Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

ARTICLE TYPE

Evolutionary mutation testing for IoT with recorded and generated events†

Lorena Gutiérrez-Madroñal*[1] | Antonio García-Domínguez[2] | Inmaculada Medina-Bulo[1]

[1]UCASE research group, University of Cadiz, Cadiz, Spain
[2]SEA research group, Aston University, Birmingham, United Kingdom

Correspondence
*Lorena Gutiérrez-Madroñal, Escuela Superior de Ingeniería, C/ Avd. Universidad de Cádiz 10, Puerto Real. Email: lorena.gutierrez@uca.es

Abstract

Mutation testing is a testing technique that has been applied successfully to several programming languages. Despite its benefits for software testing, the high computational cost of mutation testing has kept it from being widely used. Several refinements have been proposed to reduce its cost by reducing the number of generated mutants: one of those is Evolutionary Mutation Testing (*EMT*). *EMT* aims at generating a reduced set of mutants with an evolutionary algorithm, which searches for potentially equivalent and difficult to kill mutants that help improve the test suite. *EMT* has been evaluated in two contexts so far: web service compositions and object-oriented C++ programs. This study explores its performance when applied to Event Processing Language (EPL) queries of various domains. The study also considers the impact of the test data, since a lack of events or the need to have specific values in them can hinder testing. The effectiveness of *EMT* with the original test data generators and the new *IoT-TEG* (IoT Test Event Generator) tool is compared in multiple case studies.

KEYWORDS:
Evolutionary Mutation Testing; Guided Evolutionary Mutation Testing; Internet of Things; IoT-TEG; Event Processing Language; Genetic Algorithm; CEP

1 | INTRODUCTION

The *Internet of Things* (IoT) is becoming increasingly important for government, academy and industry all over the world. Haller et al. [HKS08] defined the IoT as a world where physical objects are integrated into the network, and where these objects can become active participants in business processes. One of the main drawbacks of IoT systems is the amount of information they have to handle. The information in the IoT world is especially critical in areas such as e-commerce, fraud detection, sensor networks, market data, etcetera. This huge volume of information arrives as events that need to be monitored and processed in real time in order to make correct decisions.

The events need to be processed and monitored in order to make correct decisions, a wrong decision can trigger a mistaken action that could be very critical depending on the area: medicine, environment, retail and so on. Given that processing the data is crucial, testing the IoT systems that will manage that information is required.

In the literature, depending on the author and even the company, the challenges for testing IoT applications are categorised in different ways. We have identified the following challenges:

- Environment: IoT applications work in very dynamic environments that change in unpredictable ways.
- Complexity: IoT applications may have multiple real-time scenarios and their use cases require the coordination of many entities.

†Paper partially funded by The Ministry of Economy and Competitiveness (Spain) and the FEDER Fund, under the National Program for Research, Development and Innovation, Societal Challenges Oriented, Project DArDOS TIN2015-65845-C3-3-R, and the Excellence Network SEBASENet (TIN2015-71841-REDT).

- Scalability: creating a test environment to assess functionality that can scale to many devices reliably is difficult.

- Availability: the majority of software to test IoT is proprietary, costly, and not publicly available.

IoT testing is concerned with security and performance; if IoT systems are tested before being launch, we will be able to detect errors and prevent possible failures. The drawback is that testing real-time systems, as was described by Heath [Hea91], is one of the most complex and time-consuming activities. Its computation complexity is a specially important issue because it typically consumes 50% of the overall development effort and budget given that testing embedded systems is more difficult than testing conventional systems. The real-time requirements like timeliness, simultaneity, and predictability make the tests costly.

Mutation testing is a widely applied technique that consists of evaluating how well the test suite can tell apart the original program from one seeded with a common defect (a "mutant"). Moreover, this technique models common mistakes that programmers make when implementing. Applying this technique we can simulate different errors that programmers make and prevent the applications from making wrong decisions. For instance, this is extremely important when receiving information from sensors. A sensor reporting a high temperature value could mean that there is a fire but our system will not trigger the alarm if the rule or pattern to identify it has an error; in [VM17] how to detect a fire through an IoT system is presented. In the same way, if our system does not interpret correctly the values sent by a pacemaker and other sensors because the pattern or rule which define a heart-attack is not well defined, the person could be no salved; a IoT system is proposed in [LHZ17] to perceive a heart-attack.

The multiple scenarios that the mutation testing will generate, represent one of the previous challenges: complexity. It has to be taken into account the main drawback of mutation testing [OU01]: the high computational cost involved in the execution of the large number of mutants produced for some programs against their test suites. Several well-studied cost reduction techniques to avoid biased results can be found. *Evolutionary Mutation Testing* (*EMT*) [DEGM11] consists of generating a reduced set of mutants by means of an evolutionary algorithm, which searches for potentially equivalent and difficult to kill mutants to help improve the test suite. *EMT* was effective with WS-BPEL web service compositions [DEGM11] and object-oriented C++ programs [DPMBS+17]. Using *EMT* will help us to solve the complexity challenge.

The relevant contributions of this paper are as follows:

- **An application to apply the *EMT* technique to IoT systems**. The IoT systems use specific languages to process the events that they have to monitor: Event Processing Languages (EPL). We have chosen an open source EPL created by EsperTech [Espb] for their Esper engine (Esper EPL from now on). A stream oriented EPL which is the most used programming language if we talk about event processing.

- In order to apply *EMT* to Esper EPL, **a bridge has been developed to connect the MuEPL [GM17] mutation system and *GAmera* [DJEBGDMB09]**. MuEPL is a tool which has been developed to apply mutation testing to Esper EPL, and *GAmera* is a tool with an evolutionary algorithm which uses *EMT*. In this paper we describe how this connection has to be done.

- **An improvement of the *EMT* technique**. The modification was done based on the idea of a guided mutation approach.

- **A comparison between IoT-TEG generated events and the original ones**. IoT-TEG is a system which automatically generates events for testing. This paper compares if the generated IoT-TEG events can replace the test data that previously existed for the multiple EPL queries under test.

- **An evaluation of the fitness landscapes produced by EMT**. Each combination of program and test suite generates its own search space, and the performance of EMT will depend on how individuals are distributed over it. This paper compares the fitness landscapes produced for four different scenarios and suggests ways to improve EMT based on them.

The remainder of this paper is organised as follows. In section 2 the background of our investigation is detailed. Section 3 describes how mutation testing and *EMT* has to be applied to IoT systems as well as different adaptations that should be taken into account. Section 4 states our research questions and experiment design, and then answers the questions through the analysis of the results obtained. In Section 5, we review the related work. The last section presents the conclusions and future lines of research.

2 | BACKGROUND

2.1 | Internet of Things

The IoT brings connection and interaction of objects and systems providing information in the form of events. This amount of events has to be filtered and processed to make correct decisions in urgent situations such as reacting to emergencies on patient's health conditions. As a consequence of the huge sum of events that the systems have to handle, new ways of obtaining, processing, and transmitting information have to be put into action in order to reduce the computational cost. Continuously monitoring ongoing activities and responding with a minimal latency is one of the major challenges that IoT systems have to deal with. Complex Event Processing (CEP) was introduced by David Luckham [Luc02] and tries to address this challenge and to solve the decision making problem. However, after working with this solution, in the first workshop on Event Processing [Luc06], Luckham identified his “Top Worries About Future Directions”, which where among others includes:

“The usage of higher level languages for events patterns and rules.”

An event pattern was defined by Luckham as a template which contains event templates, relational operators and variables. These event patterns match sets of events by replacing variables with values. The event templates included by event patterns are event forms, and they may include parameters. These event templates match single events by replacing their variables with values. And finally, a rule in event processing, is a method for processing events.

Agreeing with Luckham, Schiefer et al. [SRRS07] state that the specification of a standardised language for the definition of event patterns and event rules (also known as an Event Processing Language or EPL), is a key issue. Nowadays there are several EPLs with different characteristics and purposes, but they have one thing in common: all of them work with events.

In order to know the context where these events are processed and these patterns run, the IoT basic characteristics have to be indicated [VFG+13]:

- **Interconnectivity**: Anything can be interconnected to the network with the global information and communication infrastructure.
- **Things-related services**: The IoT is capable of providing thing-related services within the constraints of things because, both the technology in physical world and information world change.
- **Heterogeneity**: The IoT devices are based on different hardware platforms and networks, and they can interact with each other through different networks.
- **Dynamic changes**: The state as well as the context of devices change dynamically. Moreover, the number of devices changes dynamically.
- **Enormous scale**: The number of devices that need to be managed and that communicate with each other is huge. Even more critical will be the management of the generated data and their interpretation.
- **Safety**: The creators and recipients of the IoT must be designed for safety. The safety has to be not only of our personal data but also our physical well-being.

2.2 | Esper Event Processing Language

Etzion and Niblett [EN10] defined *event processing agents* as software modules that process events. These agents are specified using an event processing language (EPL), of which there are several styles in use:

- Rule-oriented languages:
  - Production rules (*if condition then action*): when the condition is satisfied, the action is performed.
  - Active rules (*event-condition-action*): when an event occurs, the conditions are evaluated and, if they are satisfied, an action is triggered.
  - Logic rules, based on logical assertions and a deductive database.
- Imperative programming languages define the operators which will be applied to the events. Each operator is a transformation in the event.
- Stream-oriented languages are inspired by relational algebra, and often SQL.

In this work, we will use the Esper EPL [Espb], which is stream-oriented and its syntax is very similar to SQL. We chose Esper EPL because it is conceptually an extension of SQL (thus easier to adopt), it can be embedded into Java applications, and it is open source. According to EsperTech, the Esper CEP engine can process around 500,000 events per second on a workstation, and between 70,000 and 200,000 on a laptop.

Esper EPL is inspired on SQL, but it works on continuous streams of events rather than tables. Instead of rows in a table, we have events in a stream. An EPL statement is continuously executed during the runtime. While the execution is taking place, EPL queries are triggered if the application receives pre-defined or timer-triggering events.

Example 1: Example of an Esper EPL query

```
select * from TemperatureEvent
match_recognize (
  measures A as temp1, B as temp2
  pattern (A B)
  define
    A as A.temperature > 400,
    B as B.temperature > 400)
```

Suppose we have a system that needs to keep temperatures under control, and that it is producing measurements periodically. The system contains several thermometers which send the data to a central monitoring system; the central monitoring system process the `TemperatureEvent` events according to the query in Example 1. Example 1 [Mil] shows a query which would throw a warning if we had 2 consecutive measurements above 400. This is one situation where we would need to quickly respond to emerging patterns in a stream of data events. The query contains a pattern, `pattern (A B)`, which will be triggered when an event A and an event B arrive, being A followed by B. Moreover, the temperature property of these two events has to be greater than 400.

2.3 | Mutation Testing

Mutation testing is a well-known fault-based technique that has been used to evaluate and improve the quality of the test suites in a system that is being tested. This technique 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 developers make.

Mutation testing has been applied to different domains as new technologies have appeared. Given the popularity of real-time systems, the features of traditional programming languages have been extended to be used with real-time systems. Even though traditional programming languages, such as Java, C and Ada, have evolved, previous studies that applied mutation testing [MOK06, ADH+89, OVP96] did not consider this progress. In addition, a prior survey of the overall state of mutation testing [JH11b] showed many mutation systems, but none were related to Esper EPL.

Ahmed et al. [AZY10] commented that the research on mutation testing can be classified into several phases: definition of a set of mutation operators; mutation system development; empirical studies of the technique and use of cost-reduction techniques. In our previous works [GMSZ+12, GMDJMB15], we showed initial versions of the Esper EPL mutation operators based on EPL syntax. However, after an analysis of real projects housed on the github platform[1], a more realistic mutation operator classification was presented in [GM17].

In order to apply mutation testing to Esper EPL, the MuEPL mutation system has been developed [GM17]. Given that mutation testing is expensive, parallel execution was included in this system. Although the reduction of the computational cost was significant, we considered that we can also reduce the expenses of mutation testing even more using well-studied cost reduction techniques to avoid biased results. *Evolutionary Mutation Testing* (*EMT*) [DEGM11] consists of generating a reduced set of mutants by means of an evolutionary algorithm, which searches for potentially equivalent and difficult to kill mutants to help improve the test suite. *EMT* was effective with WS-BPEL web service compositions [DEGM11] and object-oriented C++ programs [DPMBS+17]. This paper evaluates the effectiveness of *EMT* when applied to several Esper EPL queries on events produced by IoT systems.

2.4 | Evolutionary Mutation Testing

EMT [DEGM11] uses an evolutionary algorithm (implemented in the *GAmera* [DJEBGDMB09] tool) to guide mutant selection towards a subset that inspires the design of new test cases. The algorithm favours *strong mutants*. There are two kinds:

- *Potentially equivalent* mutants survive the existing test suite. These need to be inspected to see if they are really equivalent to the original program. If they are not, they will lead to a new test case: either because the mutation was not reached, or because the test suite could not tell the mutant apart.

[1]www.github.com

Operator	Location	Attribute

Figure 1 Mutant encoding

Example 2: Original query and mutated query (first appearance of $>$ is changed by $<$)

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

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

Table 1 Predefined positions of operators and their attributes

Operators	Attributes
Arithmetic op. replacement (AOR)	+, *, -, /
Relational op. replacement (ROR)	<, >, =, !=, <=, >=

- *Difficult to kill* mutants are killed by a specific test case, which kills no other. It models a subtle, hard-to-find bug, and it is useful as a place where the algorithm should focus.

2.4.1 | Individual encoding

The generated mutants in the original program are considered individuals for the evolutionary algorithm in EMT. MuEPL encodes each mutant with three fields (see Figure 1):

1. Operator: position of the mutation operator within the list (1 is the first operator).
2. Location: position within the mutants of that operator (1 is the first mutant of that operator).
3. Attribute: identifier of the variant inserted in a location (1 is the first variant).

For the sake of clarity, consider the information in Example 2 and Table 1 . The depicted mutant is identified by MuEPL as:

1. Operator = 2: the ROR operator is applied.
2. Location = 1: the first relational operator in the code is mutated.
3. Attribute = 1: the relational operator is changed by the first variant in the predefined set of attributes.

The MuEPL encoding has a flaw: the range of the location and attribute fields changes according to the operator. This is a problem when applying the EMT genetic operators (gene mutation and crossover), so EMT uses a *normalised* representation where the range of the Location and Attribute fields is the same regardless of the value of the Operator field:

- The normalised Location is an integer between 1 and the least common multiple of the number of locations for each operator. For instance, if we had two operators, one with 2 locations and one with 3, the range of Location would be between 1 and 6.

  When executing a mutant, a normalised Location L_i of the i-th operator is mapped back to the $\lceil (L_i \times l_i)/L \rceil$-th location, where l_i is the number of locations available for that i-th operator. In the example above, an individual for the first operator with a normalised Location of 4 would be mapped to the denormalised Location $\lceil (4 \times 2)/6 \rceil = 2$.

- The normalised Attribute is similar to the Location, being an integer between 1 and the least common multiple of the maximum values of the Attribute field for the various operators.

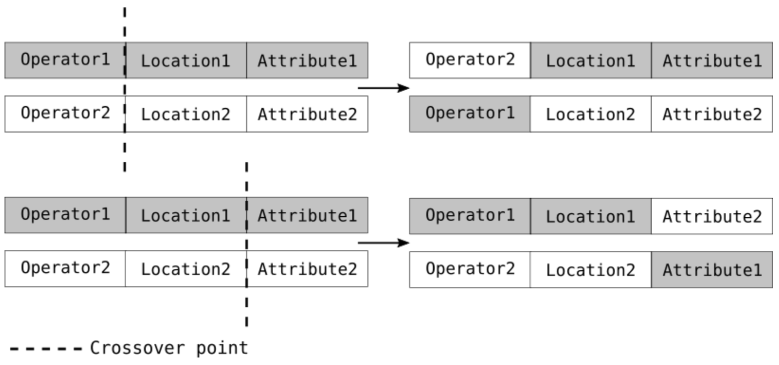

Figure 2 Crossover operator

2.4.2 | Main steps

The evolutionary algorithm starts by generating a random set of individuals as its first generation. Later generators are a combination of:

1. *Random generation*: a configuration parameter (percentage of new individuals, N) determines how many mutants will be randomly generated. Mutants are generated according to uniform distributions over the valid ranges of the three fields (Operator, Location, and Attribute), following the normalised ranges as mentioned above.

2. *Mutations and crossover*: the other new individuals are produced by modifying and combining pairs of parent individuals in the previous generation. The parents are picked through roulette selection [Gol89], with a probability proportional to their fitness[2]. The parents will them go randomly through one or two of these processes, according to further probabilities set in the configuration:

   - *Mutation*: one of the three fields (operator, location or attribute) is perturbed, while honouring the range of valid values for that field. If the original value was α, the maximum value for that field is U, and the probability of mutation is p_m, the new value is computed as:

   $$\beta = \alpha + \text{uniform\_random}(-10(1-p_m), 10(1-p_m)) \pmod{U}$$

   - *Crossover*: a random crossover point is selected between the fields of the individuals. This may result in either the operator or the attribute field being swapped across the individuals (see Figure 2).

The algorithm assigns fitness values to the mutants, favouring strong mutants. The fitness function counts the test cases which detect the mutant, and the mutants killed by those test cases. Mutants with high values for both will have low fitness, as they do not require very specific test cases. Potentially equivalent mutants will have the maximum fitness, as they have not been killed by any test case. Difficult to kill mutants will have the second best value, since they were killed by only one test case which killed no other mutant.

One special feature of EMT is that the fitness function evolves over time, since it is based on all the individuals generated so far.

2.5 | IoT Test Event Generator

IoT-TEG is a Java-based tool which takes an event type definition file and a desired output format (JSON, CSV, and XML, the most common across IoT platforms). IoT-TEG is made up of a *validator* and an *event generator* (Figure 3). The validator ensures the definition follows the rules set by IoT-TEG. The generator takes the definition and generates the indicated number of events according to it. IoT-TEG automatically generates a wide variety of events for the testing phase of any event-processing program in a user-friendly way, helping develop a more reliable event-based system.

Previous studies suggested there were no differences in testing effectiveness between using events generated by IoT-TEG, or events recorded from various case studies [GMMBDJ17]. These results confirm IoT-TEG can simulate many types of events occurring in industrial applications, and solve the main challenges developers face when they test event-processing programs:

1. Lack of data for testing,

2. needing specific values for the events, and

[2]Using a method with quick convergence such as roulette selection is intentional: the aim is to find the strong mutants as quickly as possible.

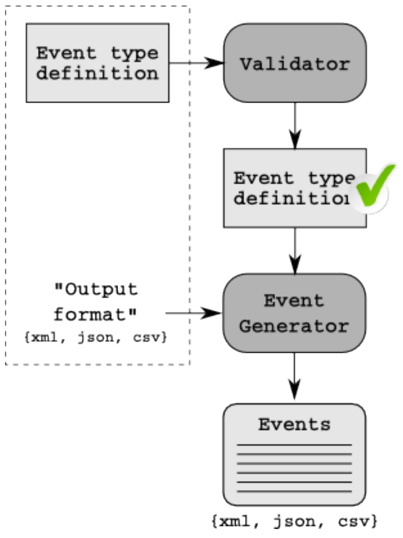

Figure 3 IoT-TEG Architecture.

3. needing the source to generate the events.

For the sake of clarity, Example 3 shows an event type definition that could be used to test the queries of Example 1.

Example 3: Event type definition example

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TemperatureEvent">
  <block name="feeds" repeat="150">
    <field name="created_at" quotes="true" type="ComplexType">
     <attribute type="Date" format="yy-MM-dd"></attribute>
     <attribute type="String" format="T"></attribute>
     <attribute type="Time" format="hh:mm"></attribute>
    </field>
    <field name="entry_id" quotes="false" type="Integer" min="0" max="10000">
    </field>
    <field name="temperature" quotes="false" type="Float" min="0" max="500" precision="1"></field>
  </block>
</event_type>
```

The defined event type contains three properties: `created_at`, `entry_id` and `temperature`. These properties are defined as fields in the event type definition. The `created_at` field is complex type and `entry_id` and `temperature` are simple types. The property that is evaluated in the Example 3 queries is `temperature`.

Apart from the mentioned challenges that IoT-TEG solves in order to test event-processing programs, it incorporates more functionality for testing. IoT-TEG supports defining the behaviour of the different properties of the event according to rules established by the user. So, in Example 3, the user could define the behaviour of the `temperature` property, or any other `TemperatureEvent` property, according to the test to apply. Moreover, IoT-TEG has a specific functionality to generate events for programs which use Esper EPL queries. This functionality helps to automatically generate events with specific values in accordance with the program which will process them. IoT-TEG analyses the Esper EPL queries and generates events depending on the logical and relational operations. Following the Example 1, the generated events will have their temperature higher than 400.

This paper explores the question of whether IoT-TEG can replace the hand-written test data generators that previously existed for the various EPL queries under test, by evaluating whether *EMT* is more or less effective with the events IoT-TEG generates.

3 | MUTATION TESTING AND EMT APPLIED TO ESPER EPL

In this section we highlight some concerns that have to be taken into account in order to understand how mutation testing and *EMT* have been applied to Esper EPL. A brief description of Esper EPL mutation operators as well as how the order of the events are considered to define the killing criteria is explained. Finally, the integration of MuEPL with *EMT* is presented.

Mutation operators have been defined and implemented for Esper EPL in previous works [GMSZ+12, GMDJMB15]. After analysing over 3700 queries in real projects, the operators were refined, and a complete classification was created [GM17]. There are 34 mutation operators across four categories, depending on the type of element they are related to. These mutation operators model common mistakes that programmers can make when implementing an Esper EPL query. Some operators are specific for Esper EPL, while others are adapted from other languages, specially from SQL [TSCIR07]. These are identified by these uppercase letters:

- P - *Pattern expression operators*
- W - *Windows operators*
- R - *Replacement operators*
- I - *SQL injection attack operators*

Table 2 lists the operators. As previously mentioned, some operators are specific to the Esper EPL language, while others have been adapted from SQL, which contains mutation operators in traditional languages, such as the operator RRO. The Esper EPL's specific operators are marked by ☆.

The mutation operators were implemented into the MuEPL tool. MuEPL can find which mutations are applicable, generates them and runs test suites against the mutants.

3.1 | Killing criteria

Once the mutants are generated, the test cases are run against them. A value is reported when the mutant fails to pass the test case, and another value is reported when it passes the test. These results are examined to determine whether the mutants are dead or if they are still alive after the test suite execution. The killing criteria must be defined to determine whether the mutants are killed or not.

To determine if a mutant is alive or not, we compare the number and content of the events of the original program and the mutant program. First, the number of processed events is checked, and then an analysis is conducted to verify that the processed events are the same although in different order. This criterion is the killing criteria.

It has to be taken into account that Esper EPL is a stream-oriented language, and not only the network latency can alter the order of the events, but also the number of involved EPL queries. In order to clarify this point, an example will be shown. The program used as the example is Ecological Island [RGOBP+18] (Section 4.3.1), which runs four queries. The events that this program receives are from the ThingSpeak platform [Thi]. Their structures (i.e., the different properties that the events contain) are the following (Example 4):

Example 4: Events from 16302 ThingSpeak channel.

```
{timestamp=2015-11-01T00:28:10Z, islandId=16302,
 filledPercentage=90, blocked=0, alertLevel=2,
 islandName=Island01, temperature=2}
{timestamp=2015-11-01T00:28:10Z, islandId=16302,
 blocked=0, alertLevel=4, islandName=Island01,
 temperature=2}
...
```

In Example 5, the first 10 output events of the first execution are shown.

Example 5: Ecological Island Execution 1.

```
... filledPercentage=50, blocked=0, alertLevel=1, islandName=Island01, temperature=2
... blocked=0, alertLevel=4, islandName=Island01, temperature=2
... filledPercentage=90, blocked=0, alertLevel=2, islandName=Island01, temperature=2
... blocked=0, alertLevel=4, islandName=Island01, temperature=51}
... filledPercentage=50, blocked=0, alertLevel=1, islandName=Island01, temperature=51}
... filledPercentage=50, blocked=0, alertLevel=1, islandName=Island01, temperature=45}
... filledPercentage=90, blocked=0, alertLevel=2, islandName=Island01, temperature=45}
... blocked=0, alertLevel=4, islandName=Island01, temperature=45}
... filledPercentage=90, blocked=0, alertLevel=2, islandName=Island01, temperature=69}
... blocked=0, alertLevel=4, islandName=Island01, temperature=69}
...
```

Operator		Description
		Pattern Expression Mutation
PFP	☆	Adds/Removes the parenthesis in a *followed by* operation
PGR	☆	Removes the *guard expressions*
PNR	☆	Removes the `not` the keyword of the negated conditional expressions in the pattern
POC	☆	Changes the order of the events in a *followed by* operation
POM	☆	Increases/Decreases the timer value by one unit in the pattern observer (`timer:at`, `timer:interval`)
PRE	☆	Replaces the query pattern by the pattern filter expression
		Windows Mutation
WLM	☆	Increases/Decreases the data window length by one
WTM	☆	Increases/Decreases the time window by one
WBL	☆	Turns a batch window length into an ordinary window length
WBT	☆	Turns a batch window time into an ordinary window time
		Injection Attack Mutation
IWR	☆	Removes the "where" condition from a query
ICN		negates the condition expression of a query
		Replacement Mutation
RAF		Replaces an aggregate function (`max`, `min`, `avg`, `sum`, `count`, `median`, `stddev`, `avedev`) by another of the same kind. The `distinct` keyword could also be added
RAO		Replaces an arithmetic operator (+, -, *, /, %) by another of the same kind
RBR		Replaces each between-condition `a between x AND y` by `a > x AND a <= y`, `a >= x AND a < y` and its negative
RGR		Removes a group-by expression, and adds an appropriate aggregation function
RJR		Replaces the keywords `inner`, `left outer`, `right outer`, `full outer`, `outer` of the `"join"` clause by another of the same kind
RLM		Exchanges a wildcard character (%, _) in the `"like"` expression patterns
RLA		Adds a wildcard character (%, _) at the beginning and at the end of the `"like"` expression patterns
RAW		Removes a wildcard character (%, _) at the beginning of the `"like"` expression patterns
RBW		Removes a wildcard character (%, _) at the end of the `"like"` expression patterns
RLO		Replaces a logical operator (`and`, `or`) by another of the same kind
RNO		Replaces a number `e` by `e + 1` and `e - 1`
RNW		Exchanges the keyword `is null` and `is not null`
ROM		Exchanges or adds (if it does not exist) a keyword `asc`, `desc` in `"order by"` expressions
ROS		changes the order of the properties in the ''`order by`'' expressions
RRO		Replaces a relational operator (=, <>, <, >, <=, >=) by another of the same kind
RRR_1	☆	Exchanges a single row function {(`cast`, `instanceof`), (`prevwindow`, `prevcount`)}
RRR_2	☆	Exchanges a single row function (`prev`, `prevtail`, `prior`)
RSC		Replaces or adds (if it does not exist) a keyword in the `select` clause (`rstream`, `irstream`). The `distinct` keyword can be also added
RSR_1		Replaces the keyword in type I subqueries (`all`, `any`, `some`) by another keyword of type I, type II or type III subqueries with the appropriate modifications
RSR_2		Replaces the keyword in type II subqueries, (`in`, `not in`) by another of the keywords of type II, type I or type III subqueries with the appropriate modifications
RSR_3		Exchanges, in type III subqueries, the keyword (`exist`, `not exist`)
RTU	☆	Replaces one time unit (`milliseconds`, `seconds`, `minutes`, `hours`, `days`) by another of the same kind

Table 2 Esper EPL mutation operator definitions

The Example 6 shows the first 10 output events after executing the Ecological Island program a second time under the same conditions.

Example 6: Ecological Island Execution 2.

```
... filledPercentage=90, blocked=0, alertLevel=2, islandName=Island01, temperature=2
... filledPercentage=50, blocked=0, alertLevel=1, islandName=Island01, temperature=2
... blocked=0, alertLevel=4, islandName=Island01, temperature=2
... filledPercentage=50, blocked=0, alertLevel=1, islandName=Island01, temperature=51}
... blocked=0, alertLevel=4, islandName=Island01, temperature=51}
... filledPercentage=90, blocked=0, alertLevel=2, islandName=Island01, temperature=45}
... blocked=0, alertLevel=4, islandName=Island01, temperature=45}
... filledPercentage=50, blocked=0, alertLevel=1, islandName=Island01, temperature=45}
... filledPercentage=90, blocked=0, alertLevel=2, islandName=Island01, temperature=69}
... blocked=0, alertLevel=4, islandName=Island01, temperature=69}
...
```

As the examples show, the order of the output events is different. The red, green and yellow output events in Example 5 are in a different order from those in Example 6. Hence, the output events are the same, but their order is different.

Moreover, we have to take into account that the mutants and the original programs must be executed by receiving the same events. To achieve this capability, the sources of the events in each program being tested must be modified to be the same. Because the generated mutants are based on the original program, the source used as the input in the original and the mutated programs is the same.

3.2 | Integration with EMT

The *EMT* evolutionary algorithm (which is language-agnostic needs to be complemented with language-specific mutation operators, such as those in MuEPL [GM17]. To fill this gap, an additional component (*bridge* from now on) was developed. The bridge does two things:

- Transform *GAmera* commands (analysis, generation, mutant execution) to MuEPL commands.
- Convert inputs and outputs between MuEPL and *GAmera*.

EMT may need to be revised based on the language being integrated, or the execution environment. This can be done thanks to the component-based architecture of *GAmera*, where parts of the algorithm can be swapped out for other implementations. Possible adaptations and how they affect to the algorithm genetic behaviour are explained in the following lines.

The presented adaptations are based on the mutant representation that *GAmera* recognises. As it was mentioned in Section 2.4, *GAmera* identifies a mutant by three fields: operator, location and attribute. Mutation operators can treat the attribute field in two ways:

- **Fixed range**: the possible number of mutations is the same for all locations. We can always replace a relational operator with any of the other relational operators.
- **Variable range**: the possible number of mutations depends on the location. For instance, we can only replace a variable reference with a reference to a compatible variable, and those will depend on where we are in the program.

Since the *EMT* algorithm needs fixed ranges to select mutants fairly, variable ranges are turned into fixed ranges by considering each variation as its own location, leaving the attribute field set to 1. This way, the same result can be achieved without changing the *GAmera* genetic algorithm.

As mentioned before, Esper EPL queries can be built and invoked within Java applications. To capture these queries for mutation, the mutation system includes a modified version of the Esper EPL libraries, which replaces the ones within the system under test. This is kept as an implementation detail of MuEPL, which is transparent to *GAmera*.

4 | EVALUATION

The previous sections introduced *EMT*, Esper EPL mutation testing and IoT-TEG. This section will evaluate the effectiveness of *EMT* for Esper EPL queries, whether using previously captured events, events generated by IoT-TEG, or a combination of the two.

4.1 | Research questions

The following research questions are raised:

RQ1: Can *EMT* reduce the computational complexity of mutation testing beyond that of random selection?

This can be further divided into two dual questions:

Table 3 Evolutionary algorithm parameters (Table 11 of [DEGM11])

Parameter	Value
Population size	5%
Randomly generated individuals	10%
Crossover probability	70%
Mutation probability	30%

RQ1.1: Given a target percentage of strong mutants, can *EMT* find it faster than random selection?

RQ1.2: Given a target percentage of all mutants, can *EMT* find more strong mutants than random selection?

RQ2: Are events from IoT-TEG better, just as good or worse for *EMT*?

RQ1 and its subquestions are equivalent to the research questions in the studies that evaluate *EMT* for WS-BPEL [DEGM11] and C++ [DPMBS+17]. These questions will be answered one more time but in an IoT context. Again, *EMT* would outperform random selection if it needed fewer mutants to find the same proportion of the strong mutants of the program under test. Multiple case studies will be used to check if *EMT* is still stable in performance across programs and configurations.

RQ2 is included to check if IoT-TEG [GM17] can cover real-world event types, or it needs to be improved. Many event types in practice are simple and the IoT-TEG generated events can be used instead of the original ones. For complex event types with multiple properties and varied value ranges, the IoT-TEG generated events can be used, but may need to be refined to obtain better results.

4.2 | Experimental process

After defining the research questions, the next part is collecting data to answer them. These were the general steps followed for each case study:

1. Procure the test suites: in addition to the test suite with the events recorded from the original source, we will have a "typical" test suite generated through IoT-TEG, and a combination of the typical IoT-TEG and original test suites. The "typical" test suite is selected among 30 different test suites to have the mean mutation score (proportion of non-equivalent mutants killed).

2. Execute all mutants against all the "typical" test suites, recording test results and test execution times into "simulation record" YAML files. This is mostly for saving time when evaluating many different configurations of *EMT* and random selection: rather than having to re-run the same mutants again and again, we can refer back to our records. Real execution times can always be recovered from the recorded times.

3. Extract the list of the strong mutants from the recorded test results. These are the mutants we want to find through *EMT* and random selection.

4. For each "typical" test suite, and each target percentage P within $\{30\%, 45\%, 60\%, 75\%, 90\%\}$:

   (a) Run *EMT* with 30 different seeds, stopping when we find $P\%$ of the strong mutants, and measuring how many mutants had to be run. Ideally, we want to stop before executing $P\%$ of all the mutants: the fewer, the better. 30 runs are used to evaluate the stability of the algorithm across seeds.

   EMT is a complex algorithm with many different parameters. This study uses the values recommended in the first study on *EMT* [DEGM11], which are listed in Table 3 . The population size specifies how many mutants will be produced in each generation, and it is a percentage of the total number of mutants for each application. As there are two ways of generating mutants (randomly and by crossover/mutation), the sum of both approaches is 100%.

   (b) Run *EMT* 30 more times with the same seeds and typical test suite, but stopping instead when we run $P\%$ of all mutants, and measuring how many strong mutants we found. In this case, we want to find more than $P\%$ of all the mutants: this would confirm that the search is being directed towards them.

   (c) Repeat the two steps above for random selection, with the same seeds and typical test suite. Instead of a generational approach, random selection will simply shuffle the list of mutants according to the seed, and then execute the mutants in sequence one by one until the stopping criteria is met.

5. To answer RQ1, compare the results of *EMT* and random selection: if *EMT* reduces the cost, it should find the target percentage of the strong mutants sooner, and it should generate more strong mutants given a target percentage of all mutants.

6. To answer RQ2, compare the results of *EMT* across i) the test suite from the recorded events, ii) the typical test suite generated by IoT-TEG, and iii) the combination between i) and ii). If IoT-TEG is equivalent to the original events, the results should be equivalent.

To know about the significance of the results using *EMT* and random selection executions, we have run a statistical test. We have used STATService [3], which selects an appropriate statistical test depending on the data (smart statistical test). The STATService proposed tests indicate that the case studies data are not normally distributed. For that reason, comparisons are done through the non-parametric *Mann-Whitney U* test using R [4]; the *Mann-Whitney U* test compares mean ranks across populations.

4.3 | Case studies

We have selected three case studies with different purposes and from different sources: *Ecological Island* [RGOBP$^+$18], *Terminal Self-Service* [Espa] and *DENMEvaP* [Gad15]. These case studies were chosen as they demonstrate how Complex Event Processing can derive more meaningful events from IoT device-to-cloud event streams, rather than simply routing events to endpoints by attributes as done in the Microsoft Azure IoT Hub [Mic]. In terms of scale, our selected case studies are similar or larger than technology demonstrator open testbeds such as the recently published "Asset Tracking" Eclipse Open IoT Testbed [Ecl18]: while it combines leading edge solutions by RedHat, Azul, Codenvy and Samsung, the demo instance only tracks 10 trucks at the date of writing.

Various large-scale wireless sensor network testbeds have been published in the literature over the last few years, such as FIT IoT-Lab or SmartSantander [TMV15]. However, at the time of experimentation these were generally either not publicly available, or did not allow for the integration of EsperTech EPL queries. In any case, while industrial applications of CEP in IoT would have more devices and therefore process more events, EPL would still be used in the same manner, and we would have similar challenges in writing correct queries.

The measurements of the various features of each case study can be seen in the appropriate subsection. These features are: the number of queries, the number of patterns, the number of time and length (T&L) windows, the number of time units, the number of `where` clauses and the percentage of presence of each feature according to the number of queries.

The number of mutants created with MuEPL in the listed programs is depicted in Table 4 . The mutants are calculated per operator and the total is added up at the end of the table. In addition, the average of Esper EPL mutants injected in each application are shown (this mean only considers the operators that produce at least one mutant).

Table 5 includes, for each program and Esper EPL-specific mutation operator, the number of the measured features, the number of locations where the operator can be applied (N) and the applicability (defined as a percentage) of the operators (A). Operators producing no mutants are not shown in Table 5 . As an example of the meaning of A, the operator PNR can be applied at two locations of *Terminal Service*; the number of patterns in this application is 3 (see Section 4.3.2), so the value of A in Table 5 is 66.7% (2/3). In addition, the total is added up at the end of the table as well as the average of the applied operators in each application.

Thanks to a previous execution of all the mutants, we know how many of them are strong with the current test suite (used as a ground truth to compute our results). The number of total mutants, valid mutants and the percentage of strong mutants of each case study are given in the appropriate subsection.

4.3.1 | Ecological Island

This program was implemented at the University of Cadiz [RGOBP$^+$18]. Its goal is to promote the development of *Smart Cities*. The ecological islands are gifted with "intelligence" in order to reduce business costs, reduce environmental pollution, and increase energy efficiency. The activities involved are: fire detection, blocked container input, half filled container and full filled container. The program contains four EPL queries, and obtains the JSON events from five channels of the *ThingSpeak* platform (a public IoT platform) [Thi].

Taking into account the different measured features contemplated in Table 5 and the four queries, it can be said that all four queries use a pattern, a time/length window, and a time unit. Therefore, these three features are applied 100% of the time.

4.3.2 | Terminal Self-Service

This case study is about a J2EE-based self-service terminal managing system in an airport that gets a large number of events (around 500 events per second) from connected terminals. Some events indicate abnormal situations and others observe activity:

[3] http://moses.us.es/statservice/
[4] https://www.r-project.org/

Oper.	Terminal	DENMEvap	Ecolog.	Total
PFP	3	15	0	18
PGR	0	0	0	0
PNR	2	1	0	3
POC	3	15	0	18
POM	2	2	0	4
PRE	3	15	4	22
WLM	2	0	0	2
WTM	2	8	8	18
WBL	1	0	0	1
WBT	1	4	4	9
RAF	0	45	0	45
RAO	4	28	0	32
RBR	0	0	0	0
RGR	2	20	0	22
RJR	0	0	0	0
RLM	0	0	0	0
RLA	0	0	0	0
RAW	0	0	0	0
RBW	0	0	0	0
RLO	4	29	8	41
RNO	2	68	46	116
RNW	0	0	0	0
ROM	0	0	0	0
ROS	0	0	0	0
RRO	35	700	75	810
RRR$_1$	0	0	0	0
RRR$_2$	0	0	0	0
RSC	30	185	20	235
RSR$_1$	0	0	0	0
RSR$_2$	0	0	0	0
RSR$_3$	0	0	0	0
RTU	20	72	16	108
IWR	1	17	0	18
ICN	1	17	0	18
Total	118	1241	181	1540
Mean	6.556	71.82	51.37	94.05

Table 4 Distribution of mutants by program and operator, classified by category (P: patterns, W: windows, R: replacement, I: SQL injection).

- Checkin - Indicates a customer started a check-in dialog.
- Cancelled - Indicates a customer cancelled a check-in dialog.
- Completed - Indicates a customer completed a check-in dialog.
- OutOfOrder - Indicates the terminal detected a hardware problem.
- LowPaper - Indicates the terminal is low on paper.

This example provided by EsperTech Inc. [Espb], with six EPL queries, receives from an integrated event generator a set of events. The events provide information about the source terminal. Each event carries similar information: the terminal identifier and a timestamp.

Taking into account the different measured features contemplated in Table 5 and the six queries, it can be said that: i) in three of the six queries a pattern appears (50% of application), ii) in two of them a time or length window appears (33.3%), iii) in five queries a time unit can be found (83.3%), and iv) there is one occurrence of a `where` clause in the six queries (16.7%).

	Terminal		DENMEvap		Ecological		Total	
Pattern	3		15		4		22	
Oper.	N	A (%)	N	A (%)	N	A (%)	N	A (%)
PFP	3	100.0	15	100.0	0	0.0	18	81.8
PNR	2	66.7	1	66.7	0	0.0	3	13.6
POC	3	100.0	15	100.0	0	0.0	18	81.8
POM	2	66.7	1	66.7	0	0.0	3	13.6
PRE	3	100.0	15	100.0	4	1.0	22	100.0
Windows	2		4		4		10	
Oper.	N	A (%)	N	A (%)	N	A (%)	N	A (%)
WLM	1	50.0	0	0.0	0	0.0	1	10.0
WTM	1	50.0	4	100.0	4	100.0	9	90.0
WBL	1	50.0	0	0.0	0	0.0	1	10.0
WBT	1	50.0	4	100.0	4	100.0	9	90.0
Time units	5		9		4		18	
Oper.	N	A (%)	N	A (%)	N	A (%)	N	A (%)
RTU	5	100.0	9	100.0	4	100.0	18	100.0
Where clauses	1		17		0		18	
Oper.	N	A (%)	N	A (%)	N	A (%)	N	A (%)
ICN	1	100.0	17	100.0	0	0	18	100.0
Total&Mean	23	62.2	81	69.2	16	40	120	58.8

Table 5 Quantitative statistics by category and program, for each of the Esper EPL-specific mutation operators.

4.3.3 | DENMEvaP

DENMEvaP (Distributed Event-driven Network Monitoring Evaluation Prototype) is a program developed by Gad [Gad15] which contains 37 EPL queries that process events in PCAP (Packet CAPturer) format. The program is divided into five modules which are monitoring different tasks. A minimal prototype consists of a network with at least two computers:

- The sensor node acts as event producer and acquires raw data in the monitored computer network.
- The data processing node hosts, e.g., the communication infrastructure, the CEP engine, and the event consumer that is used for measuring the end-to-end throughput.

Taking into account the different measured features contemplated in Table 5 and the thirty seven queries, it can be said that: in fifteen of the thirty seven queries a pattern appears (40.5% of application), in four of them a time or length window appears (10.8%), in nine queries a time unit can be found (24.3%) and there are twenty one occurrences of `where` clauses in the thirty seven queries (56.7%).

As it was mentioned before, DENMEvaP is divided into modules and in order to test the behaviour of each one a set of PCAP events has been generated. The test suite of the DENMEvaP program was obtained from the repository[5]. The author of DENMEvaP briefly explains the origin of each one. Some of the events were created with Ettercap[6], to simulate different attacks to the system. Others were obtained doing different

[5]https://github.com/fg-netzwerksicherheit/DENMEvaP
[6]https://ettercap.github.io/ettercap/

simulations to capture the traffic during congested periods thanks to Netem[7]. Another set of events was obtained under a simulated attack using the Mausezahn packet generator[8].

4.4 | Discussion of results

In the following subsections the results of the different case studies will be analysed. First, the contents and origins of the test suites will be described. Next, several comparisons are done to answer the research questions:

- *EMT* versus random selection, through two metrics: how many mutants were needed to find S% of the strong mutants, and how many strong mutants were found after running M% of all mutants.
- Original test suite versus the one produced by IoT-TEG, using the same metrics.

Finally, an improvement in *EMT* is done and the previous executions are compared against the new results.

4.4.1 | Test suites

As we mentioned, the origins and the content of the test suites of the analysed case studies will be explained. In order to answer RQ2, the IoT-TEG generated events have been generated following the structure and behaviour of the original events of each case study.

Ecological Island

Test suite 1 has the results produced by running either 5 test cases derived from the ThingSpeak platform, or 5 test cases generated by IoT-TEG. We conflated these results in Table 6 as they produced identical executions:

- The ThingSpeak test cases illustrate one common problem when testing IoT systems: the event source may not produce enough examples, as mentioned in Section 2.5. This is the case for Ecological Island: the event channels were available, but not sending out events. For this reason, only 5 test cases provided by the ThingSpeak channels were obtained. Each test case contains 100 events.
- Alternatively, we may generate events through IoT-TEG. We used it to generate 30 test suites, each with 5 test cases, and each test case containing 100 events. We ran all the test suites against all the mutants. The test suite with the mean mutation score was entered into *EMT*.

Test suite 2 combines the ThingSpeak and IoT-TEG test suites into a single test suite with 10 test cases. Finally, *Test suite 3* adds 5 more test cases from IoT-TEG. The execution steps are the same as in previous phases.

Terminal Self-Service

The original event source for this case study is a handmade event generator. We used the original generator and IoT-TEG to produce two different sets of 30 test suites, with 5 test cases each. For each set of test suites, we picked the test suite with the mean mutation score. *Test suite 1* is the one from the original generator, and *Test suite 2* is the one from IoT-TEG. Given that the original generator produced a random number of events between 350 and 370, IoT-TEG was made to produce a random number of events within the same range. *Test suite 3* combines both test suites into a single test suite with 10 test cases.

DENMEvaP

As mentioned in Subsection 4.3.3, the program is divided into five modules. Three modules were discarded from the study because all their mutants could be easily killed with their original test suites. The BruteForce module had 6 queries, where 286 mutants could be injected. SnifferCongestion had 4 queries and produced 61 mutants. Each of these two modules was tested through two different test suites:

- *Test suite 1* was the original network packet capture in PCAP format, as produced by the author of DENMEvaP. The PCAP file for BruteForce had 6917 events, and the one for SnifferCongestion had 133 events.
- *Test suite 2* was generated through IoT-TEG and had the same number of events as the original PCAP file. DENMEvaP was adapted so it could read the JSON files generated by IoT-TEG.

[7]https://wiki.linuxfoundation.org/networking/netem
[8]http://www.perihel.at/sec/mz/mzguide.html

	EMT		Random		
S%	Mean	SD	Mean	SD	p-value
Test suite 1 - 5 ThingSpeak/5 IoT-TEG					
0.15	15.38	4.42	14.09	3.01	0.81
0.30	26.19	4.73	29.26	4.86	<0.01
0.45	39.12	4.60	44.77	5.02	<0.01
0.60	52.73	4.77	58.67	5.17	<0.01
0.75	67.16	4.80	72.38	4.96	<0.01
0.90	84.18	5.26	87.81	3.30	<0.01
Test suite 2 - 5 ThingSpeak + 5 IoT-TEG					
0.15	14.88	3.33	14.09	3.01	0.81
0.30	26.21	4.09	29.26	4.86	<0.01
0.45	37.99	5.34	44.77	5.02	<0.01
0.60	51.34	5.22	58.67	5.17	<0.01
0.75	67.37	4.57	72.38	4.96	<0.01
0.90	84.70	2.45	87.81	3.30	<0.01
Test suite 3 - 5 ThingSpeak + 10 IoT-TEG					
0.15	15.27	3.31	14.09	3.01	0.93
0.30	27.55	4.35	29.26	4.86	0.09
0.45	39.80	3.48	44.77	5.02	<0.01
0.60	52.41	4.53	58.67	5.17	<0.01
0.75	66.83	5.00	72.38	4.96	<0.01
0.90	84.68	4.12	87.81	3.30	<0.01

Table 6 Percentages of all mutants executed to find S% of the strong mutants with *EMT* and random shuffle for Ecological Island (less is better), and p-values for the Mann-Whitney U tests with "*EMT* < random" alternative.

	EMT		Random		
M%	Mean	SD	Mean	SD	p-value
Test suite 1 - 5 ThingSpeak/5 IoT-TEG					
0.15	16.50	5.69	15.14	3.52	0.10
0.30	34.48	5.55	30.49	4.63	<0.01
0.45	50.22	5.05	44.70	5.02	<0.01
0.60	67.05	4.29	60.55	5.82	<0.01
0.75	80.16	5.08	75.19	4.49	<0.01
0.90	91.42	3.41	89.89	3.02	0.07
Test suite 2 - 5 ThingSpeak + 5 IoT-TEG					
0.15	16.99	4.96	15.14	3.52	0.05
0.30	35.19	5.87	30.49	4.63	<0.01
0.45	51.58	5.51	44.70	5.02	<0.01
0.60	67.10	4.35	60.55	5.82	<0.01
0.75	80.22	3.42	75.19	4.49	<0.01
0.90	92.90	2.45	89.89	3.02	<0.01
Test suite 3 - 5 ThingSpeak + 10 IoT-TEG					
0.15	16.50	4.45	15.14	3.52	0.17
0.30	33.06	4.95	30.49	4.63	0.03
0.45	50.87	5.99	44.70	5.02	<0.01
0.60	66.89	4.56	60.55	5.82	<0.01
0.75	79.67	4.34	75.19	4.49	<0.01
0.90	92.62	2.81	89.89	3.02	<0.01

Table 7 Percentages of strong mutants found after executing M% of all mutants with *EMT* and random shuffle for Ecological Island (more is better), and p-values for the Mann-Whitney U tests with "*EMT* > random" alternative.

4.4.2 | RQ1: Can EMT reduce the cost beyond random selection?

With regards to cost, each case study has been studied in two ways: RQ1.1 checks if we can reach the desired strong mutants faster with *EMT*, and RQ1.2 checks if we find more strong mutants if we decide to run a certain percentage of all mutants. RQ1.1 is only possible within a study, since we have previously inspected the mutants manually to know if they are strong or not. RQ1.2 represents the practical scenario when users will define a "search budget", specifying how many of all the mutants they wish to look through.

For each case study, two tables will be used to present their results: one for RQ1.1 and one for RQ1.2. The RQ1.1 tables contain, for each test suite, the mean and standard deviation of the proportion of all the mutants needed by EMT and random selection to find S% of the strong mutants, where S% goes in increments of 15%. The RQ1.2 tables are dedicated to the proportion of strong mutants that were found after executing M% of all the mutants. The RQ1.1 and RQ1.2 tables include p-values comparing both executions using Mann-Whitney U tests with "*EMT* produces better results than random selection" as the alternative hypothesis.

Ecological Island

Tables 6 and 7 contain the results for the Ecological Island case study. The total number of mutants is 181, where 61 are strong (33.7%).

Judging from the p-values, it can be generally concluded that *EMT* outperforms random selection once 45% or more of the strong mutants are desired, or 45% or more of all mutants are executed. Another important observation is that the means do not change much between test suites, suggesting that the number of test cases does not influence significantly the results.

	EMT		Random		
S%	Mean	SD	Mean	SD	p-value
Test suite 1 - 5 Original Generator					
0.15	14.86	2.01	14.80	2.53	0.50
0.30	29.32	2.89	29.55	2.87	0.42
0.45	44.60	2.85	45.23	2.81	0.21
0.60	58.16	3.11	60.40	3.38	<0.01
0.75	74.32	2.53	75.82	2.22	0.01
0.90	89.52	2.00	90.54	2.00	0.02
Test suite 2 - 5 IoT-TEG					
0.15	14.60	2.29	14.80	2.53	0.26
0.30	29.29	2.95	29.55	2.87	0.39
0.45	44.12	3.33	45.23	2.81	0.07
0.60	58.28	2.49	60.40	3.38	<0.01
0.75	74.52	2.46	75.82	2.22	0.01
0.90	89.63	2.35	90.54	2.00	0.07
Test suite 3 - Both					
0.15	15.14	2.49	14.80	2.53	0.62
0.30	30.40	3.17	29.55	2.87	0.88
0.45	44.44	2.63	45.23	2.81	0.20
0.60	58.84	2.41	60.40	3.38	0.03
0.75	74.21	2.33	75.82	2.22	<0.01
0.90	90.28	2.18	90.54	2.00	0.34

Table 8 Percentages of all mutants executed to find S% of the strong mutants with *EMT* and random shuffle for Terminal Self-Service (less is better), and p-values for the Mann-Whitney U tests with "*EMT* < random" alternative.

	EMT		Random		
M%	Mean	SD	Mean	SD	p-value
Test suite 1 - 5 Original Generator					
0.15	15.61	2.08	14.35	2.73	0.02
0.30	30.55	2.91	29.79	2.50	0.16
0.45	46.20	2.87	44.85	3.38	0.05
0.60	61.94	2.59	59.66	2.80	<0.01
0.75	76.58	2.27	74.68	1.99	<0.01
0.90	91.52	1.79	90.38	1.78	<0.01
Test suite 2 - 5 IoT-TEG					
0.15	15.78	2.63	14.35	2.73	<0.01
0.30	31.31	3.32	29.79	2.50	0.03
0.45	46.33	3.57	44.85	3.38	0.03
0.60	62.19	2.57	59.66	2.80	<0.01
0.75	76.46	2.52	74.68	1.99	<0.01
0.90	91.65	2.19	90.38	1.78	0.02
Test suite 3 - Both					
0.15	15.27	2.62	14.35	2.73	0.11
0.30	29.58	3.43	29.79	2.50	0.75
0.45	45.99	2.52	44.85	3.38	0.11
0.60	61.56	2.34	59.66	2.80	<0.01
0.75	76.84	2.74	74.68	1.99	<0.01
0.90	90.89	2.09	90.38	1.78	0.20

Table 9 Percentages of strong mutants found after executing M% of all mutants with *EMT* and random shuffle for Terminal Self-Service (more is better), and p-values for the Mann-Whitney U tests with "*EMT* > random" alternative.

Terminal Self-Service

The results for Terminal Self-Service are in Tables 8 and 9 . The total number of mutants is 118, where 79 are strong (66.9%).

In this case, most of the p-values are over 0.01, so in general we cannot reject the null hypothesis that *EMT* performs similarly to random selection. Still, we can see that there are scenarios in which *EMT* can outperform random selection, and this is especially evident with test suite 2 for Table 9 , where p-values are consistently below 0.05. This suggests that for this particular case study, the specific choice of test cases (rather than the number of them) may play a factor in the performance of *EMT*.

DENMEvaP — BruteForce

Tables 10 and 11 have the results for BruteForce. The total number of mutants is 286, where 248 are strong (86.7%).

The p-values in this case are very high, so there is very little evidence supporting the idea that *EMT* outperforms random selection. In fact, it is quite the opposite: the means are consistently better for random selection. Interestingly, in this case study the results were identical across test suites (both means and standard deviations), suggesting that the specific choice of test cases nor the test results did not influence the results. This may point to a simplicity in the queries themselves that may lend itself poorly to a more complex strategy like *EMT*.

DENMEvaP — SnifferCongestion

Tables 12 and 13 show the results for SnifferCongestion. The total number of mutants is 61, where 34 are strong (55.7%) for *Test suite 1*, 49 are strong (80.3%) in *Test suite 2* and 33 are strong (54.1%) for *Test suite 3*.

	EMT		Random		
S%	Mean	SD	Mean	SD	p-value
Test suite 1 - PCAP					
0.15	16.53	1.23	15.18	0.53	1.00
0.30	32.30	1.01	29.86	0.91	1.00
0.45	47.89	1.32	44.61	1.08	1.00
0.60	62.52	0.98	59.58	1.21	1.00
0.75	77.17	0.62	74.68	1.01	1.00
0.90	90.81	0.23	89.57	0.69	1.00
Test suite 2 - JSON					
0.15	16.53	1.23	15.18	0.53	1.00
0.30	32.30	1.01	29.86	0.91	1.00
0.45	47.89	1.32	44.61	1.08	1.00
0.60	62.52	0.98	59.58	1.21	1.00
0.75	77.17	0.62	74.68	1.01	1.00
0.90	90.81	0.23	89.57	0.69	1.00
Test suite 3 - PCAP and JSON					
0.15	16.53	1.23	15.18	0.53	1.00
0.30	32.30	1.01	29.86	0.91	1.00
0.45	47.89	1.32	44.61	1.08	1.00
0.60	62.52	0.98	59.58	1.21	1.00
0.75	77.17	0.62	74.68	1.01	1.00
0.90	90.81	0.23	89.57	0.69	1.00

Table 10 Percentages of all mutants executed to find S% of the strong mutants with *EMT* and random shuffle for BruteForce (less is better), and p-values for the Mann-Whitney U tests with "*EMT* < random" alternative.

	EMT		Random		
M%	Mean	SD	Mean	SD	p-value
Test suite 1 - PCAP					
0.15	14.06	1.17	14.68	0.56	0.99
0.30	27.82	0.96	29.99	0.84	1.00
0.45	42.07	1.24	45.01	1.12	1.00
0.60	57.12	1.13	60.07	1.17	1.00
0.75	72.63	0.75	75.19	1.02	1.00
0.90	88.97	0.33	90.20	0.66	1.00
Test suite 2 - JSON					
0.15	14.06	1.17	14.68	0.56	0.99
0.30	27.82	0.96	29.99	0.84	1.00
0.45	42.07	1.24	45.01	1.12	1.00
0.60	57.12	1.13	60.07	1.17	1.00
0.75	72.63	0.75	75.19	1.02	1.00
0.90	88.97	0.33	90.20	0.66	1.00
Test suite 3 - PCAP and JSON					
0.15	14.06	1.17	14.68	0.56	0.99
0.30	27.82	0.96	29.99	0.84	1.00
0.45	42.07	1.24	45.01	1.12	1.00
0.60	57.12	1.13	60.07	1.17	1.00
0.75	72.63	0.75	75.19	1.02	1.00
0.90	88.97	0.33	90.20	0.66	1.00

Table 11 Percentages of strong mutants found after executing M% of all mutants with *EMT* and random shuffle for BruteForce (more is better), and p-values for the Mann-Whitney U tests with "*EMT* > random" alternative.

The p-values are still very high, and the means are consistently better for random selection than for *EMT*. The number of test cases seems to have a minor influence in the results, judging from the slight drop in performance (1–4%) for both *EMT* and random selection with *Test suite 3* when compared to the other test suites.

Overall results

In general, *EMT* outperformed random selection in one case study (Ecological Island), did roughly the same in another (Terminal Self-Service), and did worse in the two DENMEvaP ones. This suggests that there may be features of these case studies which reduce the effectiveness of *EMT*. In the Terminal Self-Service case study, *EMT* performed better with a specific test suite. The DENMEvaP BruteForce case study may lack the complexity required to merit the use of *EMT*, since results were exactly the same regardless of the chosen test suite. The DENMEvaP SnifferCongestion may have had too few mutants to give *EMT* space to evolve a solution (61 versus the 181 in Ecological Island).

In general, we see the need to improve *EMT* to guide the search more explicitly for smaller mutant spaces, and to further evaluate the fitness landscape of the four case studies to see if there are significant differences which may explain the varying levels of performance of *EMT*.

4.4.3 | RQ2: Are events from IoT-TEG better, just as good or worse for EMT?

In order to answer RQ2, the results of the previous tables will be compared, especially those from *Test suite 1* and *Test suite 2* of each case study.

- As discussed in Section 4.3.1, the Ecological Island *Test suite 1* (Tables 6 and 7) conflates the identical results of two separate test suites: one from the ThingSpeak platform, and one from IoT-TEG. The results were the same, so it did not matter whether we used the original events or IoT-TEG for this case study.

	EMT		Random		
S%	Mean	SD	Mean	SD	p-value
Test suite 1 - PCAP Original Generator					
0.15	16.67	3.25	15.03	4.35	0.98
0.30	31.31	4.50	29.34	4.76	0.89
0.45	45.46	5.90	42.62	5.97	0.96
0.60	58.96	5.88	56.94	5.29	0.95
0.75	73.44	4.48	71.04	5.57	0.94
0.90	86.61	4.09	85.96	3.84	0.75
Test suite 2 - JSON IoT-TEG					
0.15	18.20	2.60	14.26	2.40	1.00
0.30	31.86	3.00	28.31	2.82	1.00
0.45	46.72	2.75	44.15	3.07	1.00
0.60	60.93	2.83	58.03	2.71	1.00
0.75	74.32	2.63	72.46	2.56	0.99
0.90	89.73	1.87	88.58	2.49	0.95
Test suite 3 - Both					
0.15	14.21	3.20	12.02	4.66	1.00
0.30	28.80	5.00	28.03	4.88	0.64
0.45	43.17	6.60	41.42	6.28	0.85
0.60	57.65	7.04	55.90	5.60	0.87
0.75	72.08	4.80	70.66	5.62	0.82
0.90	87.81	3.96	85.57	3.96	0.99

Table 12 Percentages of all mutants executed to find S% of the strong mutants with *EMT* and random shuffle for SnifferCongestion (less is better), and p-values for the Mann-Whitney U tests with "*EMT* < random" alternative.

	EMT		Random		
M%	Mean	SD	Mean	SD	p-value
Test suite 1 - PCAP Original Generator					
0.15	16.76	3.55	15.78	4.47	0.17
0.30	30.69	5.97	30.88	5.39	0.43
0.45	44.90	6.41	46.47	6.10	0.87
0.60	59.61	5.62	62.45	5.87	0.96
0.75	75.78	5.05	76.47	4.50	0.69
0.90	90.10	3.58	90.78	3.06	0.80
Test suite 2 - JSON IoT-TEG					
0.15	15.85	2.87	16.46	2.57	0.94
0.30	30.27	2.84	31.50	3.02	0.97
0.45	44.56	2.74	46.19	2.86	0.99
0.60	59.73	2.88	61.50	2.50	0.99
0.75	75.10	3.19	76.26	2.71	0.93
0.90	90.00	1.88	90.61	1.90	0.89
Test suite 3 - Both					
0.15	16.97	3.70	15.86	4.82	0.20
0.30	30.71	6.30	30.61	5.70	0.40
0.45	46.06	7.14	45.86	6.45	0.53
0.60	60.20	6.05	61.92	5.99	0.85
0.75	76.06	5.65	76.16	4.89	0.49
0.90	89.29	4.04	90.81	3.33	0.95

Table 13 Percentages of strong mutants found after executing M% of all mutants with *EMT* and random shuffle for SnifferCongestion (more is better), and p-values for the Mann-Whitney U tests with "*EMT* > random" alternative.

- *Test suite 1* and *Test suite 2* of Terminal Self-Service (Tables 8 and 9) are from the original event generator and IoT-TEG, respectively. Both test suites were picked as having the median mutation score among 30 similarly generated test suites. The same median mutation score was obtained for both. The results for *EMT* are quite similar across test suites, as well as those for random selection.
- The results of *Test suite 1* and *Test suite 2* of BruteForce can be seen in Tables 10 and 11 . Again, the results using *EMT* and random selection are the same whether using the original program or IoT-TEG.
- The results of *Test suite 1* and *Test suite 2* of SnifferCongestion can be seen in Tables 10 and 11 . While finding S% of the strong mutants (Table 10), there is no noticeable difference between the original events or the IoT-TEG ones. However, there is a difference when we are comparing the percentage of strong mutants found after executing M% of all mutants (Table 11).

After comparing the behaviour of the IoT-TEG generated events against the original test events, we can provide a general answer for RQ2: the IoT-TEG events are as good as the original ones for the purposes of testing, since we could only find one case where they made a difference. In addition, IoT-TEG can address the challenges mentioned in Section 2.5 that the developers face when they are testing event-processing programs: lack of data for testing, needing specific values for the events, and needing the source to generate the events.

4.5 | Guided EMT

The previous section did not show *EMT* as a consistent winner over a random shuffle of the individuals. As an evolutionary algorithm, this could be due to several factors: lack of diversity in the newly generated individuals, crossover and mutation operators that guide the search away from the strong mutants, or a badly designed fitness function. Given that *EMT* performed well with WS-BPEL and C++, our intuition was that strong mutants in Esper EPL queries were distributed differently than in WS-BPEL/C++ programs, causing issues with the new individual generator and the mutation operators. We decided to perform a few refinements in *EMT* to further guide the search, and compare the results.

4.5.1 | Introduced changes

An inspection of Table 4 shows that the mutants in the various queries are heavily concentrated in the RNO, RRO, RSC, and RTU operators. This tends to happen with operators that mutate commonly used language features. The original *EMT* individual generator produced individuals by generating values uniformly for all three fields (Operator, Location and Attribute), which meant that even if an operator had many more locations than another, we would have the same probability of generating an individual from either. This meant that the individual generator would not achieve a uniform distribution over the mutants. Guided *EMT* instead picks a random mutant across the entire list of all the mutants for a program, ensuring all mutants have the same probability to be chosen.

Another change was replacing the uniformly distributed random perturbations in the genetic mutation operator with something more targeted. In [ZST05] a new operator, called guided mutation, is introduced: it generates offspring through combination of global statistical information and the location information of solutions found so far. After some initial tests with a direct re-implementation of guided mutation, we found that a simpler approach performed better for the various case studies. Our new guided mutation operator behaves very similar to the original one:

1. First, it chooses randomly which field to mutate, following a uniform distribution.
2. If it is mutating the Location or Attribute fields, it works as usual.
3. If it is mutating the Operator field, it generates a number x between 0 and 1:
   (a) If $x < \alpha$ (where α is the "sampling rate" and is a parameter of the operator, currently set to 0.9), the new behaviour kicks in. It will choose randomly an operator, with a probability proportional to the proportion of all strong mutants found in that operator so far. This favours language features in the queries that have not been tested well so far.
   (b) Otherwise, it uses the random perturbation mentioned in Section 2.4.2.

4.5.2 | Case study results

Tables 14 to 20 compare *guided EMT* with the original version of EMT and random selection, in terms of how many mutants had to be run to find a certain proportion of the strong mutants. Likewise, Tables 15 to 21 compare guided EMT with the other options, in terms of how many strong mutants were found when running a certain proportion of all the mutants.

We can see the following results across the various case studies:

- Ecological Island: *guided EMT* produces the best results overall, both for RQ1.1 and RQ1.2.
- Terminal Self-Service: random selection is still the best for RQ1.1, and regular *EMT* is the best option for high percentages in RQ1.2.
- BruteForce: *guided EMT* does much better across the board than regular *EMT*, but still not as well as random selection. *Guided EMT* only produced better results for some values of M%.
- SnifferCongestion: *guided EMT* does better in some scenarios than traditional *EMT*, but it is rather contested. It is still outperformed for the most part by random selection: only in some M% percentages the guided EMT obtains better results.

After comparing all the tables, generally it can be said that the *guided EMT* is better than the original *EMT* where the original *EMT* excelled, intensifying its good points. However, it does not fully solve yet the weaknesses in the Terminal Self-Service and DENMEvaP case studies. As we mentioned, the mutants in the various queries are heavily concentrated in some specific operators that mutate commonly used language features. The improvement in the EMT helps to have more probability of generating these mutants which have many more locations than another. As a consequence, the results of *guided EMT* are better than the results of the original *EMT*.

Despite of the improvements in EMT, random selection appears in several executions as the best option. With the exception of the Ecological Island case study, in the other ones random selection is the best option to find the S% of the strong mutants and in the majority of cases to obtain a high percentage of strong mutants after executing M% of all mutants.

The Terminal Self-Service and DENMEvaP case studies have in common one factor: their original events have been generated. Their event source is not a sensor or a server that sends the events in real time, but a generator that tries to meet the query conditions. It could be argued that *EMT* did well with the events generated by IoT-TEG for Ecological Island, but this generation process was designed to mimic the originally

S%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - 5 ThingSpeak/5 IoT-TEG				
0.15	13.68	3.82	0.04	0.18
0.30	23.92	4.18	0.03	<0.01
0.45	34.81	4.07	<0.01	<0.01
0.60	48.45	4.15	<0.01	<0.01
0.75	63.00	5.86	<0.01	<0.01
0.90	80.02	5.86	<0.01	<0.01
Test suite 2 - 5 ThingSpeak + 5 IoT-TEG				
0.15	13.94	3.48	0.06	0.30
0.30	24.48	4.92	0.05	<0.01
0.45	36.34	5.34	0.09	<0.01
0.60	50.02	5.26	0.15	<0.01
0.75	65.36	5.91	0.07	<0.01
0.90	81.20	5.21	<0.01	<0.01
Test suite 3 - 5 ThingSpeak + 10 IoT-TEG				
0.15	13.37	3.11	0.01	0.15
0.30	23.46	4.65	<0.01	<0.01
0.45	35.58	4.80	<0.01	<0.01
0.60	48.88	4.82	<0.01	<0.01
0.75	63.98	6.45	0.03	<0.01
0.90	80.99	4.78	<0.01	<0.01

Table 14 Percentages of all mutants executed to find S% of the strong mutants with guided *EMT* for Ecological Island (less is better), and p-values for the Mann-Whitney U tests with "GuidedEMT < *EMT*" and "GuidedEMT < random" alternatives.

M%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - 5 ThingSpeak/5 IoT-TEG				
0.15	20.33	5.70	<0.01	<0.01
0.30	38.96	5.66	<0.01	<0.01
0.45	55.63	5.65	<0.01	<0.01
0.60	71.04	5.37	<0.01	<0.01
0.75	84.04	4.71	<0.01	<0.01
0.90	94.10	2.91	<0.01	<0.01
Test suite 2 - 5 ThingSpeak + 5 IoT-TEG				
0.15	19.56	5.51	0.03	<0.01
0.30	37.70	6.44	0.01	<0.01
0.45	54.59	6.36	0.06	<0.01
0.60	68.69	4.92	0.11	<0.01
0.75	82.68	4.43	<0.01	<0.01
0.90	93.44	3.04	0.14	<0.01
Test suite 3 - 5 ThingSpeak + 10 IoT-TEG				
0.15	20.27	6.16	<0.01	<0.01
0.30	39.02	6.15	<0.01	<0.01
0.45	55.63	5.36	<0.01	<0.01
0.60	70.11	5.18	<0.01	<0.01
0.75	82.35	4.57	0.01	<0.01
0.90	93.83	2.71	0.05	<0.01

Table 15 Percentages of strong mutants found after executing M% of all mutants with guided *EMT* for Ecological Island (more is better), and p-values for the Mann-Whitney U tests with "GuidedEMT > *EMT*" and "GuidedEMT > random" alternatives.

captured events from the real sensors connected to the ThingSpeak platform. For this reason, we cannot detect too much of a difference between the original and IoT-TEG generated test suites. Moreover, the DENMEvaP case study test suite is composed by one test case for each module, and neither *EMT* nor *guided EMT* have enough information to generate the strong mutants.

4.5.3 | Studying the EMT fitness landscapes

Given the large difference in results between the various case studies, we decided to investigate further on where these could come from. As a search-based technique, the performance of EMT is dependent on the characteristics of the search space created by each specific combination of test suite and program under test: depending on the distribution of fitness values for the individuals, EMT may behave better or worse.

This distribution is known as a *fitness landscape*. Pitzer and Affenzeller [PA12] defined a fitness landscape $\mathscr{F}$ as $(\mathscr{S}, f, d)$, a set of the two functions f and d that define the fitness value and distances between solutions in $\mathscr{S}$. In our case, $\mathscr{S}$ would be the set of all normalized individuals, f would be the EMT fitness function, and a first attempt at d could be the Euclidean distance between two normalized individuals a and b (with o_x, l_x, and a_x as the operator, location and attribute of the individual):

$$d(a,b) = \sqrt{(o_a - o_b)^2 + (l_a - l_b)^2 + (a_a - a_b)^2}$$

Given a fitness landscape, we may want to apply various analysis techniques to it. One of the most common techniques is *fitness-distance correlation*: if fitness is strongly negatively correlated with distance to the global optima, then we know that solutions become gradually better the closer we get to it. In other words, there is a "slope" which guides the search. Unfortunately, this technique is not a good fit for EMT, where we may have as many global optima as strong mutants: we could use the "average" distance to the global optima, but it would not be as representative.

Studying the *density of states* is a better option in our case. The idea was first proposed in the 90s by Rosé, Ebeling and Asselmeyer [REA96]. In its original form, the search space is sampled (due to its huge size in traditional problems) to estimate the proportion of solutions that have certain

S%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - 5 Original Generator				
0.15	16.19	2.97	0.95	0.94
0.30	30.99	3.47	0.98	0.97
0.45	45.79	3.14	0.95	0.80
0.60	60.45	2.37	1.00	0.48
0.75	75.85	2.29	0.99	0.49
0.90	91.05	1.79	1.00	0.79
Test suite 2 - 5 IoT-TEG				
0.15	15.99	2.90	0.97	0.90
0.30	31.30	3.62	0.99	0.97
0.45	45.65	3.47	0.95	0.62
0.60	60.71	3.39	1.00	0.56
0.75	76.27	3.24	0.99	0.75
0.90	91.30	1.90	1.00	0.91
Test suite 3 - Both				
0.15	16.61	2.59	0.99	0.99
0.30	31.55	4.49	0.86	0.97
0.45	46.07	3.93	0.98	0.85
0.60	61.33	3.63	1.00	0.83
0.75	76.27	3.09	1.00	0.73
0.90	91.81	1.81	1.00	0.99

Table 16 Percentages of all mutants executed to find S% of the strong mutants with guided *EMT* for Terminal Self-Service (less is better), and p-values for the Mann-Whitney U tests with "GuidedEMT < *EMT*" and "GuidedEMT < random" alternatives.

M%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - 5 Original Generator				
0.15	15.40	3.22	0.52	0.06
0.30	29.62	3.71	0.86	0.57
0.45	44.98	3.25	0.96	0.49
0.60	60.59	2.34	0.95	0.11
0.75	75.36	2.44	0.95	0.10
0.90	90.25	1.97	0.99	0.63
Test suite 2 - 5 IoT-TEG				
0.15	15.27	3.13	0.76	0.11
0.30	29.49	3.67	0.96	0.55
0.45	44.94	3.43	0.95	0.39
0.60	60.38	3.61	0.97	0.14
0.75	74.56	2.75	0.99	0.53
0.90	90.08	1.91	1.00	0.76
Test suite 3 - Both				
0.15	15.06	3.12	0.60	0.18
0.30	29.16	4.44	0.51	0.71
0.45	44.18	3.94	0.96	0.70
0.60	59.66	3.24	1.00	0.57
0.75	74.81	3.19	0.99	0.51
0.90	89.37	1.78	1.00	0.98

Table 17 Percentages of strong mutants found after executing M% of all mutants with guided *EMT* for Terminal Self-Service (more is better), and p-values for the Mann-Whitney U tests with "GuidedEMT > *EMT*" and "GuidedEMT > random" alternatives.

fitness values. Conceptually speaking, a search-based optimisation problem is easy if the density doesn't sharply drop off as fitness becomes higher. Conversely, if there is a sudden drop in the density after a certain fitness value, it will be very hard to find solutions that are better than that.

Interestingly, the challenge in EMT for our EPL queries is not the size of the search space, but in the cost of computing the fitness values (which require running a complex software system against collections of inputs). We can skip sampling and simply check the fitness of every individual, assuming we have full information (i.e. all individuals have been previously run). If we do this against case study, using the final test suites that combine the original events with those from IoT-TEG, we obtain the *fitness histograms* in Figure 4 [9]. The histograms count how many individuals we have within certain ranges of fitness values. These histograms have a number of interesting findings behind them:

- The most striking feature is that the individuals are distributed across very few different fitness values in each case study.

  Part of this is due to the discrete nature of the fitness function, which performs integer arithmetic over an execution matrix whose elements can only be 0 or 1. This also means that given n test cases, we can have up to 2^n different execution rows in the matrix, and therefore only as many different fitness values. Due to the way the DENMEvaP queries were designed, n is 1 for the BruteForce and SnifferCongestion case studies: for each of their mutants, we run the single case study that consists of a large collection of events, and check the complex events that were generated from it. This is the reason for all the individuals being split across two fitness values in the DENMEvaP case studies.

  Terminal Self-Service and Ecological Island are better designed in this regard, with $n = 10$ for the combined original+IoT-TEG test suite. Terminal Self-Service exhibits 8 different fitness values over 6 intervals, which would be enough to guide the search. However, Island only has 3 different fitness values (-1156, -220 and 3620), with most individuals in the worst interval, very few in the middle and the rest at the highest fitness interval. This means that some of the tests in the test suites overlap each other, detecting the same mutants and turning to 1 or 0 at the same time.

[9]We observed very similar results for the smaller test suites. To save space, we omitted their histograms.

S%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - PCAP				
0.15	15.78	1.23	$<$0.01	0.95
0.30	30.57	1.42	$<$0.01	0.98
0.45	45.12	1.21	$<$0.01	0.97
0.60	59.70	1.22	$<$0.01	0.73
0.75	75.03	1.04	$<$0.01	0.91
0.90	89.64	0.70	$<$0.01	0.62
Test suite 2 - JSON				
0.15	15.78	1.23	$<$0.01	0.95
0.30	30.57	1.42	$<$0.01	0.98
0.45	45.12	1.21	$<$0.01	0.97
0.60	59.70	1.22	$<$0.01	0.73
0.75	75.03	1.04	$<$0.01	0.91
0.90	89.64	0.70	$<$0.01	0.62
Test suite 3 - PCAP and JSON				
0.15	15.78	1.23	$<$0.01	0.95
0.30	30.57	1.42	$<$0.01	0.98
0.45	45.12	1.21	$<$0.01	0.97
0.60	59.70	1.22	$<$0.01	0.73
0.75	75.03	1.04	$<$0.01	0.91
0.90	89.64	0.70	$<$0.01	0.62

Table 18 Percentages of all mutants executed to find S% of the strong mutants with guided *EMT* for BruteForce, DENMEvaP (less is better), and p-values for the Mann-Whitney U tests with "GuidedEMT < *EMT*" and "GuidedEMT < random" alternatives.

M%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - PCAP				
0.15	15.17	1.22	$<$0.01	0.02
0.30	29.93	1.47	$<$0.01	0.36
0.45	44.88	1.15	$<$0.01	0.75
0.60	60.30	1.21	$<$0.01	0.27
0.75	74.91	1.11	$<$0.01	0.86
0.90	90.15	0.60	$<$0.01	0.59
Test suite 2 - JSON				
0.15	15.17	1.22	$<$0.01	0.02
0.30	29.93	1.47	$<$0.01	0.36
0.45	44.88	1.15	$<$0.01	0.75
0.60	60.30	1.21	$<$0.01	0.27
0.75	74.91	1.11	$<$0.01	0.86
0.90	90.15	0.60	$<$0.01	0.59
Test suite 3 - PCAP and JSON				
0.15	15.17	1.22	$<$0.01	0.02
0.30	29.93	1.47	$<$0.01	0.36
0.45	44.88	1.15	$<$0.01	0.75
0.60	60.30	1.21	$<$0.01	0.27
0.75	74.91	1.11	$<$0.01	0.86
0.90	90.15	0.60	$<$0.01	0.59

Table 19 Percentages of strong mutants found after executing M% of all mutants with guided *EMT* for BruteForce, DENMEvaP (more is better), and p-values for the Mann-Whitney U tests with "GuidedEMT > *EMT*" and "GuidedEMT > random" alternatives.

- The second observation that we can make is that Island exhibits a different pattern that clearly differs from the others: the strong mutants (the individuals with the highest fitness values) are the minority, rather than the majority, as one would expect from a strongly tested program. The other programs (for which EMT does not perform as well) have a majority of strong mutants.

  This suggests that EMT works especially well after we have done traditional testing to an acceptable degree, to a point in which it is hard to find mutants which we are not killing already and we need a guided process to locate them. If there are more strong mutants to find than killed ones, a simple undirected random search will do better at finding a few scenarios that our test suite is not covering well.

Based on these observations, we make the following suggestions with regards to the improvement and further use of EMT and MuEPL:

- The EMT fitness function should be made more nuanced, by adding more values beyond 0 (surviving) and 1 (killed) to the matrix. This would increase the number of different fitness values for the individuals, and would better guide the search of the genetic algorithm behind EMT.

  Mutation testing theory indicates that in order to kill a mutant, a test case must meet three conditions: reach the mutation, cause a change in internal state (error), and propagate that change to the observable behaviour (causing a test failure). One option to add nuances to the execution matrix is to give a certain score if the mutation is reached, a bit more if there is also a change in internal state, and the maximum score if the killing criterion is achieved.

  Another option would be to define a priority over the test cases, and value some of the test cases more than others. For instance, it would be interesting to see if test cases that cover a harder part to reach of the query should be valued more than others.

- EMT could be extended with a simple meta-heuristic that switches over to random search over the remaining individuals when the percentage of executed mutants has gone above a certain threshold. Given a more nuanced fitness function, another option would be to evaluate simpler local search-based approaches, such as hill climbing, which may do better at traversing exhaustively the large sets of individuals with the same fitness (strong mutants) that we may be looking for.

S%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - PCAP Original Generator				
0.15	18.47	3.60	0.95	1.00
0.30	31.42	5.20	0.52	0.90
0.45	43.93	5.03	0.19	0.86
0.60	57.38	5.51	0.08	0.69
0.75	69.51	5.34	$<$0.01	0.14
0.90	86.45	3.85	0.37	0.73
Test suite 2 - JSON IoT-TEG				
0.15	17.16	2.01	0.02	1.00
0.30	31.04	3.42	0.15	1.00
0.45	46.23	3.37	0.38	0.99
0.60	59.07	3.59	0.02	0.85
0.75	73.83	2.77	0.32	0.98
0.90	89.40	1.76	0.27	0.87
Test suite 3 - Both				
0.15	17.32	2.85	1.00	1.00
0.30	29.67	5.81	0.77	0.80
0.45	42.68	6.63	0.43	0.83
0.60	56.07	6.58	0.24	0.61
0.75	68.80	6.46	0.03	0.15
0.90	85.96	3.91	0.03	0.67

Table 20 Percentages of all mutants executed to find S% of the strong mutants with guided *EMT* for SnifferCongestion, DENMEvaP (less is better), and p-values for the Mann-Whitney U tests with "GuidedEMT < *EMT*" and "GuidedEMT < random" alternatives.

M%	Mean	SD	p-value (EMT)	p-value (Random)
Test suite 1 - PCAP Original Generator				
0.15	15.78	4.47	0.83	0.50
0.30	32.84	6.28	0.13	0.11
0.45	47.75	6.40	0.06	0.32
0.60	63.63	6.25	$<$0.01	0.23
0.75	78.73	4.93	0.04	0.09
0.90	91.96	2.31	$<$0.01	0.06
Test suite 2 - JSON IoT-TEG				
0.15	16.46	2.57	0.06	0.50
0.30	32.79	4.15	$<$0.01	0.08
0.45	47.69	3.45	$<$0.01	0.03
0.60	61.56	3.35	0.02	0.45
0.75	76.05	2.39	0.11	0.55
0.90	90.61	1.83	0.10	0.48
Test suite 3 - Both				
0.15	15.86	4.82	0.80	0.50
0.30	33.43	6.59	0.07	0.05
0.45	46.97	7.66	0.40	0.42
0.60	64.44	7.03	0.01	0.11
0.75	77.88	5.41	0.17	0.16
0.90	91.11	3.08	0.02	0.36

Table 21 Percentages of strong mutants found after executing M% of all mutants with guided *EMT* for SnifferCongestion, DENMEvaP (more is better), and p-values for the Mann-Whitney U tests with "GuidedEMT > *EMT*" and "GuidedEMT > random" alternatives.

- EMT is highly dependent on the number of non-overlapping test cases. For this reason, test suites should ideally be designed in a way that reduces overlaps between tests. If two tests find the exact same bugs (they entirely overlap), they are only increasing test execution times while giving no concrete benefit. Previous studies with EMT have generally minimized test suites by removing the overlaps, but in industrial environments this may not be feasible, and we may only have the option of raising awareness about test overlapping among the developers.
- We observed many surviving mutants within our EPL-based case studies. Some of these may be strong non-equivalent mutants, while others may be equivalent. Further studies are required to evaluate the proportions of equivalent mutants that were generated by the EPL mutation operators, and refine their definitions to reduce them. Equivalent mutants only increase the cost of mutation testing, as they do not help in finding new test cases, and finding if a mutant is equivalent can only be done manually: it is an undecidable problem [BA82].

5 | RELATED WORK

Testing real-time systems is one of the most complex and time-consuming activities [Hea91] and some strategies have been described in the literature to address this problem: from using evolutionary algorithms [WG98, WSJE97, WGG+96] to employing framework based on timed automata [KT09].

Mutation testing can be used for testing software at several levels: unit level, integration level and specification level [JH11c]. Speaking about 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 [MOK06], C [ADH+89], Ada [OVP96]. There are also mutation testing studies which are focused on real-time systems; i.e. R. Nilsson et al. have published works where [NOA04] they propose a model based mutation testing which analyses models of real-time systems to generate test cases for the automated testing of timeliness, in [NOM06] they show a method using heuristic-driven simulation to generate test

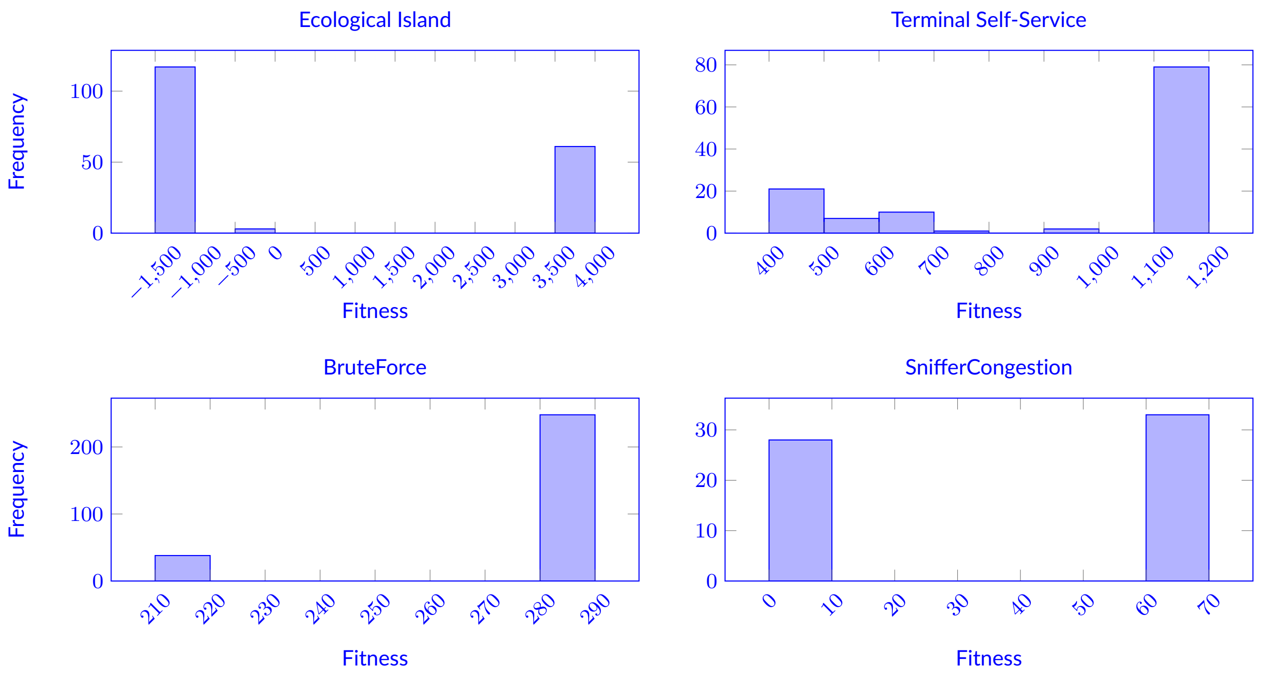

Figure 4 Fitness histograms for the combined original+IoT-TEG test suites

cases (based on the previous model) and in [NO07] they evaluate a mutation-based framework for the automated testing of timeliness by applying it on a small control system.

If testing a real-time system is a time-consuming activity, it has to be taken into account the main drawback of mutation testing [OU01]: the high computational cost involved in the execution of the large number of mutants produced for some programs against their test suites. Nowadays several techniques to alleviate the cost of mutation testing by reducing the number of mutants generated exist [JH11a], such as *selective mutation* [BMV01] which selects a subset of the mutation operators, *mutant sampling* [Bud80] which selects a subset of the mutants randomly, or *high order mutation (HOM)* [JH08] which combines more than a single fault *(change)* into a mutant. *EMT* [DEGM11] was recently proposed to improve the test suite and applied to 3 WS-BPEL compositions. The technique was later extended to generate HOMs [BMGDDJMB11].

EMT was applied to reduce the computational cost involved in mutation testing with WS-BPEL compositions. To carry out this task, *GAmera* was created, a mutation testing system powered by a genetic algorithm [DJEBGDMB09]. After its application to three WS-BPEL compositions, the number of mutants suffered a reduction compared to random pick techniques. Moreover, configuration parameters where established in order to optimise the genetic algorithm results. Given that the *GAmera* genetic algorithm was implemented to be easily used with any programming language, the *EMT* has been also successfully applied to classes implemented using C++. In [DPMBS+17] the same configuration parameters have been used and the results show a representative reduction using this technique than other random pick techniques.

Talking about testing IoT, the majority of works consider that there are no errors in the code, they try to check if the IoT network and devices work properly, and that the data arrives.

For instance, Kim et al. [KAH+18] indicate that the amount of IoT devices and their behaviour cause new challenges to the scalability of traditional software testing, and the heterogeneity of IoT devices increases cost as well as the complexity of coordination testing. As a consequence, the automated IoT testing framework *Service-IoT-TaaS* is introduced. They try to solve the constraints regarding to coordination, cost and scalability issues of traditional software testing in the context of standards-based development of IoT devices. Despite they reach several phases of IoT testing, they consider that the code has no errors. Their efforts are focused on the IoT network and how the messages arrive.

In [WDT+12], they focus their efforts on providing IoT information correctly through standard service interfaces. Wang et al. present a description ontology that integrates the existing efforts for modeling the IoT domain concepts and is extended with essential concepts such as testing to ensure the correct functionality of IoT services at both design and runtime stages.

6 | CONCLUSION AND FUTURE WORK

This work presents how *EMT* can be applied to the Esper EPL programming language. The importance of testing this programming language, which is well suited for the event-driven IoT systems, cannot be understated. Given that many IoT systems live or die on their rapid reaction to events, it is crucial to test that the system can trigger the expected responses on demand.

Mutation testing is the chosen technique to test the Esper EPL programming language, a widely applied technique. The computational cost involved in its execution is its main drawback. In order to address this problem, several techniques have been created, *EMT* being one of them. *EMT* has been successfully applied to classes implemented using C++ and WS-BPEL compositions. The results using *EMT* for those languages were shown to be better than the ones using random selection to find mutants to improve the test suite. Locating these mutants will help not only to improve our test suite, but also to achieve the *EMT* goal: to reduce the involved computational cost. The fact that *EMT* has been used with different programming languages as well as various mutation operators, makes *EMT* a reference technique to improve the test cases in any context. Our proposal to use it with Esper EPL is based on its background.

In order to use *EMT* with Esper EPL, the behaviour of the technique has been analysed using the *GAmera* tool. *GAmera* is a tool which includes a genetic algorithm which applies the *EMT*. The genetic algorithm can be used with different programming languages thanks to the genetic algorithm implementation. A bridge was developed to connect the involved systems: MuEPL and *GAmera*. The bridge was implemented according to both systems characteristics. This means that *GAmera* can be used with any programming language thanks to its adaptability.

The previous connection help us to answer RQ1. The experimental process to carry out the executions to answered the research questions has been explained, as well as the decision of choosing these research questions.

After comparing the results of each case study, RQ1 cannot be clearly answered. Depending on the case study, the results show that sometimes the *EMT* is better than random selection, but not always. As a consequence of the previous results we have improved *EMT*. This modification allows *EMT* to generate more diverse individuals and target operators with a high likelihood of producing strong mutants. This modification was done in the spirit of the guided mutation approach proposed by [ZST05]. We named this improvement *guided EMT*.

The experimental process was performed again using *guided EMT*, the results were more clear now and it can be said that the guided EMT is better than the original EMT not only in the percentage of all mutants executed to find S% of strong mutants, but also the percentage of strong mutants found after executing M% of all mutants, with the exception of one case study. It cannot be said the same after comparing the *guided EMT* against random selection. With the exception of one case study, in the other ones random selection is the best option to find the S% of the strong mutants and in the majority of cases to obtain a high percentage of strong mutants after executing M% of all mutants. This could be because of the source of the events. The events have been generated in order to meet the conditions in the queries, and random selection appears to be the best option to generate the strong mutants. Given that the IoT-TEG generated events have been generated following the original events behaviour, we cannot see too much difference between the test suites. Additionally, the DENMEvaP case study test suite is composed by one test case for each module, and neither the *EMT*nor the *guided EMT* have enough information to generate the strong mutants.

The experiments also helped answer RQ2. The events from the IoT-TEG tool were compared against the original ones in each case study, producing test suites with the same number of test cases and events. The results showed there was no noticeable difference using the original events and the ones generated by IoT-TEG. We can conclude that IoT-TEG can be safely used to replace hand-written generators and captured events for the purposes of mutation testing and EMT.

Given the difference in performance of *EMT* across the various case studies, we performed a simple analysis of their fitness landscapes. We found that the individuals were distributed across a small number of different fitness values, and that there was very little information for *EMT* to search on for the DENMEvaP case studies. We have outlined several ways in which the *EMT* fitness function could be made more nuanced, and the potential benefits of integrating higher-level metaheuristics or alternative search algorithms, which could be better at handling the case where most mutants are strong. Our future work is to implement and evaluate these improvements in traditional and *guided EMT*, reduce the proportion of equivalent mutants generated by the EPL mutation operators, and then re-evaluate both approaches against a wider variety of event-driven programs. Moreover, we want to apply the fitness landscape analysis to the others programming languages in which *EMT* has been implemented: C++ [DPMBS+17] and WS-BPEL [DEGM11].

References

[ADH+89] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of Mutant Operators for the C Programming Language. Technical report SERC-TR-41-P, Purdue University, West Lafayette, Indiana, 1989. Software Engineering Research Center, Purdue University.

[AZY10] Z. Ahmed, M. Zahoor, and I. Younas. Mutation operators for object-oriented systems: A survey. In 2010 The 2nd International

Conference on Computer and Automation Engineering (ICCAE), volume 2, pages 614–618, Feb 2010.

[BA82] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. Acta Informatica, 18(1):31–45, March 1982.

[BMGDDJMB11] Emma Blanco-Muñoz, Antonio García-Domínguez, Juan José Domínguez-Jiménez, and Inmaculada Medina-Bulo. Towards higher-order mutant generation for ws-bpel. In Proceedings of the International Conference on e-Business (ICE-B), 2011, pages 1–6. IEEE, 2011.

[BMV01] Ellen Francine Barbosa, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. Software: Testing Verification and Reliability, 11, 2001.

[Bud80] T. A. Budd. Mutation Analysis of Program Test Data. PhD thesis, Yale University, 1980.

[DEGM11] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing. Information and Software Technology, 53(10):1108–1123, October 2011.

[DJEBGDMB09] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. GAmera: an automatic mutant generation system for WS-BPEL compositions. In Rik Eshuis, Paul Grefen, and George A. Papadopoulos, editors, Proceedings of the 7th IEEE European Conference on Web Services, pages 97–106, Eindhoven, The Netherlands, November 2009. IEEE Computer Society Press.

[DPMBS+17] P. Delgado-Pérez, I. Medina-Bulo, S. Segura, García-Domínguez A., and Domínguez-Jiménez J.J. Gigan: Evolutionary mutation testing for c++ object-oriented systems. In Software Verification and Testing (SAC-SVT 2017), Marrakech, 2017.

[Ecl18] Eclipse Foundation. iot.eclipse.org — Testbeds / Asset Tracking, 2018.

[EN10] Opher Etzion and Peter Niblett. Event Processing in Action. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[Espa] EsperTech. Espertech self-service terminal. http://www.espertech.com/esper/release-5.2.0/esper-reference/html/examples.html#examples-terminalsvc-J2EE.

[Espb] EsperTech. Espertech website. http://www.espertech.com. [Online; accessed 20-december-2016].

[Gad15] Ruediger Gad. Event-driven Principles and Complex Event Processing for Self-adaptive Network Analysis and Surveillance Systems. PhD thesis, University of Applied Sciences Frankfurt am Main, 2015.

[GM17] Lorena Gutiérrez-Madroñal. Generación Automática de Casos en Procesamiento de Eventos con EPL - Authomatic Generation of Cases in Event Processing using EPL. PhD thesis, University of Cadiz, 2017.

[GMDJMB15] L. Gutiérrez-Madroñal, J.J. Dominguez-Jimenez, and I. Medina-Bulo. Mutation testing in event programming language. In 7th European Symposium on Computational Intelligence and Mathematics, ESCIM 2015, 2015.

[GMMBDJ17] L. Gutiérrez-Madroñal, I. Medina-Bulo, and J.J. Domínguez-Jiménez. Iot–teg: Test event generator system. Journal of Systems and Software, 2017.

[GMSZ+12] L. Gutiérrez-Madroñal, H. Shahriar, M. Zulkernine, J.J. Dominguez-Jimenez, and I. Medina-Bulo. Mutation testing of event processing queries. In Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, pages 21–30, 2012.

[Gol89] David E. Goldberg. Genetic algorithms in search, optimization, and machine learning, 1989.

[Hea91] Walter S. Heath. Real-time Software Techniques. Van Nostrand Reinhold Co., New York, NY, USA, 1991.

[HKS08] Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. The internet of things in an enterprise context. Springer, 2008.

[JH08] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on, pages 249–258, Sept 2008.

[JH11a] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on, 37(5):649 –678, October 2011.

[JH11b] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on, 37(5):649–678, September 2011.

[JH11c] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 37(5):649–678, 2011.

[KAH+18] H. Kim, A. Ahmad, J. Hwang, H. Baqa, F. Le Gall, M. A. Reina Ortega, and J. Song. Iot-taas: Towards a prospective iot testing framework. IEEE Access, 6:15480–15493, 2018.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. Form. Methods Syst. Des., 34(3):238–304, June 2009.

[LHZ17] Chao Li, Xiangpei Hu, and Lili Zhang. The iot-based heart disease monitoring system for pervasive healthcare service. Procedia Computer Science, 112:2328–2334, 2017.

[Luc02] D Luckham. The power of events, volume 204. Addison-Wesley Reading, 2002.

[Luc06] D Luckham. A current view of event processing. http://complexevents.com/wp-content/uploads/2006/03/dluckham-workshop-03-2006.pdf, 2006. "[Online; First Workshop on Event Processing; accessed: 2016 02]".

[Mic] Microsoft. Compare Event Grid, routing for IoT Hub.

[Mil] Adrian Milne. Complex Event Processing Made Easy (using Esper). http://www.adrianmilne.com/complex-event-processing-made-easy/. [Online; accessed 9-May-2018].

[MOK06] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for java. In Proceedings of the 28th international conference on Software engineering, pages 827–830. ACM, 2006.

[NO07] Robert Nilsson and Jeff Offutt. Automated testing of timeliness: A case study. In Proceedings of the Second International Workshop on Automation of Software Test, page 11. IEEE Computer Society, 2007.

[NOA04] Robert Nilsson, Jeff Offutt, and Sten F Andler. Mutation-based testing criteria for timeliness. In Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, pages 306–311. IEEE, 2004.

[NOM06] Robert Nilsson, Jeff Offutt, and Jonas Mellin. Test case generation for mutation-based testing of timeliness. Electronic Notes in Theoretical Computer Science, 164(4):97–114, 2006.

[OU01] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the Orthogonal, pages 34–44. Springer US, Boston, MA, 2001.

[OVP96] A Jefferson Offutt, Jeff Voas, and Jeff Payne. Mutation operators for ada. Technical report, Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.

[PA12] Erik Pitzer and Michael Affenzeller. A Comprehensive Survey on Fitness Landscape Analysis. In Recent Advances in Intelligent Engineering Systems, Studies in Computational Intelligence, pages 161–191. Springer, Berlin, Heidelberg, 2012.

[REA96] Helge Rosé, Werner Ebeling, and Torsten Asselmeyer. The density of states — A measure of the difficulty of optimisation problems. In Parallel Problem Solving from Nature — PPSN IV, Lecture Notes in Computer Science, pages 208–217. Springer, Berlin, Heidelberg, September 1996.

[RGOBP+18] D. J. Rosa-Gallardo, G. Ortiz, J. Boubeta-Puig, , and A. García de Prado. Sustainable waste collection (swat): One step towards smart and spotless cities. In Service-Oriented Computing - ICSOC 2017 Workshops, pages 1–12. Ed. Springer International Publishing AG, part of Springer Nature, 2018.

[SRRS07] Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-driven rules for sensing and responding to business situations. In Proceedings of the 2007 inaugural international conference on Distributed event-based systems, pages 198–205. ACM, 2007.

[Thi] ThingSpeak. Thingspeak website. https://thingspeak.com. [Online; accessed 24-january-2017].

[TMV15] Anne-Sophie Tonneau, Nathalie Mitton, and Julien Vandaele. How to choose an experimentation platform for wireless sensor networks? A survey on static and mobile wireless sensor network experimentation facilities. Ad Hoc Networks, 30:115–127, July 2015.

[TSClR07] Javier Tuya, Ma José Suárez-Cabal, and Claudio de la Riva. Mutating database queries. Inf. Softw. Technol., 49(4):398–417, April 2007.

[VFG+13] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Harald Sundmaeker, Markus Eisenhauer, Klaus Moessner, Franck Le Gall, and Philippe Cousin. Internet of Things - Converging Technologies for Smart Environments and Integrated Ecosystems, chapter Internet of Things Strategic Research and Innovation Agenda, pages 7–142. River Publishers, January 2013. ISBN 978-87-92982-73-5.

[VM17] S. R. Vijayalakshmi and S. Muruganand. A survey of internet of things in fire detection and fire industries. In 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), pages 703–707, Feb 2017.

[WDT+12] Wei Wang, Suparna De, Ralf Toenjes, Eike Reetz, and Klaus Moessner. A comprehensive ontology for knowledge representation in the internet of things. In Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on, pages 1793–1798. IEEE, 2012.

[WG98] Joachim Wegener and Matthias Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Syst., 15(3):275–298, November 1998.

[WGG+96] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, Harmen Sthamer, and Bryan Jones. Systematic testing of real-time systems. In 4th International Conference on Software Testing Analysis and Review (EuroSTAR 96), 1996.

[WSJE97] Joachim Wegener, Harmen Sthamer, BryanF. Jones, and DavidE. Eyres. Testing real-time systems using genetic algorithms. Software Quality Journal, 6(2):127–135, 1997.

[ZST05] Qingfu Zhang, Jianyong Sun, and E. Tsang. An evolutionary algorithm with guided mutation for the maximum clique problem. IEEE Transactions on Evolutionary Computation, 9(2):192–200, April 2005.

□