most of the modern programming languages.

One of the first attempts of using search based optimization techniques was suggested by Korel [16]. Korel initially proposed the alternating variable method as described in section II.D.2, which was adopted and used by the Metallaxis framework as a search method for finding appropriate test cases. Daimler [14] developed an automated evolutionary tool for testing C programs based on various structural testing criteria. This approach targets on producing test cases based on path, branch, lcsaj paths, and data flow coverage criteria. It is this tool's fitness function that is extended and incorporated to the Metallaxis system for the purposes of mutation testing. A tool called eToc was build by Tonella [22] for the structural testing of java programs. This tool aims at producing effective method sequences by employing a genetic algorithm.

A different in philosophy approach to tackle the automated test data generation process uses paths and is based on symbolic execution [23] and constraint solving techniques in order to generate the required test data. Various automated tools have been built based on these approaches [6] and [24]. However, as they encounter the path feasibility problem, the handling of pointer aliasing and the handling of non-linear constraints their use has been limited.

Recently, dynamic approaches and symbolic execution were successfully combined in order to effectively automate the test data generation activity. This was achieved by a method called concolic or dynamic symbolic execution [24]. According to this approach, during actual program execution a set of constraints can be built and solved in order to produce test data. The difference with symbolic execution is that unhandled program expressions can be effectively simplified based on actual execution values. Based on this idea Lakhotia et al. [8] extend the alternating variable method to handle dynamic data structures and thus perform search based testing to a higher range of applications.

Harman and Mcminn [7] in a comprehensive theoretical and empirical study for search based optimization suggest that simple techniques such as hill climbing (alternating variable method) are mostly effective for structural test generation. Based on the above suggestions the present framework utilizes the hill climbing technique in order to effectively produce mutation based test cases.

All the above mentioned techniques nevertheless, do not deal directly with mutation testing and its inherent problems such as the handling of mutants on the one hand and their excessive number together with the equivalent ones on the other. This mater has received little attention in the literature towards automatically resolving it. The most fundamental attempt is the one due to DeMillo and Offutt who developed the constraint based testing Technique [6]. This technique introduced the concept of reachability, necessity and sufficiency conditions which has been embodied in a tool called Godzilla. Godzilla contains the first two of these conditions, formulating them as mathematical systems of constraints. Constraint based testing although empirically shown to be an effective technique it can not handle effectively arrays, loops and non linear expressions. Additionally, it fails to efficiently deal with the path explosion problem.

Dynamic approaches based on searching input domain sets have been proposed as a possible answer to the limitations of the above approach. Bottaci [25] proposed a fitness function composed of the reachability distance (measures the closeness of the test data and the mutant statement) of the produced tests and the necessity distance (measures the closeness to kill the mutant statement). In [19] an evolutionary approach for the generation of mutation test data was proposed. The authors suggest the employment of a search based minimization technique for generating test data. The search process is directed by an appropriate fitness function. Specifically, the authors adopt the ant colony optimization algorithm [19] as a metaheuristic search engine and an implementation of the Bottaci [25] fitness function as they implement only the reachability part. This approach is the closest one to the present proposed framework. The main differences are that the proposed framework extends the fitness function to effectively direct the search towards the constraint based testing necessity condition [6]. Additionally, the proposed framework presents a novel technique to efficiently gain the required fitness information through mutant schemata. Finally, the Metallaxis system forms an integrated mutation testing framework capable of: producing mutants, generating mutation based test data, determining the mutants killed and effectively minimizing the produced test sets due to mutation.

Another similar to the present approach is that of Baudry et al. [26]. In this work genetic algorithms are employed in order to augment an existing test suite according to mutation. The mutation score is used for performing fitness calculations. The drawback of this method is that it does not include any guidance to the search method by quantitatively measuring the closeness of killing specific mutants. Thus, making the search inefficient and even ineffective for killing many mutants.

The idea of utilizing mutant schemata in order to help automated tools to perform mutation was initially introduced in [10] with the aim of reusing existing structural automated tools for performing mutation. The underlying technique to achieve this, was to reduce the mutant killing problem to the covering branches one. This was performed based on mutant schemata. In [27] the schemata approach has been extended utilizing dynamic symbolic execution for producing strong mutation based tests. The extension utilizes a shortest path heuristic [28] in order to efficiently reach, infect and reveal the introduced mutants. Here this approach was extended in order to utilize an effective fitness function for employing search based optimization techniques.

Finally, a lot of effort has been put by the research community in order to reduce the test suite size produced during the testing process i.e. [29]. This activity is usually performed by aiming to maintain the same to the original test suite coverage. To this extend various heuristic algorithms have been proposed. All these algorithms can be directly applied in the context of mutation. The difference with the other testing criteria is that mutation requires a considerable amount of execution time and effort in order to determine the killed mutants by each test case. The approach proposed here uses a weak version of mutation in order to drastically reduce the amount of required time and also effort by the process.

V. DISCUSSION

The origin of the Metallaxis framework is due to the integrated use of mutant schemata and evolutionary testing techniques. This integration helps to extract dynamic program information concerning the introduced mutants and fitness calculations efficiently. Based on the dynamic nature of this scheme the needed information can be obtained with only one execution run. Additionally, the extend of which mutants are killed by each test case is also provided giving an advantage for employing test suite minimization techniques and the determination of which mutants have been killed unintentionally i.e. without targeting on them.

The conducted case study indicates the ability of the proposed method to produce high quality test cases from scratch (starting from random inputs) in a small amount of time. From this study it becomes evident that equivalent mutants pose an additional burden to test case evolution as they force the method not only to search for non killable mutants but also by misleading the mutation score calculated due to their presence. Perhaps the use of some heuristic approaches such as the one suggested by Bueno and Jino [30] for the identification of possible infeasible paths, could be employed in order to help overcome this problem by identifying likely to be equivalent mutants. This paper also reveals that simple dynamic approaches can be quite effective for the production of high quality test cases. Based on the dynamic nature of the adopted approach, the problems caused by pointers and non linear expressions are limited.

The framework described here uses a quite simple but practical approach, based on the mutant schemata technique [4], in order to perform the test data generation process. This approach adds some complexity to the underling test program (schematic program), as it adds program decisions (mutant evaluations), which are abstracted away by the employed fitness function and test evolution approach based on the program input domain.

A. Tool characteristics

The framework proposed in this paper employs a variant of weak mutation in order to guide the test data search effectively. In doing so, it achieves by a single execution path to decide which mutants are dead and which are alive. As it relies on complete execution paths, it may not be quite beneficial in search evolution phase as it does not stop the execution after some execution point as it is usually the case when performing weak mutation. However, it helps the test minimization process to proceed without requiring any extra executions. Additionally it helps to determine which mutants have been killed collaterally at each test evolution execution. This results in major savings especially at the beginning of the test evolution process and gradually falling as the evolution continues. To achieve further savings one could consider a mixed strategy, a matter of ongoing research.

B. *Limitations*

The proposed framework in this paper has several limitations which are currently under further research. First, it can handle java programs only at the intra method level. Thus, it does not handle method sequences or Object Oriented features. Second, it does not incorporate any flag removal or handling [31] mechanism, fact that makes inefficient the performed search in those specific cases. Finally, the inability of the mutant schemata technique to handle certain Object Oriented mutants as identified in [13]. Here it must be noted that in the case of Logical operators, a necessary special handling was enforced. This is due to the short circuit evaluation mechanism performed by java language. In order to keep the program execution paths unaffected with the presence of mutants, the logical operator’s evaluations were performed when both logical operands were executed.

VI. Conclusion and Future Work

The Metallaxis framework system, as described here forms an automation of the mutation testing method. The framework uses state of the art techniques to efficiently generate the candidate mutants and produce high quality test data. Based on a preliminary study the system achieves to produce test cases able to kill the majority of the introduced mutants. In future, extensions of the framework to include strong and higher order mutants [32], [33] are planned. Additionally, the employment of flag removal techniques and heuristic methods for detecting equivalent mutants are also considered. Finally, the integrated use of approaches considering method sequences and dynamic inputs such as the Tonella [22] approach and the Lakhotia et al. [8] are also examined.

References

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

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

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

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

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

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

[7] M. Harman and P. McMinn, "A theoretical and empirical study of search based testing: Local, Global and Hybrid Search," IEEE Trans. Softw. Eng., To appear.

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

[9] M. Papadakis and N. Malevris, "An Effective Path Selection Strategy for Mutation Testing," in Proceedings of the 16th Asia-Pacific Software Engineering Conference, 2009, pp. 422-429.

[10] M. Papadakis, N. Malevris, and M. Kallia, "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.

[11] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," IEEE Trans. Softw. Eng., vol. 8, pp. 371-379, 1982.

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

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

[14] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," Information and Software Technology, vol. 43, pp. 841-854, 2001.

[15] Z. Awedikian, K. Ayari, and G. Antoniol, "MC/DC automatic test input data generation," in Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Montreal, Canada, 2009, pp. 1657-1664.

[16] B. Korel, "Automated Software Test Data Generation," IEEE Trans. Softw. Eng., vol. 16, pp. 870-879, 1990.

[17] K. Lakhotia, M. Harman, and H. Gross, "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.

[18] V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," Mathematics of Operations Research, vol. 4, pp. 233-235, 1979.

[19] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in Gecco London, England, 2007, pp. 1074-1081.

[20] H. H. Sthamer, "The Automatic Generation of Software Test Data Using Genetic Algorithms," PHD thesis, University of Glamorgan / Prifvsgol Morgannwg, 1995.

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

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

[23] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT a formal system for testing and debugging programs by symbolic execution," SIGPLAN Not., vol. 10, pp. 234-245, 1975.

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

[25] L. Bottaci, "A genetic algorithm fitness function for mutation testing," in SEMINAL: Software Engineering using Metaheuristic INovative Algortithms, Workshop, 2001, pp. 3-7.

[26] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon, "From genetic to bacteriological algorithms for mutation-based testing: Research Articles," Softw. Test. Verif. Reliab., vol. 15, pp. 73-96, 2005.

[27] M. Papadakis and N. Malevris, "Automatic Mutation Test Case Generation Via Dynamic Symbolic Execution," in ISSRE, 2010.

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

[29] J. Offutt, J. Pan, and J. M. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets " in 12th Int'l Conf. Testing Computer Software 1995, pp. 111-123.

[30] P. M. S. Bueno and M. Jino, "Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data," in Proceedings of the 15th IEEE international conference on Automated software engineering, 2000, p. 209.

[31] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability Transformation," IEEE Trans. Softw. Eng., vol. 30, pp. 3-16, 2004.

[32] Y. Jia and M. Harman, "Higher Order Mutation Testing," Inf. Softw. Technol., vol. 51, pp. 1379-1393, 2009.

[33] M. Papadakis and N. Malevris, "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, 2010, pp. 90-99.