A Weak Mutation Testing Framework for BPMN

Chatri Ngambenchawong and Taratip Suwannasart

***Abstract*—A BPMN is a model that describes a process, showing the sequence of operations and related business information. Execution semantics were introduced in BPMN 2.0 to support the definition of executable processes. In order to test a BPMN model, many researchers focus on test case generation technique. Mutation Testing is a technique to evaluate the quality of test cases by introducing a fault to the original program and the mutated version of the program is called a mutant. A previous research [1] proposed mutation operators for a BPMN model. In order to apply the mutation operator, we need a framework for mutant generation. Thus, this paper proposes a framework for mutant generation based on Weak Mutation testing technique which can generate mutants, deploy mutants on BPMN Engine, and calculate three metrics which are execution time, mutation score, and test effectiveness.**

***Index Terms*—BPMN, BPMN Engine, Business Process Automation, Weak Mutation Testing.**

I. INTRODUCTION

NOWADAYS, many organizations recognize the importance of developing business process. A Business Process Modeling Notation (BPMN) [2] is one of a standardized notation for creating visual models of business or organizational processes in order to provide a standard notation readily understandable by all business stakeholders. BPMN 2.0 was introduced by Object Management Group (OMG) in 2011. This version is more flexible. The execution semantics were introduced to support the definition of executable processes. This makes it possible to test a BPMN model. Many researches on this field have been published. Most of them are related to test case generation [3, 4], but they have not focused on judging the quality of the generated test cases.

Mutation Testing [5] is one of Software Testing techniques that can evaluate the quality of test cases. In Mutation testing, we can introduce a syntax change to the source code. The source code that is mutated with a syntax change is called a mutant. We can use test cases that are used to test the original source code to the test the mutant and check if the test cases are able to find the injected error by comparing between the results of the original program and the mutant. If the results are not the same, we said that the test cases can kill the mutant. This technique is called strong mutation testing. However, the strong mutation has a major drawback because of the expensive computational cost. William E. Howden's [6] proposed his work for reducing the computational cost by introducing a technique called *"Weak Mutation Testing"* that considers between a component of the original program and the mutant.

Previously, Phra Pridsadi and Taratip [1] proposed a Mutation Operator for a BPMN Model. There are twenty-five mutation operators in four categories which are Identifier Mutation Operator, Expression Mutation Operator, Activity Mutation Operator, as well as Exception and Event Mutation Operator. However, the mutation operators should be applied to a BPMN model to verify usability of the operators, and measure quality of test cases.

Thus, we present a weak mutation testing framework by using BPMN Mutation Operator [1] which can generate mutants, automatically deploy mutants on a BPMN Engine, and calculate metrics such as execution time, mutation score, and test case effectiveness.

We organize the rest of this paper as follows. Section II describes related work. Section III explains necessary background knowledge. Section IV presents an analysis of BPMN Mutation Operators. Section V illustrates an example of mutant generation by applying candidate mutation operators. In section VI, we present the proposed Weak Mutation Testing framework for a BPMN model. Finally, conclusion and future work are discussed in section VII.

II. RELATED WORK

Phra Pridsadi and Taratip [1] proposed twenty-five BPMN mutation operators for a BPMN model in four categories as follows.

1) Identifier Mutation Operator – This category includes mutation operators which are assignment operators.
2) Expression Mutation Operator – This category includes mutation operators which control decisions of the model's activities including timing and duration.
3) Activity Mutation Operator – This category includes mutation operators which control the model's activity process as concurrent and sequence process.
4) Exception and Event Mutation Operator – This category includes mutation operators which control the failure and unpredictable activity in the model during model is processing.

Details of the BPMN mutation operators are described in TABLE I. However, this research has not verified usability of the operators, and measure quality of test cases.

Antonia et al. [7] were interested in how to calculate test effectiveness. He revised test effectiveness, proposed by Anna Derezinska [8] , which defined from the relation between the mutation score and a test result. He proposed test

Manuscript received January 06, 2019; revised January 26, 2019.
C. Ngambenchawong and T. Suwannasart are researchers with the Software Engineering Lab, Center of Excellence in Software Engineering, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand, E-mail: Chatri.N@student.chula.ac.th, Taratip.S@chula.ac.th

TABLE I
MUTATION OPERATORS FOR BPMN MODEL

Operator	Description
Identifier Mutation Operators	
IVR	Replaces a variable identifier by another of the same type.
ITR	Replaces the identifier value by another of different type.
Expression Mutation Operators	
EAR	Replaces the arithmetic operator in<conditionExpression>.
ERR	Replaces the relational operator in <conditionExpression>.
ELR	Replaces the logical operator in <conditionExpression>.
ETA	Replaces the value by another number in <timeDuration>.
EDA	Replaces the date by different date in <timeDate>.
ERA	Replaces the repeating round of timer by different value in <timeCycle>.
ECA	Replaces the duration of timer by different value in <timeCycle>.
Activity Mutation Operators	
ASR	Replaces a value of "isSequential" between true and false in <multiInstanceLoopCharacteristics>.
ACR	Replace a value of "loopCardinality" by zero or by half of its initial value or initial value plus one in <multiInstanceLoopCharacteristics >.
AAM	Replaces the arithmetic operator of "completionCondition" in < multiInstanceLoopCharacteristics >.
ARM	Replaces the relational operator of "completionCondition" in <multiInstanceLoopCharacteristics >.
ALM	Replaces the logical operator of "completionCondition" in <multiInstanceLoopCharacteristics>.
ATR	Replaces a value of "testBefore" between true and false in <standardLoopCharacteristics >.
AMR	Replace a value of "loopMaximum" by zero ot by half of its initial value or initial value plus one in <standardLoopCharacteristics>.
AAS	Replaces the arithmetic operator of <loopCondition> in <standardLoopCharacteristics>.
ARS	Replaces the relational operator of <loopCondition> in <standardLoopCharacteristics>.
ALS	Replaces the logical operator of <loopCondition> in <standardLoopCharacteristics>.
AAA	Replaces the arithmetic operator of<completionCondition> in <adHocSubProcess >
ARA	Replaces the relational operator of <completionCondition> in <adHocSubProcess >
ALA	Replaces the logical operator of <completionCondition> in <adHocSubProcess>
AOR	Replaces a value of "ordering" between true and false in <adHocSubProcess>
ARR	Replaces a value of "cancelRemainingInstances" between true and false in <adHocSubProcess>
Exception and Event Mutation Operators	
XBR	Replaces a value of "behavior" by "None", "One", "All", or "Complex" in <multiInstanceLoopCharacteristics>

Note: arithmetic operator (+, -, *, /, mod), relational operator (<, <=, >, >=, ==, !=) and logical operator (and, or)

effectiveness formula by starting from the average number of test cases killing dead mutants *($\overline{K}_d$)* as shown in equation 1.

$$\overline{K}_d = \frac{\sum K_m}{D} \quad (1)$$

K_m is a number of test cases that kill mutants, and D is a number of dead mutants. Test effectiveness *(E)* is calculated by using equation 2.

$$E = MS(P,T)\ x\ \frac{\overline{K}_d}{T} \quad (2)$$

MS(P, T) is the Mutation Score of Program*(P)* under Test Cases*(T)*, $\overline{K}_d$ is an average number of test cases killing dead mutants, and T is a total number of test cases. Therefore, in this paper we use the weak mutation testing for a BPMN model to generate possible mutants for each mutation operator. The generated mutants are tested against test cases that we already have generated to test the original BPMN model. After that three metrics are calculated which are execution time, mutation score (representing quality of test cases), and test effectiveness.

III. Background

A. Business Process Model and Notation (BPMN)

The BPMN [2] is a modeling language that is managed and updated by the Object Management Group (OMG™) for describing functional behaviors of a business process. The main purpose of a BPMN is to create a visual model of business or organizational processes in standardized notation in order to provide readily understandable by all business stakeholders. Currently, the specification is version 2.0, and ISO adopted the BPMN and published it as ISO/IEC 19510:2013 [9] which improved model's capability such as BPMN model interchange between a BPMN designer that is managed by the BPMN Model Interchange Working Group (BPMN MIWG).

The components of a BPMN model are classified into five main groups

1) Flow Objects - Flow objects are controls that are used for describing a business process behavior. There are three Flow Objects: Events - Events can occur at the beginning, the middle, and the end of a process, Activities - Activities are tasks or steps that occur during a process, and Gateways - Gateways are used to control a flow of a process.
2) Data - Data is represented with the four elements: Data Object, Data Input, Data Outputs, and Data Stores.
3) Connecting objects - Connecting objects are used for connecting flow objects together and connecting between flow objects and other information objects. There are four types of connecting objects: Sequence Flows are used to show a sequence of an activity, Message Flows are used to show sequences of messages between senders and receivers, associations are used for connecting data and artifacts and Data Associations are used to show the relation of between a data and an activity.
4) Swimlanes - Swimlanes are used for categorizing objects of the model, consisting of Pools and Lanes.
5) Artifacts - Artifacts are used as additional descriptions of a process. There are two types of Artifacts: Group and Text Annotation.

B. BPMN Engine

A BPMN engine [10] is a tool that helps a process architect to put programming logic into a BPMN model. It can execute a BPMN model without converting the model to source code. In general, a BPMN engine consists of three elements: BPMN Designer, BPMN Model, and Process Engine. The BPMN Designer is a tool that helps business analysts and technical analysts work together by using standard modeling language, and graphical notation. The BPMN Model is a business process that is stored in XML format under *<bpmn:definitions>* tag. The Process Engine is a tool that can create a BPMN model to an executable workflow.

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

C. Mutation Testing

Mutation testing or Strong Mutation [5] is one of the most effective techniques to evaluate the quality of a test suite which has been well-known and studied in many years. This testing technique is a fault-based approach in the unit level of software testing by introducing only one fault in the program. A fault is created by applying a mutation operator to the original program. The source code that is mutated from an original program is called a mutant. However, the major disadvantage is expensive computational cost and time since we can create many mutants from an original program.

The mutation testing process is started from an original program P and corresponding test cases T and described as follows:

1) Program P produces a collection of mutants by using mutation operators to seed a simple fault into P.
2) The original Program P and mutants are executed by using test cases T. Next, the output would be considered if the output of mutant is different from the original program with same input data, the mutant would be killed. Otherwise, the mutant is live.
3) We have to create new test cases to kill Live mutants. In case that the Live mutants cannot be killed, we will call these Live mutants as equivalent mutants.

To measure the test case quality, a tester calculates mutation score: *MS(P,T)* which represents the ratio of number of killed mutants *(M_k)* divided by difference of number of total mutants *(M_t)* and number of equivalent mutants *(M_q)* as shown in equation 3.

$$MS(P,T) = \frac{M_k}{M_t - M_q} \quad (3)$$

D. Weak Mutation Testing

Weak Mutation Testing [6] is another mutation testing that reduces the expensive computational cost and time. William E. Howden's proposed this technique by focusing only a component in a program from giving an example. The program P which C is a simple component of P and mutated version of C produces C'. So, P' is the mutated version of P containing C'. There are five types of program components which William E. Howden defined as follows:1) Variable Reference 2) Variable Assignment 3) Arithmetic Expression 4) Relation Expression and 5) Boolean Expression. Nevertheless, there was no clear definition of a program component.

J.Offutt [11] proposed his work that given a clear definition of component (C) which are categorized into four types as follows:

1) EX-WEAK/I (Expression-WEAK/1) The first type of weak mutation testing is comparing the state after the first execution of the innermost expression between an original program and a mutant. There is the expression of the original program Z = (A+B) * (C+D) and a mutant which is Z = (A+B) * (C-D). The result of the expression (C+D) and (C-D) must be compared between the original program and the mutant.
2) ST-WEAK/I (Statement-WEAK/1) This mutation type compares the state after the first execution between the statement of the original program and the mutated statement. The statement of the original program is Z = (A+B) * (C+D) and the mutated version is Z = (A+B) * (C-D), are compared after the first execution.
3) BB-WEAK/1 (Basic-Block-WEAK/1) This type of weak mutation considers a basic block which is the maximal sequence of instructions with one entry and one exit. The result of for the original program of loop i is i <= 50 and the result of the mutated version is i < 50.
4) BB-WEAK/N (Basic-Block-WEAK/N) The type of weak mutation is the extended type of BB-WEAK/1. This type considers N times execution of a basic block. Since, a mutant in a basic block component sometimes cannot be killed at the first loop of execution. This technique compares each loop execution between the original program and the mutant. An example of the basic block component for BPMN is <multiinstance LoopCharacteristics>, and <testbefore>.

E. Mutation Operator

In mutation testing, we use mutation operators to create a set of mutants. Thus, in mutation operators have been proposed for supporting many programming languages. Mutation operators are categorized into four types as. *Procedural Programming Language* such as C [12], and Fortran [13]. As time passes *Object-Oriented* was introduced and adapted to mutation testing Technique such as Java [14], Python [15]. *Set-Oriented Language* is for database management like SQL [16] and the last type *Process Modeling Languages* is for WS-BPEL [17] and BPMN [1]. Mutation operators are designed for each programming language, but some mutation operators are designed based on the same concept such as arithmetic expression.

IV. ANALYZE BPMN MUTATION OPERATOR

This section describes BPMN mutation operators proposed in [1] in order to adapt in weak mutation technique

1) EX-WEAK/1: This mutation type considers only an expression so the mutation operator in categories Identifier Mutation Operators mostly in Expression Mutation Operators and Activity Mutation Operators, except ERA, ECA, ASR, ACR ATR, AMR, AOR and ARR can be considered in this expression analysis.
2) ST-WEAK/1: This second type of weak mutation testing covers all mutation operators of EX-WEAK/1 and XBR which in Exception and Event Mutation Operators, for instance ARS: <completionCondition> under adHocProcess Notation which check the completeness of AdHoc Process like if-else statement and XBR:<multiInstanceLoopCharacteristics>attribute which invoke call back after the instance of task completed like switch-case statement.
3) BB-WEAK/1 and BB-WEAK/N These types consider a block component of BPMN and cover mutation operator which cannot use EX-WEAK/1 and ST-WEAK/1 to kill mutants. BB-WEAK/1 used for the first-time execution and BB-WEAK/N is extended from BB-WEAK/1 that handle mutant in basic block which alive in the first round but can kill after n round of executions.

As the result, TABLE II demonstrates between the

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

availability (✓) of mutation operators and each type of weak mutation that can be used with BPMN Model

TABLE II
WEAK MUTATION OPERATORS FOR BPMN MODEL

Operator	Weak Mutation for BPMN Model			
	EX-WEAK/1	ST-WEAK/1	BB-WEAK/1	BB-WEAK/N
Identifier Mutation Operators				
IVR	✓	✓	✓	✓
ITR	✓	✓	✓	✓
Expression Mutation Operators				
EAR	✓	✓	✓	✓
ERR	✓	✓	✓	✓
ELR	✓	✓	✓	✓
ETA	✓	✓	✓	✓
EDA	✓	✓	✓	✓
ERA	-	-	✓	✓
ECA	-	-	✓	✓
Activity Mutation Operators				
ASR	-	-	✓	✓
ACR	-	-	✓	✓
AAM	✓	✓	✓	✓
ARM	✓	✓	✓	✓
ALM	✓	✓	✓	✓
ATR	-	-	✓	✓
AMR	-	-	✓	✓
AAS	✓	✓	✓	✓
ARS	✓	✓	✓	✓
ALS	✓	✓	✓	✓
AAA	✓	✓	✓	✓
ARA	✓	✓	✓	✓
ALA	✓	✓	✓	✓
AOR	-	-	✓	✓
ARR	-	-	✓	✓
Exception and Event Mutation Operators				
XBR	-	✓	✓	✓

V. GENERATE MUTANT

This section is an example of mutant generation by applying candidate mutation operators of each category for weak mutation testing.

Fig. 1 shows an original BPMN model in XML format, and Fig. 2 to Fig. 6 are mutated versions by using ERR mutation operator which replaces a relational operator in the expression INPUT_A + INPUT_B >=5. In this case, the original BPMN model will result in five mutants. The >= is replaced by other types of relational operator including >, <=, <, ==, and !=.

Original BPMN Model

```
<bpmn:processid="Process_1" isExecutable="true">
  ...
  <bpmn:sequenceFlow id="SeqFlow1" sourceRef="Task01"
      targetRef="ExclusiveGateway1">
   <bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
         <![CDATA[${INPUT_A + INPUT_B >= 5]]>
   </bpmn:conditionExpression>
 </bpmn:sequenceFlow>
  ...
</bpmn:process>
```

Fig. 1. Original BPMN Model

Possible Mutants BPMN Model (Change from >= to >)

```
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
      <![CDATA[${INPUT_A + INPUT_B>5}]]>
</bpmn:conditionExpression>
```

Fig. 2. A Mutant after applying ERR Operator (Change from >= to >)

Possible Mutants BPMN Model (Change from >= to <=)

```
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
      <![CDATA[${INPUT_A + INPUT_B<=5}]]>
</bpmn:conditionExpression>
```

Fig. 3. A Mutant after applying ERR Operator (Change from >= to <=)

Possible Mutants BPMN Model (Change from >= to <)

```
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
      <![CDATA[${INPUT_A + INPUT_B<5}]]>
</bpmn:conditionExpression>
```

Fig. 4. A Mutant after applying ERR Operator (Change from >= to <)

Possible Mutants BPMN Model (Change from >= to ==)

```
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
      <![CDATA[${INPUT_A + INPUT_B==5}]]>
</bpmn:conditionExpression>
```

Fig. 5. A Mutant after applying ERR Operator (Change from >= to ==)

Possible Mutants BPMN Model (Change from >= to !=)

```
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
      <![CDATA[${INPUT_A + INPUT_B!=5}]]>
</bpmn:conditionExpression>
```

Fig. 6. A Mutant after applying ERR Operator (Change from >= to !=)

VI. BPMN WEAK MUTATION TESTING FRAMEWORK

Phra Pridsadi and Taratip [1] proposed mutation operators for a BPMN Model and they generate only limited numbers of mutants using strong mutation testing and they test the mutants with test cases manually.

We have proposed a weak mutation testing framework for a BPMN Model by applying weak mutation testing techniques to generate a set of mutants and automatically execute mutated BPMN models on a BPMN engine. The structure of our proposed framework shown in Fig. 7. The components and main functions of the framework are described as follows:

1) Mutant Analyzer & Generator: At this stage there are sub-steps below.
   1.1) BPMN Validator: Testers upload a BPMN Model which consists of tags <bpmn:process> and <bpmndi:BPMNDiagram>. Then, the BPMN model is checked by using BPMN XSD schema if it is a BPMN Model or not.
   1.2) Mutant Generator: This step a set of mutants is generated based on BPMN mutation operators [1] and the mutants are stored into the mutant database.
2) Test Execution: At this stage there are sub-steps below.
   2.1) Test Controller: The original BPMN model and Mutants are loaded from the database. Firstly, the Test Controller deploys the original BPMN model to the BPMN Engine server via REST API. The Test Controller executes the original BPMN model with test cases and save the results of original BPMN model. Secondly, the Test Controller deploys each mutated BPMN model to the BPMN Engine server, executes each mutant with the same test cases used with the original BPMN model and save the results of mutants.
   2.2) Result Comparator: This step retrieves test results from the original BPMN model and mutants and compares these results and record to see if each mutant is killed or live in the test result database.
3) Metric Calculation: At this stage, the results from Test

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Execution are retrieved to calculate metrics.

3.1) Mutation Score Calculation: This step, test results are retrieved from the database and are measured sufficient of test cases by the ratio of number of dead mutants divided by the difference between total mutants and the number of equivalent mutants as described in section 3.

3.2) Test Effectiveness Calculation: This step retrieves mutation score and test results to calculate test effectiveness by using equation from Antonia Estero-Botaro's experiment showing the relation between mutation score and the ratio of the average of number of the test cases that kill mutants divided by total number of test cases as discussed in section 2.

3.3) Report Generator: This step creates the summary of the test results that include total number of mutants, killed mutants, live mutants, mutation score, test effectiveness, and execution time.

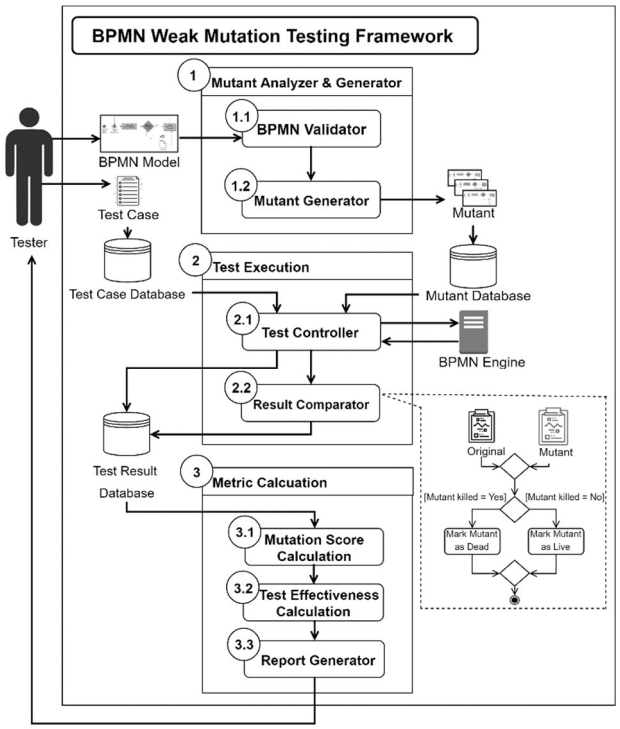

Fig. 7. BPMN Weak Mutation Testing Framework

VII. Conclusion

This paper proposes a weak mutation testing framework for a BPMN model that provides a result from comprehensive analysis that guides the tester for using all mutation operators in [1] by each level of weak mutation testing technique as shown in section IV with an example of a mutation operator in section V. The future work is to implement a tool that fully supports with all mutation operators with an open source BPMN Engine.

References

[1] P Tadeesom and T. Suwannasart. *Mutation Operators in BPMN Model*. in *ICIDE 2017*. 2017.

[2] OMG. *Business Process Model and Notation V.2.0.2 [Online]*. 2014; Available from: https://www.omg.org/spec/BPMN/2.0.2 [2019,Jan 15]

[3] P. Yotyawilai and T. Suwannasart, *Design of a tool forgenerating test cases from BPMN*, in *Proceedings of 2014 International Conference on Data and Software Engineering,ICODSE 2014*. 2014. p. 1-6.

[4] C. Nonchot and T. Suwannasart, *A tool for generating test casefrom BPMN diagram with a BPEL diagram*, in *IMECS2016*. 2016.

[5] DeMillo, R.A., R.J. Lipton, and F.G. Sayward, *Hints on test data selection: Help for the practicing programmer*, 1978: p. 34-41.

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

[7] Estero-Botaro, A., F. Palomo-Lozano, and I. Medina-Bulo. *Quantitative evaluation of mutation operators for WS-BPEL compositions.(ICSTW), 2010 Third International Conference on*. 2010. IEEE.

[8] Derezinska, A. *Quality Assessment of Mutation Operators Dedicated for C# Programs*. in *2006 Sixth International Conference on Quality Software (QSIC'06)*. 2006.

[9] ISO/IEC, *ISO/IEC 19510:2013 – Information technology - Object Management Group*

[10] Silver, B., *BPMN Method and Style, with BPMN Implementer's Guide: A structured approach for business process modeling and implementation using BPMN 2.0*. 2011: Cody-Cassidy Press Aptos.

[11] Offutt, A.J. and S.D. Lee, *An empirical evaluation of weak mutation.* IEEE Transactions on Software Engineering, 1994: p. 337-344.

[12] Delamaro M. E., M.J.C. *Proteum - A Tool for the Assessment of Test Adequacy for C Programs*. in *In: Proc. of the Conf. on Performability in Computing Systems (PCS 96)*. 1996.

[13] King, K.N. and A.J. Offutt, *A Fortran language system for mutation-based software testing.* Softw. Pract. Exper., 1991: p. 685-718.

[14] Ma, Y.-S., J. Offutt, and Y.R. Kwon, *MuJava: an automated class mutation system: Research Articles.* Softw. Test. Verif. Reliab., 2005: p. 97-133.

[15] Halas, K. *MutPy*. 2017; Available : https://github.com/mutpy/mutpy.

[16] Tuya, J., M.J. Suarez-Cabal, and C.d.l. Riva, *Mutating database queries.* Inf. Softw. Technol., 2007: p. 398-417.

[17] A. Estero-Botaro, F.P.-L., and I. Medina-Bulo. *Mutation operators for WS-BPEL 2.0.* in *ICSSEA 2008: 21th International Conference on Software & Systems Engineering and their Applications*. 2008. . Paris, France.