Mutation Testing: Guideline and Mutation Operator Classification

Lorena Gutiérrez-Madroñal
UCASE research group
University of Cádiz
Spain, Cádiz
Email: lorena.gutierrez@uca.es

Juan José Domínguez-Jiménez
UCASE research group
University of Cádiz
Spain, Cádiz
Email: juanjose.dominguez@uca.es

Inmaculada Medina-Bulo
UCASE research group
University of Cádiz
Spain, Cádiz
Email: inmaculada.medina@uca.es

***Abstract*—Mutation testing has been found to be effective to assess test suites quality and also to generate new test cases. In fact, it has been applied to many languages. Unfortunately there is no research work which focuses its attention on detect deficiencies in the mutation testing studies already done. Moreover, the mutation operators classification has not been tackled at all. When has a mutation testing study an enough grade of maturity? It has not been found an study which determines the grade of maturity of a mutation testing study. We propose a classification of mutation operators which lets us know if it is necessary to define more mutation operators, and also lets us determine if the mutation operators are good enough to consider mature the mutation testing study. The evaluation results show the benefits of the proposed classification. This classification lets the developer do a good evaluation of the programming language under mutation testing, as well as its defined mutation operators. The mutation testing process described in this paper and the mutation operators classification have been developed and analyzed with real cases. New mutation operators for Event Processing Language have been defined and used as examples to understand the proposed mutation operators classification.**

Keywords–Mutation testing; mutation testing guideline, mutation operators classification.

I. INTRODUCTION

With the success of the application of mutation testing to a number of common implementation languages [1]–[8], this paper presents the elaboration of a formal guide which lets the developer follow the right process to do a mutation testing study (MTS) of a new language, as well as a mutation operators classification to evaluate the grade of maturity of the MTS.

Mutation testing [9]–[11] is a fault-based testing technique providing a test criterion: *the mutation score*, which can be used to measure the effectiveness of a test suite in terms of its ability to detect faults. Mutants will be generated from applying *mutation operators* to the program under test. These mutation operators introduce slight syntactical changes into the program that should be detected by a high-quality test suite.

The majority of the mutant generation systems generate all the possible mutants, and normally include a vast array of mutation operators. Each mutation operator generates a large number of mutants which need to be run against the test suite to determine whether they can be told apart from the original program in some of its test cases (that means whether they are *killed* by the test suite or not). The entire process can take a long time for nontrivial programs.

Why are we proposing a guide to do an MTS? Nowadays, programming languages are adapted or created to cover new functionalities, solve problems or for new systems. Moreover mutation-based testing is a well-know and effective testing approach to reveal code-level vulnerabilities [9], [12]–[16] and has been applied for many traditional implementation languages, in which some deficiencies in mutation testing process have been found. The quality and efficiency of the programs, implemented or part-implemented with these program languages, could be proved and improved after a mutation testing study. For these reasons, we present a formal pattern for a MTS.

Each programming language under MTS should have a mutation operators classification. However, no research work which covers or studies mutation operators classifications has been found. Some definitions of the mutation operators are based on another ones because they have a similar behavior, in other cases the definitions are the same because these could be applied to languages with the same nature or the mutation operators definitions are about general changes that can be done whatever the nature of the programming language. In [17] a similar idea is proposed comparing mutation operators for Fortram, C and Java. One of Marthur's conclusions is *"A basic understanding of mutation operators, and experience in using them, helps a tester understand the purpose and strength of each operator. Further, such understanding opens the door for the construction of a completely new class of operators should one need to test an application written in a language for which no mutation operators have ever been designed!"*. That means that it is necessary to study the already defined mutation operators to avoid the duplication and define new mutation operators for new programming languages. The proposed classification helps, not only to organize the mutation operators, but also to determine if there exist mutation operators definitions with a similar behavior, and as consequence if it is necessary to define more operators to cover specific characteristics of the programming language.

It is not an easy task to determine the *grade of maturity* of an MTS. The mutation operators definitions are the key to evaluate the maturity of an MTS. A grade of maturity definition for mutation testing studies is proposed. That definition will make use of the classification presented also in this paper.

Following the study by Gutiérrez-Madroñal et al. [5], some new mutation operators for Event Processing Language (EPL) [18] have been defined in this paper. Their definitions

 ISBN: 978-1-61208-346-9

not only are included in the list of EPL mutation operators but also they help to explain the proposed mutant operators classification.

The structure of the rest of the paper is as follows: Section 2 introduces mutation testing and a description of the main steps involved in the process. Section 3 describes the process to follow to do an MTS, its goals and the relation between them, as well as the proposed classification of the mutation operators (explained with real and new mutation operators). In Section 4 are defined the new EPL mutant operators which are classified with the proposed classification. This classification is also applied and analyzed in Section 5 with mutation operators defined in different mutation testing studies. This section also includes the grade of maturity definition for mutation testing studies as well as the characteristic of its definition. And finally, in Section 6, the conclusions and future work are presented.

II. MUTATION TESTING BACKGROUND

Mutation testing [9]–[11] is a fault-based testing technique that introduces simple syntactic changes in the original program by applying mutation operators. Unlike other fault-based strategies that directly inject artificial faults into the program, the mutation method generates syntactic variations, *mutants*, of the original program by applying mutation operators. Each mutation operator represents "typical" programming errors, that the developer could make. For example, a mutation system replaces a relational operator (say $>$; i.e. $a > 26$) in the original program with other operators (such as $<, =, >=, <=$ and $<>$; i.e., $a < 26$), which is intended to represent a wrong instruction typed by the programmer. If a test case is able to distinguish between the original program and the mutant, it is said that this test case *kills* the mutant. On the contrary, if no test case in the test suite is able to distinguish between the mutant and the original program, it is said that the mutant stays *alive*. An *equivalent mutant* always produces the same output as the original program, hence it cannot be told apart from the original program. The general problem of determining if a mutant is equivalent to the original program is undecidable [19]. There are so many factors to take into account in order to find out if a mutant is equivalent. The code length, the mutant itself, and the variables used in the test case are good examples of it.

Mutation testing can be used in two ways. The first use is to measure the quality of a test suite with its *mutation score*, which is defined as the percentage of killed mutants. A formal definition:

$$ms = \frac{Km}{Tm - Em} \quad (1)$$

where Km is the number of killed mutants, Tm is the total number of mutants, and Em is the number of equivalent mutants. Normally the value of Em is not known, which is a problem because it is necessary to manually inspect the mutants to identify those that are equivalent. The second use for mutation testing is to generate new test cases in order to kill the surviving mutants, and thus improve the quality of the initial test suite. In an ideal case the mutation score reaches 100% which indicate that the test suite is adequate to detect all the faults modeled by the mutants.

One of the main drawbacks of mutation testing is the high computational cost involved. Commonly there is a large number of mutation operators that generate a wide numbers of mutants, each of them must be executed against the test suite. Under certain conditions described in an empirical study by Offut et al. [20], the number of mutants has quadratic complexity in program size.

From a theoretical point of view, the mutation testing process is divided in four main steps: analysis of the program, generation of mutants, execution of the mutants against the test suite and outputs analysis (Figure 1). Each process requires the previous results to be developed. And all of them, with the exception of the last one, output analysis, need a mutant system.

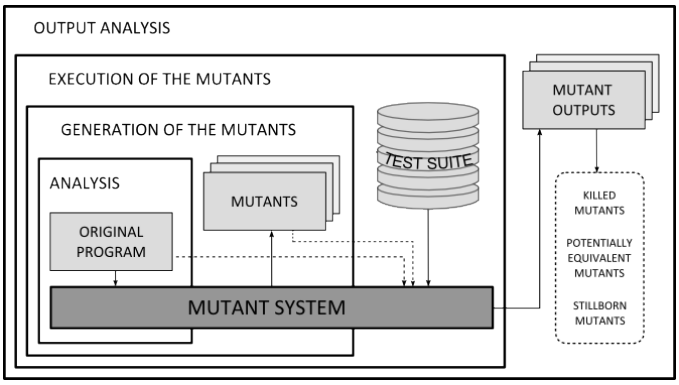

Figure 1. Phases of the Mutation Testing Process

The *analysis* can be defined as the part of the process where the programming language is studied. The analysis result is the number of occurrences of each mutation operator in the *program*. At this point it is necessary to clarify that program is used to denote the software under test, which could be a complete program or some smaller unit, such as a query. With this information, the mutant system moves to the next step, *generation of the mutants*. The mutant system does slight changes in the original program depending on the occurrences of each mutation operator and their definitions. For example, if the program under test were the next line of code "a + 6", the analysis result will say that there is one occurrence of an arithmetic operator. And if the mutation operator definition says that each occurrence has to be changed by one of the following set of elements: {-, /, * and %}, the number of generated mutants based on the previous conditions will be four. The *execution of the mutants* is the last task of the mutant system. At this point the mutants are executed against the test suite and just if their outputs can be distinguished between the mutant and the original program, the mutant can be classified as a killed mutant. The stillborn mutants also are automatically classified, because if the mutant fails to be run against the test suite or violates some static constraint defined by the language, the mutant system will notify it. The stillborn mutants fail to execute because of syntax errors or whatsoever conditions that the mutation operators producing them may have introduced. Finally in the *output analysis* the mutants are classified in order to check if the 100% in the mutation score has been reached.

 ISBN: 978-1-61208-346-9

In other case the test suite has to be improved.

III. MUTATION TESTING PROCESS

We propose a guide which describes a formal process to apply an MTS to a new language. Furthermore one of its steps includes the proposed classification for the mutation operators. Figure 2 shows the subtasks involved in each main step of the mutation testing process, our guideline describes them.

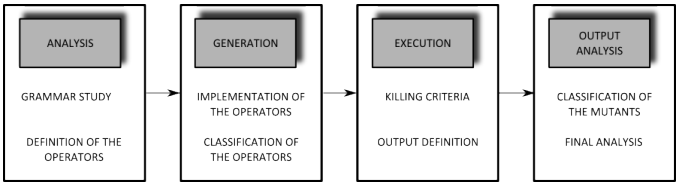

Figure 2. Subtasks of each Mutation Testing Step

A. Language selection

Depending on the programming language, some considerations should be taken into account. If the chosen programming language is based on another programming language, they should be compared (i.e., Event Processing Language (EPL) [18], an SQL-like language and SQL [5]), especially if this base programming language has an MTS. Otherwise, it should be recommendable start the MTS with the base programming language. If there is an MTS for the chosen programming language new contributions can be significant, i.e., some contributions may be done if the new MTS was focused on the changes of the latest version of the programming language. If the chosen programming language is not involved in any of the previous situations, it has to be in consideration its availability as well as its popularity. This means that it is a common programming language and it is easy to find open programs for a further study. Its availability and its popularity are important, because both will influence the MTS, i.e., it is a privative programming language which is used by a private company its grammar could be not accessible.

B. Grammar study

The programming language has to suffer an exhaustive survey. Each code line has to be considered and covered with all the possibilities. This part of the progress is linked with the next one because based on the grammar, the mutation operator definitions will be made. Studying the grammar let us know the right changes (mutations) that could be done according to the context. For example in a SQL query:

```
update table_A set field_A =
field_B * (select count (*) from table_B)
```

The first * correspond to an arithmetic operator, which could be replace by another +, -, / and %. On the contrary the last one is a wildcard that can not be replace by any of the last arithmetic operators. If this kind of situations were not considered, the resulting mutation programs would be syntactically wrong as well as the mutations would not represent common errors that a developer could make.

C. Definition of the operators

Before doing a formal definition, it has to be checked if there is a definition in other language that could be directly applied, or applied with a slight modification. After studying the grammar, the mutation operators can be defined. The operators have to represent typical programming errors such as: change the variable name for another, type a wrong (logical, arithmetic) operator, add or subtract one unit to a number, date or time, duplicate or forget a part of code and so on. The mutation operator definition has to explain how the mutation has to be done, and also each special situation that could happen. Sometimes, it is preferable to introduce an example for a visual understanding.

If the mutation operator has a similar behavior as another one that is already defined, it is preferable to use the same name but with a slight modification. For example, the SQL mutation operator “JOI, JOIN Clause”, which definition says [21]:

JOI; *JOIN Clause* - Each occurrence of a join-type keyword (INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, CROSS JOIN) is replaced by each of the others. When a join-type is replaced by CROSS JOIN, the search-conditions under the ON keyword are removed. When CROSS JOIN is replaced by another join-type, an ON clause is added and its corresponding join-condition is created based on the primary keys of the joined tables.

This definition is the base for ”EJOI” definition, a mutation operator of EPL (Event Processing Language) [5]:

EJOI; *JOIN clause* - It is applicable for INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. Each one of them is replaced by the others.

The name has been slightly modified because is a new definition but based on an existing one. In this case, the key word CROSS JOIN is not in EPL grammar.

D. Implementation of the operators

Once the operators have been defined, the next step is their implementation in the mutation system. These implementations have two elements. The first one is for detecting the operators in the program, which is included in the analysis step of the mutation testing process. Each mutation operator has a function which localizes each occurrence of the mutation operator in the program. This code is influenced by the studied grammar because it determines if the mutation operator is well located. The analysis output is the number of occurrences of each mutation operator. If in an analysis output the number of occurrences of a mutation operator is zero, this mutation operator is not applied.

The second element in the mutation operator implementation is the generation of the mutants. This code is influenced by the mutation operator definition, which says what have to be done to do the mutation. Every change generates a mutant, so if a definition of a mutation operator says that you can make five possible changes to the original instruction, five mutants will be generated. If the goal is to generate all the possible mutants, there has to be a control in order to generate the correct mutants as well as the exact number of mutants. Its output is a set of generated mutants.

 ISBN: 978-1-61208-346-9

E. Classification of the operators and SoMO classification

This part of the process has to be done after the definitions of mutation operators, but it would be recommendable to do it after their implementation as their behavior could no be the expected one.

The authors usually present their mutation operators grouping them by the nature of the change proposed; for example if the mutation operator changes values in the code, they are group in the category "Value mutation operators", if the changes are replacements, its category is "Replacement mutation operators" and so on. In this paper the approach goes a step further by proposing the SoMO (Sets of Mutation Operators) classification for the mutation operators which divides them in the following sets: *traditional*, *nature* and *specific*. The following lines describe each set and show an example for a better comprehension.

The first set of mutation operators is *traditional*. The mutation operators of this set can be found in the majority of the MTSs of any language. It is easy to discover the following mutation operators types: "Relational mutation operator", "Arithmetic mutation operator", "Logical mutation operator" or any other with a similar name. In some studies this set of mutation operators is called *traditional*, for this reason the SoMO classification uses the same name for this first set.

Other set of mutation operators to consider is the one which covers the mutation operators that do the changes in line/s of code according to the nature of the programming language. If the programming language is an object oriented one, is likely that the object oriented mutation operators will be found, such as "Heritage mutation operators" or "Polymorphism mutation operators" and so on. In the case of a query language, "Query mutation operators" will exist, for example mutation operators which affect *select clauses* or *where conditions*. The SoMO classification includes this type of mutation operators in the *nature* set.

And finally, the last set of mutation operators is the one which involves the mutation operators whose changes are based on the finality of the programming language. These types of mutation operators are common in new programming languages which have been created *recently* and/or have been created based on a *traditional* programming language. For example, the base or origin of query languages is SEQUEL (Structured English Query Language) [22], now known as SQL. The majority of query languages are based on SQL, in fact they are defined as SQL-like languages. Some of them are created for specific tools or functionalities like GQL (Google Query Language) created for developing tools which use the Google App Engine or the Google Cloud Datastore [23], [24], or EPL (Event Processing Language) for dealing with high frequency time-based event data [18], YQL (Yahoo! Query Language) to query, filter, and combine data from different sources across the Internet [25], and so on. If a study of mutation testing of these query programming languages is done, it is likely to find mutation operators which would be included in the first set as well as in the second set of mutation operators previously mentioned. But some of them can not be included in any of them because the changes are done in a specific part of code which can be only found in this programming language. These mutation operators are included in SoMO classification in *specific* set.

So the different sets of mutation operators (SoMO) are:

- *Traditional mutation operators*: They could be applied to any programming language no matter its nature.
- *Nature mutation operators*: They just could be applied according to the nature of the programming language. These operators are defined according to syntactic-like language faults.
- *Specific mutation operators*: They can not be applied to any other programming language. These operators are defined according to non syntactic-like language faults. That means that the mutations are done in the part of the language which differs from the rest of the languages.

What is the main contribution of the SoMO classification? SoMO classification is focused on the vulgarity or exclusivity of the mutation operators.

Lets define for each set of SoMO classification a GQL mutant operator (for the App Engine of Google). First set, *traditional mutation operators*:

GROR; *Google relational operator replacement* - Each occurrence of one of the relational operator $\{<, <=, >, >=, =, != \}$ is replaced by each of the other operators.

Relational operators can be found in several programming languages (their symbols could be different), no matter their nature.

For the second set of SoMO, *nature mutation operators*:

GSEL; *Google SELECT clause* - Each occurrence of one of the SELECT or SELECT DISTINCT keywords is replaced by the other.

This definition is the same that SEL mutation operator definition of SQL [21]. This reaffirms the necessity of SoMO classification, in this case it is not necessary the GSEL definition because there is a programming language with the same nature that already have that definition.

And for the third set of SoMO, *specific mutation operators*:

FRM; *From clause* - The optional FROM clause limits the result set to those entities of the given kind. A query without a FROM clause is called a kindless query and cannot include filters on properties. When this happens, the mutation consists in removing the FROM clause.

This mutation operator can just be applied to GQL because of its grammar and finality, so it is included in the *specific mutation operators* set.

What programming languages can be classified using SoMO classification? All the programming languages which are under an MTS can be classified by SoMO. If it is a base or traditional programming language such as Fortran, COBOL or Pascal, the *specific mutation operators* set will not have mutation operators. On the other hand, if it is a new programming language such as Ruby, Objective C, GQL, the *specific mutation operators* set will have mutation operators.

 ISBN: 978-1-61208-346-9

F. Killing criteria and output definition

This part of the study it is very important. If it is not done, there is just a list of definitions and generated mutants without any goal. In this section is where the conditions to consider when a mutant is killed are explained. There are two steps to develop the killing criteria:

1) According to the mutation operator definition, the mutant expected output should be determined.
2) Execute the original program and the mutant program and check if the mutant output is the expected one. If the mutant output is different, it is necessary to study the behavior of the mutant to determine the killing criteria.

The criteria to take a mutant apart depend on the output definition. For example, if EPL is the programming language for which the killing criteria wants to be done, it is necessary to take into account its nature. Let us consider the difference of the number of events as the EPL output definition. If the number of events obtained by the original program and the mutants are checked, the mutant outputs which differ in the time of response are equivalent according to the output definition, and this is fault. On the other hand, if the killing criteria is the latency of events between original and mutants, the mutant outputs which differ in the number of events are equivalent. So for EPL, the output definition must cover not only the number of events but also the latency between the original and the mutants.

It has to be defined a killing criteria for each mutation operator. Following with the EPL example, for the mutation operators which affect the time of response, their killing criteria are the latency between original and mutant, and for the ones which affect the amount of events in their output, their killing criteria are the differences in the number of events.

G. Mutant Classification

Once the killing criteria as well as the output definition are completed, it is time to execute all the mutants against a test suite which covers all the mutation operators defined before. Depending on the number of mutation operators defined and the generated mutants, the computational cost will be affected. When the execution is finished, the mutant outputs can be analyzed. Part of this analysis can be automated, in particular to determine if the mutant is killed, alive or stillborn. But it has also a manual part that is focused on the mutants that are still alive. This part consists in to discern whether these mutants are equivalent or it is necessary to improve the test suites. In [26] is described a technique called *constraint-based test data generation* which overcomes partially the equivalent mutant problem.

In the literature there are some definitions which will help the manual analysis in which the equivalent mutants have to be separated:

Equivalent mutant: A mutant is equivalent when there is not a test case which after applying it, the original program output and mutant program output can be differentiated. This means that there is not a single test case can kill the mutant.

Stubborn non-equivalent mutant: A mutant is stubborn non-equivalent when there is no test case in a test suite which after applying it, the original program output and mutant program output can be differentiated yet. This means that there is not an adequate test suite which detect the mutant, but it is not equivalent.

Equivalent mutants should not be confused with stubborn non-equivalent mutants, the set of equivalent and stubborn non-equivalent mutants is called *potentially equivalent mutants*. So if there are stubborn non-equivalent mutants in the study, the test suite has to be revised and a new test case which solves the stubborn non-equivalent mutants issue must be defined.

The automatic analysis can differentiate if the mutants are alive, killed or invalid. In the set of the killed mutants other definitions about their resistance have to be considered:

Weak mutant: A mutant is weak when is killed by every test case.

Resistant mutant: A mutant is resistant when is killed by a single test case.

Hard to kill mutant: A mutant is hard to kill when it is a resistant mutant, and the test case which kills it, just kills the resistant mutant.

This classification can be done automatically if in a matrix M (a row for each generated mutant R and a column for each test case of the test suite T), the status of the mutants after their execution are saved. The status can be considered as follows: 0 in $m_{i,j}$ means that the mutant in the i-row is alive for the test case in the j-column, 1 is use killed mutants and 2 for invalid mutants.

So it is easy to identify a weak mutant, because the value sum of its row is equal to the number of test cases in the test suite T used in the MTS:

$$\sum_{j=1}^{T} m_{i,j} = j$$

For a resistant mutant the sum of the elements of its row is equal to 1:

$$\sum_{j=1}^{T} m_{i,j} = 1$$

And finally a hard to kill mutant can be detected if the sum of the elements of its row and the sum of the elements of the column of the test case which kills it, are both equal to 1:

$$\sum_{j=1}^{T} m_{i,j} = 1 \qquad and \qquad \sum_{i=1}^{R} m_{i,j} = 1$$

H. Final Analysis

The previous mutant classification helps to discard some mutants that are not important in the MTS such as equivalent mutants and weak mutants. Detecting stubborn non-equivalent is a hard task because it has to be done manually. It is necessary to discover the path to the mutation and check the different modules and code lines which are affected because of the change, so in order to kill the mutant a new test case should be developed. To determine if a mutant is equivalent it is also necessary to do a manual analysis in the code, which has to be so exhaustive as the one for stubborn non-equivalent mutants because many stubborn non-equivalent mutants could be discarded.

 ISBN: 978-1-61208-346-9

IV. ANALYSIS OF REAL STUDY CASES

In [5] a set of mutation operators for EPL is presented, to obtain these results the six first steps of the described guideline were followed. The classification of the operators just covers their categories, so a SoMO classification have to be applied to differentiate the traditional, nature and specific mutation operators.

In this section, we first show a briefly background about EPL, then a discussion of new EPL mutation operators divided in categories is proposed. After their definition, mutation killing criteria for the proposed operators are presented. And finally, a SoMO classification, recovering the EPL mutation operators presented in [5].

A. Event Processing Language

Event Processing Language (EPL) is a SQL like query language. However, unlike SQL that operates on tables, EPL operates on continuous stream of events. As a result, a row from a table in SQL is analogous to an event present in an event stream. Example applications for EPL queries can be found in business process management and automation, finance, network and application monitoring and sensor network systems. These systems require processing of events in real-time.

Despite the fact that EPL is a SQL like language, considerable dissimilarities have been notice between EPL and SQL. Thus, in [5] were developed mutation operators due to specific features of EPL (pattern expression, sliding window of length and time, batch processing). Other point presented in [5] uses mutation testing to reveal vulnerabilities in event processing queries to assess the quality of input event streams. EPL was chosen as our case of study to develop mutation operators and killing criteria to generate high quality event streams and malicious inputs. In order to complete that study as well as to explain and to show the benefits of SoMO classification, new EPL mutation operators are proposed.

B. EPL Mutation Operators

Extension View Set Operators

SWVI; *Sorted Window View increase* - This view (ext:sort) sorts by values returned by the specified expression or list of expressions and keeps only the top (or bottom) events up to the given size. This view retains all events in the stream that fall into the sort range. The mutation consists in increasing the window size by one, so this will increase in one the specified events in the view. See an example in Table I.

SWVD; *Sorted Window View decrease* - The mutation consists in decreasing by one the number of the specified events in the view, so this will decreased by one the window size. See Table I.

SWVR; *Sorted Window View Replacement* - An expression may be followed by the optional *asc* or *desc* keywords to indicate that the values returned by that expression are sorted in ascending or descending sort order. The mutation consists in replacing each specified sort keyword by the other, or remove it. Table I shows an example of SWVR.

RWVI; *Ranked Window View increase* - This view (ext:rank) retains only the most recent among events having the same value for the criteria expression(s), sorted by sort criteria expressions and keeps only the top events up to the given size. This view is similar to the sorted window in that it keeps only the top (or bottom) events up to the given size, however the view also retains only the most recent among events having the same value(s) for the specified uniqueness expression(s). The mutation increases by one the number of specified events. An example in Table II.

TABLE I. SORTED WINDOW VIEW OPERATORS EXAMPLES

Original		SELECT sum(price) FROM StockEvent.ext:sort(10, price desc)
Mutant	SWVI	SELECT sum(price) FROM StockEvent.ext:sort(11, price desc)
	SWVD	SELECT sum(price) FROM StockEvent.ext:sort(9, price desc)
	SWVR	SELECT sum(price) FROM StockEvent.ext:sort(10, price)

RWVD; *Ranked Window View decrease* - The mutation decreases by one the number of the specified events in the view, so the window size is decreased by one. See Table II.

RWVR; *Ranked Window View Replacement* - The sort criteria expressions may be followed by the optional *asc* or *desc* keywords to indicate that the values returned by that expression are sorted in ascending or descending sort order. The mutation consists in replacing each specified sort keyword by the other, or remove it. See Table II for an example.

TABLE II. RANKED WINDOW VIEW OPERATORS EXAMPLES

Original		SELECT sum(price) FROM StockEvent.ext:rank (symbol, 8, price desc)
Mutant	RWVI	SELECT sum(price) FROM StockEvent.ext:rank (symbol, 9, price desc)
	RWVD	SELECT sum(price) FROM StockEvent.ext:rank (symbol, 7, price desc)
	RWVR	SELECT sum(price) FROM StockEvent.ext:rank (symbol, 8, price asc)

TOVI; *Time-Order View increase* - This view (ext:time_order) orders events that arrive out-of-order, using timestamp-values provided by an expression, and by comparing that timestamp value to engine system time. The mutation increases the specified timestamp by one second.

TOVD; *Time-Order View decrease* - The mutation decreases the specified timestamp by one.

Table III shows examples of the previous mutation operators.

TABLE III. TIME-ORDER VIEW OPERATORS EXAMPLES

Original		SELECT * FROM TimeEvent.ext:time_order (arrivalTime, 8 sec)
Mutant	TOVI	SELECT * FROM TimeEvent.ext:time_order (arrivalTime, 9 sec)
	TOVD	SELECT * FROM TimeEvent.ext:time_order (arrivalTime, 7 sec)

Pattern Expression Operators

NRK; *Not in regexp Keyword* - The regexp keyword is a form of pattern matching based on regular expressions. The result of a regexp expression is of type Boolean. If the input value matches the regular expression, the result value is true. Otherwise, the result value is false. The mutation removes or inserts the keyword 'not' before the keyword 'regexp'. In the Table IV there is an example.

Operator replacement

 ISBN: 978-1-61208-346-9

TABLE IV. NRK OPERATOR EXAMPLE

Original	SELECT * FROM PersonEvent WHERE name not regexp '.*Jack.*'
Mutant	SELECT * FROM PersonEvent WHERE name regexp '.*Jack.*'

FRR; *Filter Ranges Replacement* - Ranges come in the following 4 varieties. The use of round () or square [] bracket dictates whether an endpoint is included or excluded. The low point and the high-point of the range are separated by the colon : character. The mutation consists in changing a round bracket by a square one and viceversa, the possible outputs: {(), (], [) and []}. Table V shows an example of FRR.

TABLE V. FRR OPERATOR EXAMPLE

Original	mypackage.Event(x not in [0:100])
Mutant	mypackage.Event(x not in [0:100))

SQL Injection Attack Operators

LCR; *Limit Clause Remove* - The limit clause is used to limit the query results to those that fall within a specified range. This operator removes the limit clause in the query. The Table VI has a LCR example.

TABLE VI. LCR OPERATOR EXAMPLE

Original	select age, count(*) from PersonEvent group by age order by count(*) desc limit 10
Mutant	select age, count(*) from PersonEvent group by age order by count(*) desc

OPR; *Offset Parameter Remove* - The limit clause has an optional offset parameter which specifies the number of rows that should be skipped at the beginning of the result set. This operator removes the offset parameter in the query. Table VII shows an example of the OPR mutation operator.

TABLE VII. OPR OPERATOR EXAMPLE

Original	select age, count(*) from PersonEvent group by age order by count(*) desc limit 10 offset 2
Mutant	select age, count(*) from PersonEvent group by age order by count(*) desc limit 10

Killing Criteria

The Table VIII shows the list of operators and their corresponding killing criteria. The total number of events reported by an original query *(O)* and a mutated query *(M)* are compared in order to decide if a mutant is killed or not. A mutant is killed if the number of reported events is not equal to the original query. Some operators (SWVR, RWVR) require defining a different killing criterion; in these cases a mutant is killed if the order of the reported events by the original and the mutated query is different.

C. EPL SoMO classification

The EPL SoMO classification is showed in Table IX (including the new EPL mutation operators and the defined in [5]). The first column shows the sets, the second column shows the mutation operators which are included in each set and the third shows the percentage of operators involved.

TABLE VIII. KILLING CRITERIA

Category	Operator	Killing criteria
EVS	SWVI, SWVD, RWVI, RWVD, TOVI, TOVD	Event count between O and M.
	SWVR, RWVR	Event sort between O and M.
PEP	NRK	Event count between O and M.
SQIJ	LCR, OPR	Event count between O and M.

TABLE IX. EPL SoMO CLASSIFICATION

Set	Mutation Op.	% Op.
Traditional	RLOP, OEDIP, OEDDP, LINC, LDEC, TINC, TDEC, TRUN, SQNC, SQUP, EABS, EAOR, EROR, EUOI, ENLF, ENLI, ENLO, TOVI, TOVD, RWVI, RWVD, SWVI, SWVD, NRK, FRR	41'67
Nature	SQRC, SQFD, EAGR, ESEL, EGRUE, EGRUA, EJOI, EORDE, EORDK, EORDS, ESUBRI, ESUBRII, ESUBRIII, ESUBIBII, ESUBIBIII, ESUBIIBI, ESUBIIBIII, EBTW, ELKEWC, ELKEWR, ELKECA, ELKECB, ELKEAE, ELKEAB, LCR, OPR	43'3
Specific	RREP, CEOP, ESIRF, ESIRL, RGEP, BATL, BATT, SWVR, RWVR	15

The EPL SoMO classification includes more than 40% of the mutation operators in both the nature and the traditional set and a 15 percent in the specific set.

V. SoMO CLASSIFICATION ANALYSIS FOR OTHER STUDIES

Lets do the SoMO classification with the following mutation testing studies: XML Schema [27], [28], HTML [29], JSP [29], WS-BPEL 2.0 [30], [31], SOAP [32], PHP [33] and XACML [34]. All of them have something in common with markup languages: are markup languages, are like a markup language or are written in markup languages.

Table X shows the percentage of mutation operators classified in the different sets. The first column is the name of the domain in which the study has been done, the second, third and fourth are the percentage of mutation operators in each set respectively and finally the total number of mutation operators.

TABLE X. SoMO CLASSIFICATION FOR OTHER STUDIES

Domain	% Set			Total
	Traditional	Nature	Specific	
XML	17'9	82'1	0	28
HTML	0	100	0	8
JSP	0	0	100	3
WS-BPEL	38'24	26'47	35'3	34
SOAP	25	0	75	7
PHP	86'4	13'6	0	44
XACML	0	36'4	63'6	11

As it was expected the traditional and base markup languages XML and HTML have not any mutation operators in their specific set. The newest programming languages have the highest percentage of mutation operators in the specific set, with the exception of PHP which has the majority of its mutation operators in the traditional set. According to the PHP study it is an ongoing work and more mutation operators will be added over the time, so it seems to be a huge MTS. Another point to highlight on the PHP study is the 13'6% value in its nature set, this percentage is not because the markup language nature, it is because the influence from Java, C, C++, Perl and

 ISBN: 978-1-61208-346-9

Python. PHP MTS needs definitions for mutation operators of the specific set because these are the important and interesting ones, in other case this study would be incomplete given the fact that the mutation operators already defined do not contribute to a PHP MTS.

JSP is a programming language based on a markup language and SOAP is a protocol written using a markup language, but as we can see, they have no mutation operator in their nature set. The reason of the non-existence of mutation operators in the nature set could be because their authors considered them already defined, or maybe, the study was not focused on defining a set of mutation operators. However, in contrast to JSP and SOAP, WS-BPEL and XACML studies have mutation operators in their nature set. The ones which belong to this set are those whose changes are done in a part of the syntax-like markup language (i.e., remove targets, change the targets order).

The WS-BPEL study is the only one which contemplates mutation operators in all the sets, this is because this study is a complete MTS and WS-BPEL is a new programming language. As it was said in section 3.5, if a programming language is new, its specific set will have mutation operators, in other case (it is not a new programming language) its specific set will have no mutation operators.

In the majority of the cases, it is very difficult to determine if all the mutation operators in an MTS have been defined, although the interesting thing to know is if the already defined mutation operators are good enough to consider the MTS mature. The number of defined mutation operators and the SoMO classification let us define the grade of maturity of an MTS. In order to give a grade of maturity definition, some considerations have to be taken into account. Due to WS-BPEL MTS has an exhaustive, huge and tested mutation operators list and MuBPEL [4], a mutation tool for WS-BPEL language which has been used to verify this study, WS-BPEL study can be considered a finished and mature MTS. Considering WS-BPEL a mature MTS and after studying its SoMO classification, a grade of maturity definition of an MTS can be explained:

Grade of maturity of an MTS: The grade of maturity of an MTS is measured with its SoMO sets. When the percentage of its traditional set is about 40%, and the sum of the percentages of its nature and specific sets is about 60%, it can be said that the defined mutation operators are good enough to consider the MTS mature.

Why these percentages? When an MTS is started, it is normal to begin with traditional mutation operators definitions. These mutation operators are easy to identify and they are already considered as common mistakes that a developer could make. On the other hand, nature and specific sets can not be separated because, depending on the similarity of the programming language to the one in which is based on, these percentages can change. If a programming language has a high percentage of mutation operators in the nature set that means that the original programming language and the new one are very similar. But if the new programming language syntax does not allow those changes, it means that the languages are not so similar.

According to the grade of maturity definition for an MTS, the previous mutation operators definitions for the EPL MTS described in this paper, we can consider it as a mature MTS (see the percentages in Table IX). The EPL MTS has a vast mutation operators list and satisfies the grade of maturity definition of an MTS (About 40% of mutation operators belong to the traditional set and the rest of them belong to the nature and specific sets). In accordance with the grade of maturity definition, the XML study needs mutation operators of the traditional set to be a mature MTS. On the other hand, the PHP study needs mutation operators of the nature and specific sets (as it was previously indicated). The rest of MTSs have a low number of mutation operators definitions so this definition can not be applied to them.

If a comparison is made between the SoMO classifications of WS-BPEL and EPL, as the percentage of the nature set in EPL is higher than in WS-BPEL, it can be stated that EPL programming language is more similar to SQL than WS-BPEL to XML.

VI. Conclusion and Future Work

Doing an MTS is an exhaustive task and it is necessary to know the steps to follow and their goals. Due to the computational cost which involves the execution of the mutants, it is interesting to have no errors along the mutation testing process which can affect the results. This paper proposes a mutation testing guideline, with real examples in each step (including mutation operators definitions for EPL), and their goals. Moreover, the SoMO classification divides the mutation operators in traditional, nature and specific sets, and it also determines if the MTS has a enough grade of maturity. The percentage of each SoMO set determines not only what kind of programming language is under study, but also if it is an ongoing study. This helps us to determine what set of mutation operators need to be completed.

The definitions of new EPL mutation operators have been used to explain one of the steps of the mutation testing process (mutation operator classification), as well as to clarify how SoMO classification has to be applied. Adding these operators to the EPL MTS, this MTS meets the grade of maturity definition and indicates that it is a mature MTS.

In future, we plan to finalize the EPL MTS in which an EPL tool will be developed. This tool will help to determine if mutation-based testing reveals code-level vulnerabilities not only in traditional implementation languages, but also in event processing queries. This process assesses the quality of input event streams and generates event streams that can reveal implementation bugs in queries.

The presented grade of maturity definition for an MTS needs to be checked with other real mutation testing studies to verify and adjust (in case to be necessary) the proposed percentages for SoMO classification. This definition will be polished applying the SoMO classification in other MTSs which are focused in different programming paradigms, such as query languages or object oriented languages. This will help to accurate the proposed percentages of the grade of maturity definition.

 ISBN: 978-1-61208-346-9

ACKNOWLEDGMENT

This work has been funded by the Ministry of Science and Innovation (Spain) under the National Program for Research, Development and Innovation, Project MoDSOA TIN2011-27242.

REFERENCES

[1] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in Proceedings of the Fifth International Conference on Quality Software, ser. QSIC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 187–196. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2005.27

[2] M. E. Delamaro and J. C. Maldonado, "Proteum - a tool for the assessment of test adequacy for C programs: User's guide," in PCS'96: Conference on Performability in Computing Systems, 1996, pp. 79–95.

[3] S. Hussain, "Mutation clustering," Master's thesis, King's College London, 2008.

[4] A. García-Domínguez, A. Estero-Botaro, J.-J. Domínguez-Jiménez, I. Medina-Bulo, and F. Palomo-Lozano, "Mubpel: una herramienta de mutación firme para ws-bpel 2.0," in Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos, A. Ruiz and L. Iribarne, Eds., Almería, Spain, 2012. [Online]. Available: http://sistedes2012.ual.es/sistedes/jisbd

[5] L. Gutiérrez-Madroñal, H. Shahriar, M. Zulkernine, J. Dominguez-Jimenez, and I. Medina-Bulo, "Mutation testing of event processing queries," in Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, 2012, pp. 21–30.

[6] K. N. King and A. J. Offutt, "A FORTRAN language system for mutation-based software testing," Software - Practice and Experience, vol. 21, no. 7, 1991, pp. 685–718.

[7] Y. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," Software Testing, Verification and Reliability, vol. 15, no. 2, 2005, pp. 97–133.

[8] J. Tuya, M. Suarez-Cabal, and C. de la Riva, "Sqlmutation: A tool to generate mutants of sql database queries," in Mutation Analysis, 2006. Second Workshop on, 2006, pp. 1–1.

[9] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," Computer, vol. 11, no. 4, 1978, pp. 34–41.

[10] R. Hamlet, "Testing programs with the aid of a compiler," Software Engineering, IEEE Transactions on, vol. SE-3, no. 4, 1977, pp. 279–290.

[11] M. Woodward, "Mutation testing - its origin and evolution," Information and Software Technology, vol. 35, no. 3, 1993, pp. 163 – 169. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0950584993900536

[12] Y. Jia and M. Harman, "Higher order mutation testing," Information and Software Technology, vol. 51, no. 10, 2009, pp. 1379–1393.

[13] J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," Information and Software Technology, vol. 53, no. 10, 2011, p. 1108–1123. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058491100084X

[14] A. Estero-Botaro, J. Domínguez-Jiménez, L. Gutiérrez-Madroñal, and I. Medina-Bulo, "Evaluación de la calidad de los mutantes en la prueba de mutaciones," in Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos, A Coruña, Spain, 2011.

[15] M. Polo Usaola and P. Reales Mateo, "Mutation testing cost reduction techniques: A survey," Software, IEEE, vol. 27, no. 3, 2010, pp. 80–86.

[16] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, 2005, pp. 402–411.

[17] A. P. Mathur, Foundations of Software Testing, 1st ed. Addison-Wesley Professional, 2008.

[18] EsperTech, "Event processing with wsper and nesper," Last access nov, 2013. [Online]. Available: http://esper.codehaus.org/

[19] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," ACM Comput. Surv., vol. 29, no. 4, Dec. 1997, pp. 366–427. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[20] A. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in Software Engineering, 1993. Proceedings., 15th International Conference on, 1993, pp. 100–107.

[21] J. Tuya, M. Suarez-Cabal, and C. de la Riva, "Mutating database queries," Information and Software Technology, vol. 49, no. 4, 2007, pp. 398 – 417. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584906000814

[22] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, ser. SIGFIDET '74. New York, NY, USA: ACM, 1974, pp. 249–264. [Online]. Available: http://doi.acm.org/10.1145/800296.811515

[23] Google, "Google query language in google app engine," Last access nov, 2013. [Online]. Available: https://developers.google.com/appengine/docs/python/datastore/gqlreference

[24] ——, "Google query language in google cloud datastore," Last access nov, 2013. [Online]. Available: https://developers.google.com/datastore/docs/concepts/gql

[25] Yahoo!, "Yahoo query language," Last access nov, 2013. [Online]. Available: http://developer.yahoo.com/yql/

[26] A. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on, 1996, pp. 224–236.

[27] J. B. Li and J. Miller, "Testing the semantics of w3c xml schema," in Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International, vol. 1, 2005, pp. 443–448 Vol. 2.

[28] L. Franzotte and S. R. Vergilio, "Applying mutation testing in xml schemas," in SEKE, K. Zhang, G. Spanoudakis, and G. Visaggio, Eds., 2006, pp. 511–516.

[29] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, 2010, pp. 132–141.

[30] A. Estero Botaro, F. Palomo Lozano, and I. Medina Bulo, "Mutation operators for WS-BPEL 2.0," in ICSSEA 2008: Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications, 2008. [Online]. Available: http://neptuno.uca.es/redmine/wiki/gamera

[31] A. Estero-Botaro, J. Boubeta-Puig, V. Liñeiro-Barea, and I. Medina-Bulo, "Operadores de mutación de cobertura para ws-bpel 2.0," in XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD - SISTEDES 2012), Almería, Spain, 2012.

[32] R. Wang and N. Huang, "Requirement model-based mutation testing for web service," in Next Generation Web Services Practices, 2008. NWESP '08. 4th International Conference on, 2008, pp. 71–76.

[33] padraic, "Mutagenesis," Last access dec, 2013. [Online]. Available: https://github.com/padraic/mutagenesis

[34] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in Proceedings of the 16th International Conference on World Wide Web, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 667–676. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242663

 ISBN: 978-1-61208-346-9