2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

# Mutant Execution Cost Reduction

Through MUSIC (MUtant Schema Improved with extra Code)

Pedro Reales Mateo and Macario Polo Usaola
Instituto de Tecnologías y Sistemas de Información
University of Castilla-La Mancha
Ciudad Real, Spain
{pedro.reales, macario.polo}@uclm.es

*Abstract*—**Mutation testing is a very effective testing technique that creates mutants (copies of the original system with small syntactic changes) in order to design test cases that kill the mutants (identifying the syntactic changes). The main disadvantage of mutation testing is its high costs: creating mutants, executing mutants and calculating the mutation score. This paper describes the MUSIC technique, an improvement of the cost reduction technique Mutant Schema, which is meant to reduce the number of required executions and identify infinite loops at a reduced cost. Besides, an experiment was performed to evaluate the advantages and disadvantages of MUSIC and identify possible ways to improved it. As a result, we conclude that MUSIC reduces the execution cost of mutation testing and its application is therefore recommended.**

***Keywords-Mutation testing, mutant schema, music, execution cost reduction, infinite loops.***

## I. INTRODUCTION

Software testing is an important task to develop systems, and sometimes is one of the most costly tasks. In order to increase the effectiveness of software testing, the scientific community has designed many techniques to improve testing tasks. One effective technique is mutation testing which was proposed by DeMillo[1] in 1978.

To apply this technique, a tester basically has to create copies of the system under test (SUT) and introduce small syntactic changes in each copy (these copies are called mutants). The syntactic changes are introduced through mutation operators, which are well formed rules to transform the code. Then, the tester has to create tests that must be able to find all the syntactic changes introduced by the mutation operators. When a test finds the change introduced into a mutant, the mutant is considered as killed. Sometimes, a syntactic change supposes the same behavior than the original system (as an optimization or a des-optimization) and thus these mutants cannot be killed by any test case. These mutants are named equivalent mutants, and usually must be manually identified. Finally, when the test cases are created, the mutation score () is calculated to determine the quality of the designed tests.

| | | |
|---|---|---|
| | | $P$ = program under test |
| | | $TS$ = test suite |
| $MS(P,TS) = K / (T - EQ)$ | *…where:* | $K$ = killed mutants |
| | | $T$ = total mutants |
| | | $EQ$ = number of test cases |

Figure 1. Mutation score formula.

Therefore, to perform a mutation analysis (for quality assessment of a test suite) we have to undertake three tasks: 1) mutants creation, 2) execution of our test cases against the original system and the mutants, and 3) calculation of the mutation score. The two first tasks are automatic, while the third is semi-automatic, mainly due to the presence of equivalent mutants, which usually are manually identified (although there are some semi-automatic strategies [2, 3]). This paper is focused in the improvement of the second task, the execution of test cases against the original system and the mutants.

Although the execution of tests is an automatic task, the execution time can be very high: let us imagine a system under test from which 1000 mutants are generated and a test suite with 200 test cases (a relatively small system). If each test requires 0.5 seconds to be executed, to execute all the test cases against the original system and the mutants will require a maximum of 200*0.5 + 1000*200*0.5 = 100 + 100000 = 100100 seconds = 151.66 minutes = 2.52 hours, which is too much time just to get the execution results.

Therefore, there is a need to develop techniques that reduce the execution time. This paper is focused on this issue. An improvement of an existing cost reduction technique (mutant schema [4]) is proposed. The improved technique has been called MUtant Schema improved with extra Code (MUSIC). This technique is able to reduce the execution requirements, thus reducing the cost of mutation analysis.

Furthermore, this paper presents an empirical validation of the proposed technique. It demonstrates that the proposed improvement has significant advantages.

The paper is organized as follows. The next section describes the strategies proposed in the literature to reduce execution costs of mutation testing. Section III describes mutant schema in depth. Section IV shows the proposed improvement of mutant schema (MUSIC). Section V describes the experiment performed to validate the proposed technique. Section VI shows the results of the experiment. And Section VII presents some conclusions and future work.

## II. State of the Art

Some authors have proposed different strategies to reduce the execution cost of mutation testing. The next paragraphs briefly describe the most important ones. More detailed information can be found in the survey of Jia and Harman [5].

Weak mutation strategies [6-9] reduce the execution costs by stopping the execution of mutants just after the execution of the mutated statement. There are some versions of this technique but all of them have in common the same advantages and disadvantages: on the one hand, they are effective in the reduction of the execution requirements because the complete execution of the tests is not necessary; but, on the other hand, weak strategies reduce the effectiveness of mutation testing because mutant are killed easier than with strong mutation, so they must be carefully used, taking into account their drawbacks.

Another strategy is to use advanced execution environment like clusters or grids [10, 11] in order to execute some mutants in parallel. These techniques reduce the total time of execution without reducing the effectiveness of mutation testing. However, it is necessary to have advanced environments and appropriated mutation tools for them, which is not always possible.

The third strategy proposed in the literature is related to the reduction of the generated mutants. Some techniques, like selective mutation [12], mutant sampling [13] or high order mutation [14], reduce the number of generated mutants, thus the execution requirements are also reduced. However, like with weak mutation strategies, the reduction of the generated mutants also reduces the effectiveness of mutation testing; therefore these techniques must also be used carefully.

Finally, there is another strategy based in the execution type. The first mutation tools were interpreted based [15]. These tools interpret the mutated code of each mutant in order to determine killed mutants. This technique was very costly, so in order to reduce the execution cost, a compiler based technique was proposed by Delamaro et al. [16]. This technique consists in compiling each mutant and executing test cases directly, instead of interpreting them. This is the technique that current mutation tools usually use.

## III. Mutant Schema

Another technique used to reduce execution costs is mutant schema [4], which is based in program schema technique [17]. The program schema technique was proposed by Baruch et al.[17]. This technique allows the designer to compose some different programs in the same source code: thus, only one compilation is required to obtain the executable of each program. As all programs are included in a single file, it is necessary to include a mechanism to determine which program included in the schema must be executed, such as a configuration parameter. An example of a program schema is show in Figure 2, where two programs ("add" and "sub") are composed into a schema and the parameter op is used to determine which program must be executed.

**Programs**

```
int add(int a, int b){
    return a+b;
}
```

```
intsub(int a, int b){
    return a-b;
}
```

**Program schema**

```
intaddsub (int op, int a, int b){
  if(op == 0){
    return  a+b;
  }else if(op == 1){
    return a-b;
  }
}
```

Figure 2. A program schema example.

In the context of mutation testing, program schema is known as mutant schema [4]. The use of program schema to represent mutants was proposed in 1992 by Untch et al. [4] in order to reduce the compilation time of mutants.

***Original***

```
1 class ClassA{
2 public intinc (int a, int b){
3       for(b<10){
4           a++;
5           b = b+2;
6       }
7       return a;
8}}
```

***Mutant 1***

```
1 class ClassA{
2 public intinc (int a, int b){
3       for(b<10){
4           a++;
5           b = b-2;
6       }
7       return a;
8}}
```

***Mutant 2***

```
1 class ClassA{
2  public intinc (int a, int b){
3       for(b<10){
4           a++;
5           b = b*2;
6       }
7       return a;
8}}
```

***Mutant 3***

```
1 class ClassA{
2 public intinc (int a, int b){
3       for(b<10){
4           a++;
5           b = b/2;
6       }
7       return a;
8}}
```

***Mutant Schema***

```
1 class ClassA{
2    public intinc(int a, int b){
3       for(b<10){
5           a++;
6           if(exec(m1)){
7               b = b-2;
8           }else if(exec(m2)){
9               b = b*2;
10          }else if(exec(m3)){
11              b = b/2;
12          }else {
13              b = b+2;//original statement
14          }
15      }
16      return a;
17}}
```

Figure 3. A mutant schema example.

In a mutation analysis, each mutant is a copy of the original system, but with a small syntactic change in one statement. Therefore, we can consider each mutant as a new program that only differs from the original in the mutated statement. Thus, a mutant and the original system can be composed into a schema (a "mutant schema"). For the running of tests against the original system or against the mutants, the execution environment only needs to properly set the configuration to run the correct "program" included in the schema.

Figure 3 shows an example of a mutant schema. The mutation operator AOR (Arithmetic Operator Replacement) was applied to the "inc" method (in order to make the example clear, the AOR operator was only applied to the statement 5 of the original system) and three mutants were created replacing the arithmetic operator "+" by "-", "*" and "/". Then the three mutants and the original program were combined into a single mutant schema. In the schema, all the mutated statements and the original statement are included and can be selected with an "*if-else*" statement. Also, the execution environment must provide an "*execute()*" operation to determine if a mutant must or must not be executed (Figure 3 shows a specific implementation of mutant schema, but more implementations are possible).

As Untch et al. [4] demonstrated in 1993, mutant schema reduces the compilation time (in the example of Figure 3 the compilation time will be reduced approximately four times, because only one program has to be compiled instead of four, although the program of the schema is bigger than each mutant). However, the technology is currently more efficient that in 1993. So, building a tool that implements mutant schema can be more expensive than the advantages of the technique.

Nevertheless, with the emergence of object-oriented technologies, the mutant schema technique does not only reduce the compilation time, but also the time of execution.

The example in Figure 3 is implemented in Java. Let us suppose that we do not use the mutant schema and the execution of test cases against each mutant and the original system is directly made. For each mutant, the execution environment has to load the *ClassA* class in memory (because ClassA is different in each mutant) and then, the test cases are executed (note that class loading is a costly task).

With mutant schema, the execution environment has to set the configuration properly to run the correct mutant and then, the test cases are executed. Thus the class loading task has to be performed just once in the first execution. This is the reason why some Java mutation tools like Mujava[18], Javalanche[19] or Bacterio[8] implement mutant schema for mutants generation.

## IV. MUSIC (MUTANT SCHEMA IMPROVED WITH EXTRA CODE)

Mutant schema can be used to improve the execution of mutants, but not only reducing the class loading tasks. We propose to use mutant schema (keeping their advantages) and include some extra code with two goals:1) identify situations where a mutant is not necessary to be executed and 2) identify when a mutation supposes an infinite loop and stop the execution. We have called this improvement of mutant schema technique MUSIC (MUtant Schema Improved with extra Code).

### *A. Reducing executions through mutant schema*

During a mutation analysis, it is possible that a test case is executed against a mutant, but the mutated statement is not executed because the execution path does not visit it. In this case, the mutant cannot be killed by the test and the execution of the pair "test case-mutant" is not necessary and supposes a waste of time. Although some mutation tools, like Javalanche [19], identify these situations, there is not a formal proposal to do that and no studies to evaluate its advantages have been done.

In order to identify these situations, we propose to introduce some extra code in the mutant schema. This code will be included before original statements and will identify the mutants generated from those statements.

Thus, when a test case is executed against the original system and a statement is reached, all the mutants generated from that statement are identified. This allows the execution environment to determine which mutated statements will not be reached by the test case. With this information it is possible to determine which test cases must not be executed against a particular mutant that will not be killed because its mutated statements will not be reached.

Figure 4 shows the mutant schema of Figure 3, but with extra code that identifies the mutants generated from the original statement (note that the execution environment must provide the method "*mutantGenerated()*").

```
1 class ClassA{
2   public intinc(int a, int b){
3     for(b<10){
5       a++;
6       if(exec(m1)){
7         b = b-2;
8       }else if(exec(m2)){
9         b = b*2;
10      }else if(exec(m3)){
11        b = b/2;
12      }else {
13        mutantGenerated(m1, m2, m3);
14        b = b+2; //original statement
15      }
16    }
17    return a;
18}}
```

Figure 4. Mutant schema with extra code to reduce executions.

Therefore, when we execute a test case against the original system and the statement 14 of Figure 4 is reached, we will know that m1, m2 and m3 mutants must be executed with the test case. In the same way, if we execute a test case against the original system and the statement 14 of Figure 4 is not reached, we will now that the test case must not be executed against m1, m2 and m3 mutants.

| Test Case 1 | Test Case 2 |
|---|---|
| a = 10, b = 20 | a = 1, b = 9 |

Figure 5. Tests case for the example.

Figure 5 shows an example of two test cases for the original system of Figure 3. Let us suppose that we decided that the order of execution is the design order, first TC1 and second TC2.

If we execute the test cases against the mutant schema of Figure 3, first we will execute TC1 and TC2 against the original system. Then, we will execute TC1 against all the mutants (m1, m2 and m3) and, as any mutant will be killed, we will execute TC2 against all the mutants. Thus, we will perform 8 executions.

However, if we execute the test case against the mutant scheme of Figure 4, first we will execute TC1 and TC2 against the original system and we will see that TC1 must not be executed against m1, m2 and m3 mutants because their mutated statements will be never reached; therefore we will execute only TC2 against m1, m2 and m3 mutants. Thus, we will perform 5 executions.

## *B. Identifying infinite loops through mutant schema*

Another problem in a mutation analysis is related to infinite loops created as a result of a mutation. This problem produces that test case executions never end with those mutants. Typically, mutation tools implement a "time-out" configuration parameter that determines the maximum time that can be spent by a test case. Thus, when a test case spends more time against a mutant, we can consider that the mutant contains an infinite loop, so the execution is stopped and the mutant is considered killed.

Sometimes this configuration parameter can be problematic and must be set carefully. It is possible that the execution of a test case against some mutants require more time that the configured time-out time. Some reasons can be small time-out configuration, processor too busy with a lot of processes, readings from disks too slow, network access, etc. Thus, some mutants can be considered killed but in fact, they should not. One solution to that problem is to set the time-out parameter to be long enough. However, a very long time-out can produce that to execute "infinite loop" mutants takes a lot of time. Thus, mutation analyses become more costly.

This problem was not treated in the literature but it can introduce small perturbations in the calculated mutation scores and increases the cost of a mutation analysis.

In order to solve this problem we propose to include some extra code in mutant schemas to identify infinite loops. The extra code is simply a loop counter. This "loop counter" will increase a configuration parameter of the execution environment so that the execution environment can determine how many iterations are executed in the original system, and discover soon if a mutant contains an infinite loop because it is executing too many iterations, independently of the time spent in the execution. This approach removes false deaths of mutants, and can reduce the execution of infinite loops.

Figure 6 shows the mutant schema of Figure 4, this time with the extra code to count iterations. In the m1 and m3 mutants, the parameter b will never increase up to 10, thus the loop will never end, thus, m1 and m3 contain an infinite loop. Also m2 contains an infinite loop if b is 0 or lower.

If we execute the test case TC2 of Figure 5 against the mutant schema of Figure 4, we will need to set up a time-out because m1 and m3 mutants will have infinite loops. Let us suppose that we setup the time-out as 1 second. Thus, we will spend 2 seconds to execute test case 2 against m1 and m3 mutants.

```
1 class ClassA{
2   public intinc(int a, int b){
3      for(b<10){
5         a++;
6         if(exec(m1)){
7            b = b-2;
8         }else if(exec(m2)){
9            b = b*2;
10        }else if(exec(m3)){
11           b = b/2;
12        }else {
13           mutantsGenerated(m1,m2,m3);
14           b = b+2;//original statement
15        }
16        increaseLoop();
17     }
18     return a;
19}}
```

Figure 6. Mutant schema with iteration conunter extra code.

Now let us suppose that we set up the time-out to 2 milliseconds once we have checked that the execution of TC2 against the original system spends 1.5 milliseconds. And now, let us suppose that we execute TC2 against m2, but before the statement 17 of Figure 4, the process scheduler suspends the execution of the test and other process is activated; and after 3 milliseconds, the process of TC1 is activated again. At this point the execution environment will detect that the execution has spent more than 2 milliseconds; the execution will thus be stopped and m2 will be considered as killed. TC2 is not able to kill m2 though.

However, if we execute the test case TC2 of Figure 5 against the mutant schema of Figure 6, after the execution of the original system we will know that the test iterates 1 time, so when the execution of TC2 against m1 or m3 iterates 5 times, the infinite loop is identified and the execution is stopped. Depending on how fast the hardware is, the execution of TC2 against m1 and m3 can spend some milliseconds, which is lower than 2 seconds. On the other hand, TC2 will never kill m2 because it will always iterate once and the result will always be 2 like the original.

## *C. Limitations of MUSIC*

Although the proposed approach to reduce the number of executions and identify infinite loops can reduce the execution time including extra code, it has some limitations.

The main limitations of MUSIC are derived from the limitations of the mutant schema technique. Some object-oriented mutation operators produce mutants that cannot be composed into a mutant schema with the original system. These mutants have syntactic changes that suppose a change in the structure of a class. Figure 7 shows a mutant generated with the AMC (Access Modifier Change) mutation operator [20].

In this case, it is necessary to create a new class and therefore, the advantages of mutant schema are not applicable to this kind of mutants. Fortunately, most of the object-oriented mutant operators generate mutants that can be composed into a mutant schema.

| ***Original*** | ***Mutant*** |
|---|---|
| class ClassB{<br>**private** int a**;**<br>….<br>} | class ClassB{<br>**public** inta;<br>….<br>} |

Figure 7. Mutant with a structural change.

Another limitation of MUSIC is related with the execution of the original system. With our proposal, extra code is included in mutant schemes that must be executed when the original system is executed. As a result, the execution of the original system becomes more costly. Besides, the execution cost savings of the proposed technique depend on the execution order of the test cases. Thus, there can be situations where the additional cost of the original system execution is bigger than the cost savings during the mutant executions.

The last limitation is related with the identification of infinite loops. Sometimes, a mutant can execute more iterations than the original system and it must not be killed. There are two possibilities: 1) the mutant is equivalent and the mutation supposes a de-optimization of the code; or 2) the mutant executes more iterations than the original system but the output is similar to the original output because the test data is not good enough to produce a different output. This implies that the number of iterations observed during the original system must be increased to avoid killing these kinds of mutants.

This means that MUSIC has similar problems to the "time-out" approaches, but the influence of the problems is lower. Increasing the number of iterations increases the execution cost less than allowing a longer time-out; thus, it is possible to increase the number of iterations too much in order to avoid killing mutants when they must not be killed.

## V. EMPIRICAL VALIDATION

This section presents an experiment that evaluates MUSIC. We performed an experiment with two different applications. The experimental design of the experiment is presented in this section.

### *A. Research goal*

The research goal of the experiment is to investigate if MUSIC improves the mutant execution task through the identification of infinite loops and situations when a mutant must not be executed.

### *B. Application under tests*

The experiment was run on two applications. Both of them are written in Java technology and have a test suite in JUnit[21] format. The applications were: Monopoly, a monopoly game simulator and Cinema: a cinema management system. A quantitative description of the applications and their test suites is showed in Table I.

Table I shows, for each application, the number of classes, the number of lines of code and the number of test cases that composes each test suite. Also, Table I shows the mutation score achieved by each test suite. To calculate the mutation scores, the equivalent mutants were not identified.

TABLE I. QUANTITATIVE DESCRIPTION OF THE APPLICATIONS.

| **App.** | **Classes** | **LOC** | **Number of tests** | **Mutation score** |
|---|---|---|---|---|
| Cinema | 10 | 678 | 197 | 80.05 |
| Monopoly | 40 | 641 | 211 | 83.32 |

### *C. Tools and hardware*

To manage the applications and the test suites we used the Eclipse platform [22]. To perform the mutation analysis and the mutant execution tasks, we used Bacterio tool [8], where MUSIC was implemented. To analyze the collected data, we used Microsoft Excel.

The computer used to perform the experiments has a processor Intel Core 2 Duo P7450, 4GB of RAM memory and Windows 7 operative system.

### *D. Variable description*

In the experiment, we manage seven different variables. There are two independent variables and four dependent variables.

The first independent variable is related with the application of the technique presented in section IV. This variable will have two values: use only mutant schema (MS) or use mutant schema improved with extra code (MUSIC). The second independent variable is the executions order. As is commented in section IV.C, the order of the test has a big influence in the advantages of the proposed technique, so it is important to evaluate the effect of different orders. Therefore, the test execution order will have five values: design order, design reverse order and three different random orders.

The first dependent variable is the total number of executions. It will show how many pairs "test-mutant" will be executed. The second dependent variable is the original execution time, measured in seconds. This variable will show MUSIC increases the original execution time relative to mutant schema. The third dependent variable is the mutant execution time. This variable will show if MUSIC was able to decrease the cost in the mutant executions. The fourth dependent variable is the number of mutants killed when an infinite loop is detected and the fifth dependent variable is the execution time when a mutant executes an infinite loop. These two variables will show the total time spent in mutants with infinite loops.

### *E. Experimental procedure*

The next paragraph shows the experimental procedure carried out for the experiment. Note that we have to create three different execution orders randomly for each application before performing the procedure.

There are 2 parameters to identify infinite loops: the "time-out" when mutant schema technique is used and the "iterations increases" when MUSIC is used (see section IV.C). For the time-out, we established 0.5 seconds, which is long enough not to show the problems addressed in section IV.B (also a second value, 0.1 seconds, for the time-out was established in order to show the problems described in section IV.B. The data collected when the time out is 0.1 seconds is only discussed in section VI.B). For the parameter of iterations increases we fixed it = *10, this configuration allows mutants with infinite loops to execute 10 times more iterations than the original system.

The procedure is composed by 9 steps and was executed several times, one for each combination of "application-order-technique". The experimental procedure has four input parameters: application, test suite, execution test order, and the technique to apply.

1- Create mutants of the application.

2- Set up Bacterio tool to apply the technique.

3- Set up Bacterio tool to run the test cases in the selected order.

4- Execute the test suite against the original system.

5- Annotate the time spent in the execution and the number of executions.

6- Execute the test suite against the generated mutants.

7- Annotate the time spent in the execution and the number of executions.

8- Annotate the number of killed mutants by time-out and time spent to kill them.

9- If the mutants were executed less than three times, go to step 4.

Finally, all collected data must be joined and analyzed. Note that all the information to be annotated is calculated and showed by the tool Bacterio, thus we only need to collect the data to analyze it.

At the end of the process, we will have executed three times the mutants and the original system, so we will have three repetitions for each combination "application-order-technique". Then the mean of the three repetitions is calculated. These three repetitions are necessary in order to remove possible random effects (such as processor scheduler or java garbage collector) that can influence in the execution time, and in order to observe the effect of the time-out (see section IV.B).

## VI. Results and Discussion

This section shows the experimental results. We analyzed three effects. 1) The effect of MUSIC in the execution tasks (number of executions and total time), 2) the effect of MUSIC in the identification of infinite loops, and 3) the effect of the test execution order when MUSIC is applied.

### A. *Effect of MUSIC in the execution task*

Figure 8 shows the number of executions, Figure 9 shows the execution time of the original system and Figure 10 shows the execution time of the mutants, all the tables for each application with each execution order and each technique (MS with the time-out = 0.5 seconds and MUSIC). The data is the mean of the three repetitions performed (see section V.E). The mean of three repetitions is calculated in order to remove possible randomness in the collected times.

Figure 9 shows that, when MUSIC is used, the original execution time is increased around four times in the worst case. This shows the problem described in section IV.C, the extra code that must be executed in the original execution increases the cost of the original execution.

However, Figure 10 shows that MUSIC is able to drastically reduce the mutant execution time. In the chart, we can see that the mutant execution time is reduced at least to a quarter when music is used. This reduction is due to the reduction in the number of executions (Figure 8), which are also reduced to a quarter.

This indicates that MUSIC increases four times the original execution time but it also decreases four times the mutant execution time. And, as the mutant execution time is some orders of magnitude bigger than the original execution time (10 seconds vs. 1000 seconds), the advantage of MUSIC is clearly stated.

We can conclude that the number of situations when a test case must not be executed because the mutated statements of mutants will not be executed is really high and suppose up to 75% of the total executions. Thus, a technique as MUSIC, which is able to identify these situations before the execution of mutants, can drastically reduce the execution requirements (around 75%); we strongly recommend its use, even if it supposes an increase of the original execution.

### B. *Effect of MUSIC in the identification of infinite loops*

This section shows the effect of MUSIC in the identification of infinite loops. Figure 11 shows the number of infinite loops detected (note that in some cases, when a "timeout" is used, the number of infinite loops is not an integer value. This indicates that, even executing the same mutants and test cases in the same order, infinite loops are identified only in one or two of the three repetitions) and Figure 12 shows the time spent in the execution of infinite loops, both tables for each application with each execution order and each technique (MS and MUSIC). This section also includes the data collected when the time-out parameter was 0.1 seconds.

Figure 11 shows the problem described in section IV.B concerning time approaches to identify infinite loops. When the time-out is too small some mutants may be killed by time-out when they should not. When the time-out was set up to 0.1 seconds, the number of infinite loops identified was higher than the infinite loops identified when the time-out is 0.5 seconds or when the MUSIC technique is used. This indicates that some mutants may be killed by time-out but they should not.

Table II shows the mutation scores achieved. We can see that, in fact, the mutation score achieved when the time-out is 0.1 seconds is higher that when the time-out is 0.5 or when MUSIC technique is used. This shows that when the configured time-out is too small, some mutants may be killed by time-out when they should not, therefore small time-out should not be used.

In addition, when a big time-out is used, the time expended to identify infinite loops can be very long. Figure 12 shows the time spent in the identification of infinite loops. It shows that when the time-out is 0.5 seconds, the spent time is much bigger than when the time-out is 0.1. In the experiment, the spent time is relatively small, but with larger tests this time can be much longer. Also, Figure 12 shows that MUSIC technique significantly reduces the time to identify infinite loops (around 15%). Moreover, Figure 11 and Table II show that MUSIC technique does not find more infinite loops than the time-based techniques and does not kill mutants when they should not be killed. Thus, these results induce to think that MUSIC is a more reliable technique than time-based approaches.

### C. *Influence of the execution order in MUSIC*

Finally, we analyze the influence of the test execution order in MUSIC technique. Figure 8 shows that the number of execution is different when the test execution order is different. When MUSIC is not used, the influence of the test execution order is obvious. For example, in the cinema application, we can find a difference of 80000 executions between direct and reverse orders, which is around 40%.

However, when MUSIC is used, the differences are small, because the number of executions is also small. For example the biggest difference is between reverse and random 2 test execution orders of cinema applications, which is around 25%.

Although the differences in the number of executions are lower when MUSIC technique is used, there are still some differences. Thus, it would be interesting to design test ordering strategies to improve the mutants execution and reduce the number of executions.

### D. *Threats to validity*

There are some threats to validity in this experiment that must be considered when we assume the conclusions.

Construct validity is the degree to which independent and dependent variables are accurately measured [23]. In the experiments, the independent variables are nominal and cannot be measured, and the dependent variables are measured objectively by tools. Times are not deterministic, so we perform three repetitions of each execution in order to reduce the threat. Besides, tools used in the experiment can have bugs, which is another threat.

Internal validity is the degree of confidence in a cause-effect relationship between factor of interest and the observed results [23]. All the variables were controlled during the experiments by tools in order to minimize threats to internal validity.

External validity is the extent to which the research results can be generalized to the population under study and other research settings[23]. In order to minimize threats to external validity, we used two applications with different nature, and each application with several classes. However, some threats exist because it is not possible to state that size and nature of the used applications is enough.

## VII. CONCLUSIONS AND FUTURE WORK

This paper proposes an improvement of an existing mutation cost reduction technique, mutant schema. This improvement, called MUSIC (mutant schema improved with extra code), is able to identify situations where a mutant must not be executed, therefore reducing the number of total required executions in a mutation analysis. In addition, MUSIC is able to identify infinite loops faster and more reliably than other approaches based in time.

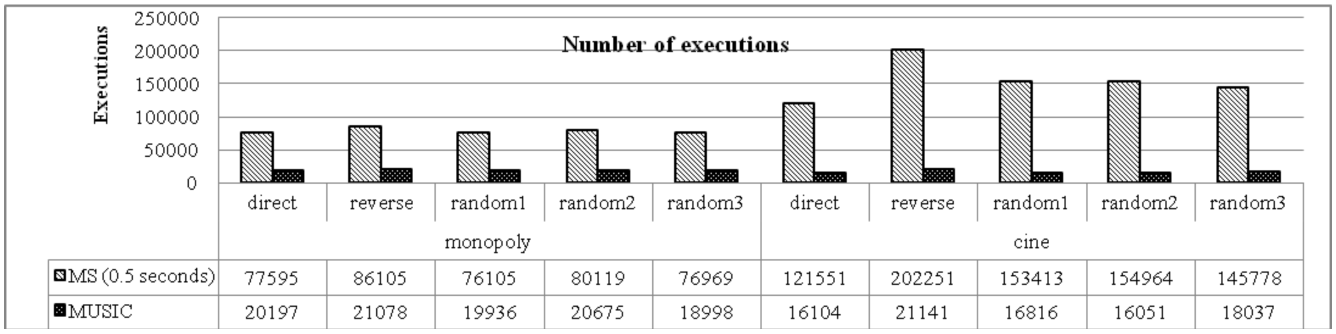
In this paper, an empirical validation of MUSIC is presented. This validation shows that MUSIC significantly reduces the number of executions (around 77%) and thus also the mutant execution time, to the detriment of the original execution time, which increases around 56%. The empirical validation also shows that the identification of infinite loops is faster and more reliable than other techniques based in time-outs. These conclusions suggest that MUSIC technique is convenient to reduce the execution time (although the execution of the original system increases significantly).

Another interesting conclusion of the experiment is related with the execution order. The experiments shows that the execution order of the tests has an influence the number of executions, and therefore the execution order influences the efficiency of MUSIC.

Taking all this into account, we plan future work to design strategies in order to determine good execution orders to maximize the efficiency of MUSIC technique.

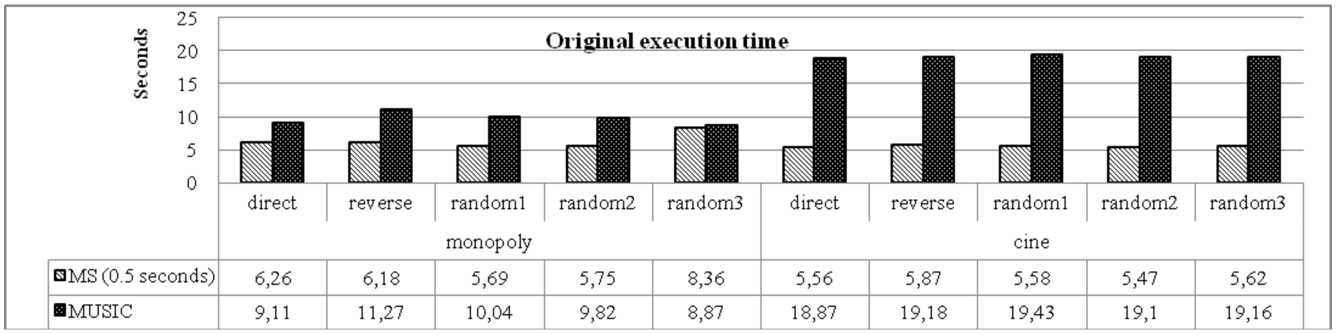
TABLE II. MUTATION SCORES

| **App** | **Test execution order** | **MS (0.5 seconds)** | **MS (0.1 seconds)** | **MUSIC** |
|---|---|---|---|---|
| monopoly | direct | 83.32 | 83.4 | 83.32 |
| | reverse | 83.32 | 83.43 | 83.32 |
| | random1 | 83.32 | 83.43 | 83.32 |
| | random2 | 83.32 | 83.35 | 83.32 |
| | random3 | 83.32 | 83.37 | 83.32 |
| cine | direct | 80.05 | 80.6 | 80.05 |
| | reverse | 80.05 | 80.08 | 80.05 |
| | random1 | 80.05 | 80.05 | 80.05 |
| | random2 | 80.05 | 80.05 | 80.05 |
| | random3 | 80.05 | 80.07 | 80.05 |



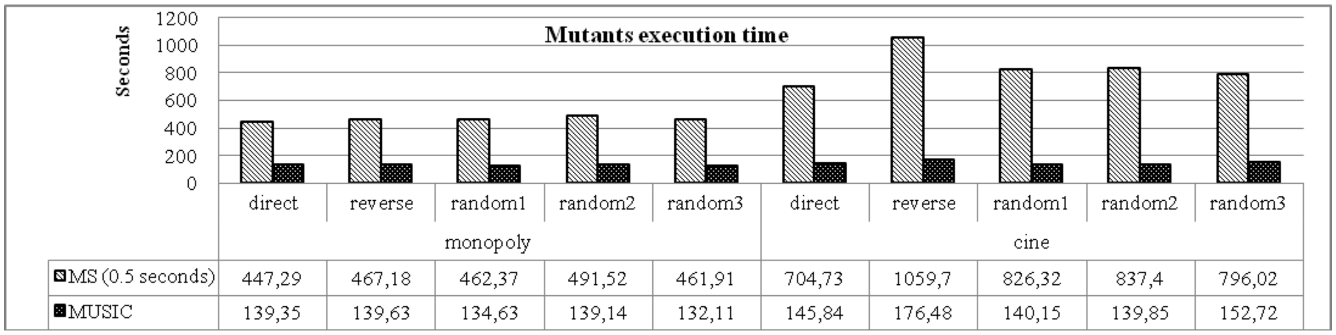
| | monopoly | | | | | cine | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | direct | reverse | random1 | random2 | random3 | direct | reverse | random1 | random2 | random3 |
| ▨MS (0.5 seconds) | 77595 | 86105 | 76105 | 80119 | 76969 | 121551 | 202251 | 153413 | 154964 | 145778 |
| ■MUSIC | 20197 | 21078 | 19936 | 20675 | 18998 | 16104 | 21141 | 16816 | 16051 | 18037 |

Figure 8. Number of executions (pairs test-mutant executed).



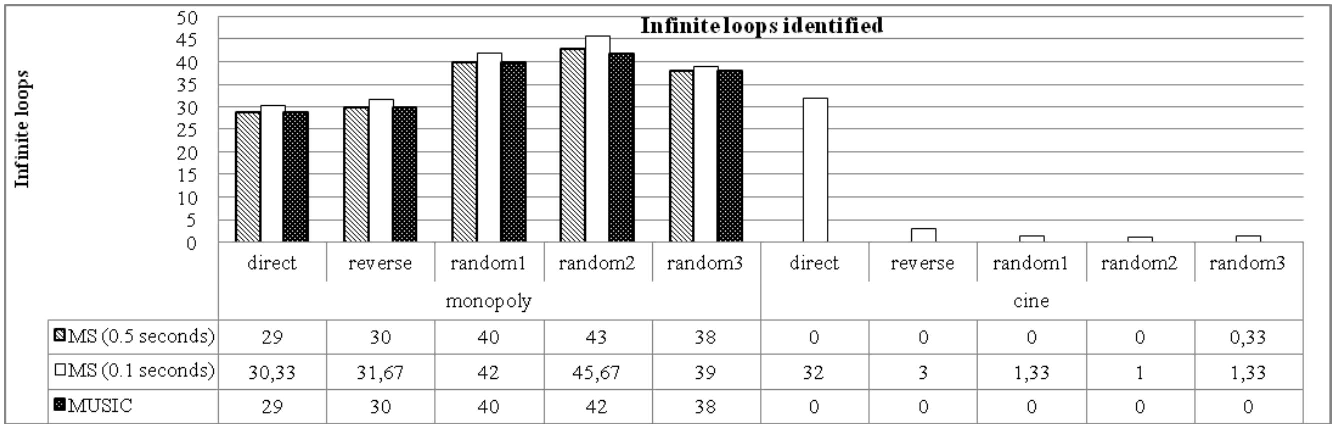
| | monopoly | | | | | cine | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | direct | reverse | random1 | random2 | random3 | direct | reverse | random1 | random2 | random3 |
| ▨MS (0.5 seconds) | 6,26 | 6,18 | 5,69 | 5,75 | 8,36 | 5,56 | 5,87 | 5,58 | 5,47 | 5,62 |
| ■MUSIC | 9,11 | 11,27 | 10,04 | 9,82 | 8,87 | 18,87 | 19,18 | 19,43 | 19,1 | 19,16 |

Figure 9. Original execution time.



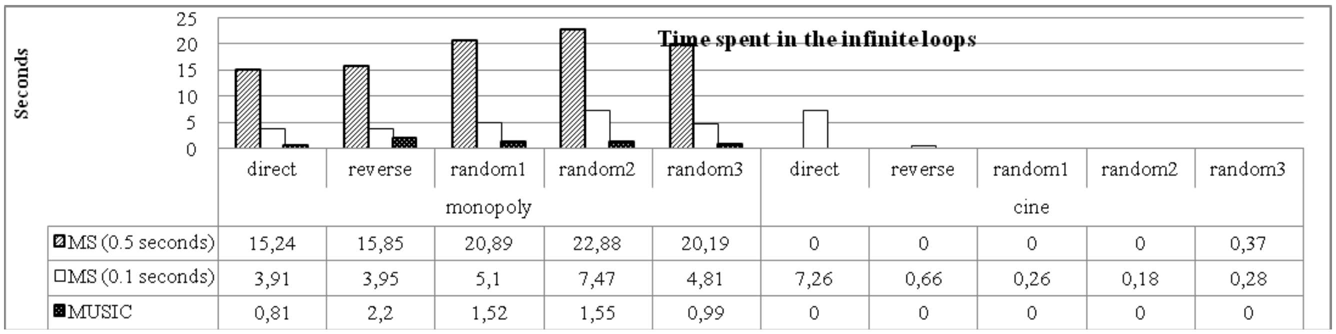
| | monopoly | | | | | cine | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | direct | reverse | random1 | random2 | random3 | direct | reverse | random1 | random2 | random3 |
| ▨MS (0.5 seconds) | 447,29 | 467,18 | 462,37 | 491,52 | 461,91 | 704,73 | 1059,7 | 826,32 | 837,4 | 796,02 |
| ■MUSIC | 139,35 | 139,63 | 134,63 | 139,14 | 132,11 | 145,84 | 176,48 | 140,15 | 139,85 | 152,72 |

Figure 10. Mutants execution time.



| | monopoly | | | | | cine | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | direct | reverse | random1 | random2 | random3 | direct | reverse | random1 | random2 | random3 |
| ▨MS (0.5 seconds) | 29 | 30 | 40 | 43 | 38 | 0 | 0 | 0 | 0 | 0,33 |
| □MS (0.1 seconds) | 30,33 | 31,67 | 42 | 45,67 | 39 | 32 | 3 | 1,33 | 1 | 1,33 |
| ■MUSIC | 29 | 30 | 40 | 42 | 38 | 0 | 0 | 0 | 0 | 0 |

Figure 11. Infinite loops identified.



| | monopoly | | | | | cine | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | direct | reverse | random1 | random2 | random3 | direct | reverse | random1 | random2 | random3 |
| ▨ MS (0.5 seconds) | 15,24 | 15,85 | 20,89 | 22,88 | 20,19 | 0 | 0 | 0 | 0 | 0,37 |
| □ MS (0.1 seconds) | 3,91 | 3,95 | 5,1 | 7,47 | 4,81 | 7,26 | 0,66 | 0,26 | 0,18 | 0,28 |
| ■ MUSIC | 0,81 | 2,2 | 1,52 | 1,55 | 0,99 | 0 | 0 | 0 | 0 | 0 |

Figure 12. Time spent in the infinited loops.

## ACKNOWLEDGMENT

This work was supported by the projects DIMITRI (TRA2009_0131) and the project PEGASO/MAGO (TIN2009-13718-C02-01) from MICINN. Pedro Reales has a doctoral grant (AP2009-3058) from the *Ministerio de Educación*.

## REFERENCES

[1] R. DeMillo, R.J. Lipton and F.G. Sayward. *Hints on test data selection: Help for the practicing programmer.* IEEE Computer. 11(4): p. 34-41, 1978.

[2] A. Offutt and J. Pan. *Detecting Equivalent Mutants and the Feasible Path Problem.* Wiley's Software Testing, Verification, and Reliability. 7(3): p. 165-192, September 1997, 1997.

[3] D. Schuler and A. Zeller. *(Un-)Covering Equivalent Mutants* In Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10). Paris, France, 6 April, 2010, 2010.

[4] R. Untch, A. Offutt and M. Harrold. *Mutation analysis using program schemata*. In International Symposium on Software Testing, and Analysis. 139-148, Cambridge, Massachusetts, June 28-30, 1993.

[5] Y. Jia and M. Harman. *An Analysis and Survey of the Development of Mutation Testing.* IEEE Transactions on Software Engineering. 37(5): p. 649-678, September 2011, 2011.

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

[7] B. Marick. *The weak mutation hypothesis*. In 4th Symposium on Testing, analysis, and verification. 190-199, Victoria, British Columbia, Canada, October, 1991, 1991.

[8] P. Reales, M. Polo and J. Offutt. *Mutation at System and Functional Levels*. In Third International Conference on Software Testing, Verification, and Validation Workshops. 110-119, Paris, France, April, 2010.

[9] M. Woodward and K. Halewood. *From weak to strong, dead or alive? An analysis of some mutation testing issues*. In Second Workshop on Software Testing, Verification, and Analysis. 152-158, Banff, Canada, July 1988, 1988.

[10] E.W. Krauser, A.P. Mathur and V.J. Rego. *High performance software testing on SIMD machine.* IEEE Transactions on Software Engineering. 17(5): p. 403-423, 1991.

[11] A.J. Offutt, R.P. Pargas, S.V. Fichter and P.K. Khambekar. *Mutation Testing of Software Using a MIMD Computer*. In International Conference on Parallel Processing. 1992.

[12] L.B. E. S. Mresa. *Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study.* Software Testing, Verification and Reliability. 9: p. 205-232, 1999.

[13] K.N. King and A.J. Offutt. *A Fortran language system for mutation based software testing.* Software: Practice and Experience. 21(7): p. 685-718, 1991.

[14] M. Polo, M. Piattini and I. García-Rodríguez. *Decreasing the cost of mutation testing with 2-order mutants.* Software Testing, Verification and Reliability. 19(2): p. 111-131, 2008.

[15] J. Offutt and K.N. King. *A Fortran 77 interpreter for mutation analysis*. In Programming Language Design and Implementation. 177-188, St. Paul, Minnesota, United States, 1987, 1987.

[16] M.E. Delamaro and J.C. Maldonado. *Proteum-A Tool for the Assessment of Test Adequacy for C Programs*. In Proceedings of the Conference on Performability in Computing Systems (PCS'96). 79–95, New Brunswick, New Jersey, July, 1996.

[17] O. Baruch and S. Katz. *Partially interpreted schemas for CSP programming.* Science of Computer Programming. 10(1), February, 1988.

[18] Y.-S. Ma, J. Offutt and Y.R. Kwon. *MuJava: an automated class mutation system.* Software Testing, Verification and Reliability. 15(2): p. 97-133, 2005.

[19] D. Schuler and A. Zeller. *Javalanche: efficient mutation testing for Java*. In European Software Engineering Conference (ESEC)/Foundations of Software Enginering (FSE). 297-298, Amsterdam, 2009.

[20] Y.S. Ma, Y.R. Kwon and J. Offutt. *Inter-class mutation operators for Java*. In 13th International Symposium on Software Reliability Engineering. 352-363, Annapolis, MD, 2002.

[21] JUnit. *JUnit, Testing Resources for Extreme Programming*. 2003 [Access January 5, 2003]; Available from: http://www.junit.org.

[22] *Eclipse Framework*. [Access date: 2011 10th February]; Available from: http://www.eclipse.org/.

[23] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. 2000, Norwell, MA: Kluwer Academic.