Mutation Testing in Event Programming Language

L. Gutiérrez-Madroñal, J.J Domínguez-Jiménez, and I. Medina-Bulo

UCASE Software Engineering Research Group,
University of Cadiz, Spain
{lorena.gutierrez,juanjose.dominguez,inmaculada.medina,}@uca.es
https://ucase.uca.es/

Abstract. Event processing queries are intended to process continuous event streams. These queries are partially similar to traditional SQL queries, but provide the facilities to express rich features (e.g., pattern expression, sliding window of length and time). An error while implementing a query may result in abnormal program behaviors and lost business opportunities. Mutation testing has been found to be effective to assess test suites quality and generating new test cases. In this work, we propose mutation-based testing of the Event Processing Language (EPL) 4.9.0 (a domain-specific language for processing events), mutation operators which modify different features of EPL queries and a new mutation analysis testing tool for EPL (MuEPL). Moreover, to evaluate MuEPL, we have applied to EPL programs.

Keywords: Mutation Testing, Event Processing Queries, Event Processing Language

1 Introduction

Mutation testing 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.

All mutants 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. When a mutant can be told apart from the original program, the mutant has been *killed* by the test suite.

As for real-time, mutation testing has been applied to many traditional programming languages that have been growing and now they can be applied to real-time systems: Java [1], C [2], Ada [3]. The present study uses the mutation testing as a way to test a non traditional programming language. We apply mutation testing to a programming language created to be used in real-time systems, more particularly in Complex Event Processing (CEP) systems. To achieve this goal its mutation operators must be defined, a mutation tool for the CEP system must be developed and finally they have to be tested (operators and mutation tool) using examples. Moreover, the mutation tool must satisfy certain real-time system peculiarities that hinder its implementation.

The structure of the rest of the paper is as follows: An introduction about mutation testing and EPL language are respectively shown in Section 2 and Section 3. Section 4 defines and clasifies the EPL mutation operators. Section 5 presents MuEPL, mutation testing tool for EPL. Section 6 describes the examples where MuEPL will be applied and evaluates the obtained results. Section 7 presents conclusions and future work.

2 Mutation Testing

Mutation testing is a fault-based testing technique providing a test criterion: the *mutation score*. This criterion can be used to measure the effectiveness of a test suite in terms of its ability to detect faults. Mutation testing generates mutants from the program under test by applying mutation operators to it. These mutation operators introduce slight syntactical changes into the program that should be detected by a high-quality test suite. Each mutation operator represents "typical" programming errors, that the developer could make. Thus, if a program contains the instruction `a > 2000` and we apply the relational mutation operator (which replaces a relational operator with another), the resulting mutant could contain the instruction `a >= 2000` instead, for example. If a test case is able to distinguish between the original program and the mutant, i. e. their outputs are different, 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. 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.

3 Event Processing Language

Esper [4] is an open-source Java-based software product for Complex Event Processing (CEP) and Event Stream Processing (ESP), that analyzes series of events for deriving conclusions from them. It offers a domain-specific language for processing events called Event Processing Language (EPL). EPL is a declarative programming language for analyzing time-based event data and detecting situations as they occur.

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. An EPL statement starts executing continuously during runtime. While the execution is taking place, EPL queries will be triggered if the application receives pre-defined or timer triggering events.

EPL 4.9.0 query example

```
select A as temp1, B as temp2 from pattern
     [every temp1.temperature > 400 -> temp2.temperature > 400]
```

EPL 4.9.0 is one of the latest versions of this language. In the above example a "Central“ needs to measure the temperature of its systems, its temperature gauges take a reading of the core temperature every second and send the data to a central monitoring system. The EPL query of the figure, warns us if we have 2 consecutive temperatures above a certain threshold (400). This is a situation where it is needed to react quickly to emerging patterns in a stream of data events.

4 EPL mutation operators

In a previous work [5], was defined a list of mutation operators, some of them have been re-defined, and a new one has been defined. After checking several EPL queries that where used in real examples, the number of operators turn into 17. Due to the decreased number of mutation operators, the operators have been re-classified in four categories depending on which kind of EPL query element the are related to. These are identified by uppercase letters: P (*Pattern expression operators*), W (*Windows operators*), R (*Replacement operators*)[1] and I (*SQL Injection attack operators*).

Operators are uniquely identified by three uppercase letters: the first one is the category identifier, and the last two letters indicate the operator within the category. The Table 1 lists their names and provides a short descriptions of each of them. Some operators are specific for the EPL language, they appear marked with ☆.

5 MuEPL architecture

MuEPL lets us apply mutation analysis to EPL 4.9.0 queries, and it is the first tool with this capability. Figure 1 shows the relations between these components.

The capturer obtains the queries while the original program is under execution. Next the analyser receives the original queries and generates the information about which mutation operators can be applied and in which locations. This information is delivered to the mutant generator, which generates every possible mutant. Then, the execution engine runs the original and its mutant against the set of test cases and compares their behaviours to determine whether the mutants have been killed or stay alive.

[1] The *Replacement operators* may appear in the pattern of the query, but we considered extend their definition and apply them not only to the pattern but also to the rest of the query.

Due to the data nature which MuEPL deals with (events in real-time), we must ensure that all the programs receive the same events. This is because we want study the behaviour of the original program and the generated mutants under the same conditions. To achieve this, the execution system includes a mechanism that can synchronise the execution threads (original and mutants). This mechanism builds a "barrier" where all threads must wait, until all threads reach it, before any of the threads can continue.

Operator	Description
	Pattern Expression Mutation
PNE	☆ Removes the `not` keyword of the negated conditional expressions in the pattern
PTI	☆ Increases the timer value by one unit in the pattern observer (`timer:at`, `timer:interval`)
PTD	☆ Decreases the timer value by one unit in the pattern observer (`timer:at`, `timer:interval`)
	Replacement Mutation
RLO	Replaces a logical operator (`and`, `or`) by another of the same kind
RTU	Replaces one time unit (`milliseconds`, `seconds`, `minutes`, `hours`, `days`) by another of the same kind
RAF	Replaces an aggregate function (`max`, `min`, `avg`, `sum`, `count`, `median`, `stddev`, `avedev`) by another of the same kind. The `distinct` keyword could be also added
RAO	Replaces an arithmetic operator (`+`, `-`, `*`, `/`, `%`) by another of the same kind
RRO	Replaces a relational operator (`=`, `<>`, `<`, `>`, `<=`, `>=`) by another of the same kind
RNO	Replaces a number `e` by `e + 1` and `e - 1`
	Windows Mutation
WLI	☆ Increases the data window length by one
WLD	☆ Decreases the data window length by one
WTI	☆ Increases the time window by one
WTD	☆ Decreases the time window by one
WBL	☆ Turn a batch window length into a ordinary window length
WBT	☆ Turn a batch window time into a ordinary window time
	Injection Attack Mutation
IRC	Removes "where" condition from a query
INC	Negates the condition expression of a query

Table 1. EPL mutation operators

6 Applying Mutation Testing to EPL programs

All the experiments in this section have been run in a Intel Core i7 machine with 4GB RAM and a 2.00GHz x 4, running Ubuntu 14.04 LTS.

We have used two programs from EsperTech website [4] to experimentally evaluate the mutation operators for EPL and MuEPL. The chosen programs are: *Self-Service Terminal* and *Transaction 3-Event Challenge*.

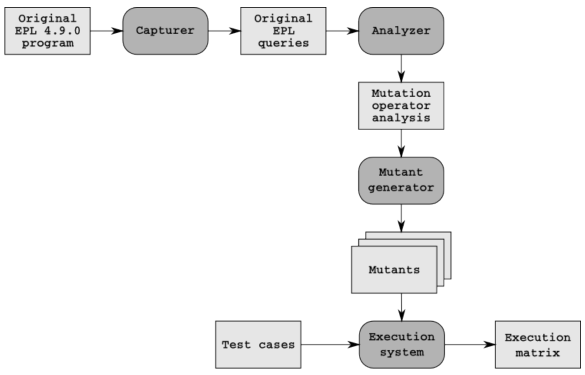

Fig. 1. MuEPL architecture

For each program it was needed to include a seed as input (and modify the random code which receive it). This is another undertaken measure to study the behaviour of the programs under the same circumtances.

Self-Service Terminal is about a J2EE-based self-service terminal managing system in an airport that gets a lot of events from connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as "paper low" or "terminal out of order". Other events observe activity as customers use a terminal to check in and print boarding tickets.
This program executes 6 queries, and for these queries 16 of the 17 mutation operators are applicable and we have used a test suite of 5 test cases. The non-applied mutation operator is: RAF, and are generated 79 mutants. The exhaustive execution of 80 threads (mutants + original) in parallel against each of the 5 test cases takes more than 3'15 hours in total.

Test cases			Outputs		
Iterations	Sec.	Seed	Killed	Live	Fail
100	20	10	72	7	0
1000	50	25	76	1	2
2500	100	25	78	0	1
10000	250	15	76	1	2
25000	100	0	78	0	1

Table 2. Self-Service Terminal outputs

The live mutants for the first test case are from the operators: PTD (1), RAO (2), RRO (1), RTU (3). In the second test case is from RRO, and for the fourth test case is from RTU. It is worth remarkable how the outputs of different mutants, depending on the number of iterations and the sleep seconds, are failed.

Transaction 3-Event Challenge tracks three components of a transaction. The example uses at least three components, since some engines have different performance or coding for only two events per transaction. Each component comes to the engine as an event that are generated by the included "event generator". The transaction events come completely out of order; a bucket (with a modifiable size) is filled, and when it is full, it is shuffled. The larger the bucket size, the more events potentially come in between two events in a given transaction and so, the more the performance characteristics like buffers, hashes/indexes and other structures are put to the test as the bucket size increases.
This program executes 5 queries, and for these queries 9 of the 17 mutation operators are applicable. The applied mutation operators are: RLO, WTI, WTD, RTU, WBT, IRC, INC, RAF, RAO and RRO, and are generated 435 mutants. The machine was not able to run 436 threads (original + mutants) in parallel against the simplest test cases: bucket size "tiniest" (20), number of transactions 10, seed 10.

7 Conclusions and future work

The present study propose the mutation testing as a way to test a programming language developed to be used in real-time systems, EPL. In spite of being a query language, the execution of all the mutants have a high cost. This is because of the main drawbacks of mutation testing: 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. [6], the number of mutants has quadratic complexity in program size.

Several techniques have been described to solve this problem; one of them is *mutant reduction technique*, which process only a subset of all the mutants. As a future work, MuEPL will be adapted to apply one of the mutant reduction techniques: *Evolutionary Mutation Testing* (EMT) [7], that finds mutants that help derive new test cases that improve the quality of the initial test suite. Applying EMT, we will obtain *strong mutants*: suviving mutants (which have not been killed by the test suite) and difficult to kill mutants (which have been killed by one and only one test case that kills no other mutant). According to our study, the mutations can be hard to find in quick executions (less number of iterations), so we have to focus our experiments on this kind of test cases.

References

1. Ma, Y., Offutt, A.J., Kwon, Y.: MuJava: a mutation system for Java. Proceedings of the 28th international conference on Software engineering, ACM (2006), pp. 827–830
2. Agrawal, H., DeMillo, R., Hathaway, B., Hsu, W., Hsu, W., Krauser, E., Martin, R., Mathur, A. Spafford, E.: Design of mutant operators for the C programming language (1999)

3. Offutt, J., Voas, J., Payne, J.: Mutation operators for ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University (1996)
4. EsperTech website, `http://www.espertech.com/esper/index.php`
5. Gutiérrez-Madroñal, L. and Shahriar, H. and Zulkernine, M. and Dominguez-Jimenez, J.J. and Medina-Bulo, I.: Mutation Testing of Event Processing Queries. Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, (2012), pp. 21-30, ISBN:1071-9458
6. Offutt, A.J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. Software Engineering, 1993. Proceedings., 15th International Conference on, pp. 100-107, ISSN 0270-5257
7. Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I.: Evolutionary mutation testing. Information and Software Technology, Elsevier (2011), v. 53, n.10 pp. 1108–1123