UNIVERSIDAD DE CÁDIZ

Generación Automática de Casos en Procesamiento de Eventos con EPL

Automatic Generation of Cases in Event Processing using EPL

por

Lorena Gutiérrez-Madroñal

Tesis Doctoral con Mención Internacional

en la
Escuela Superior de Ingeniería
Departamento de Ingeniería Informática

4 de febrero de 2017

Conformidad de los Directores

Dª María Inmaculada Medina Bulo, Profesora Titular de Universidad del área de conocimientos de Lenguajes y Sistemas Informáticos perteneciente al Departamento de Ingeniería Informática de la Universidad de Cádiz y D. Juan José Domínguez Jiménez, Profesor Titular de Universidad del área de conocimientos de Lenguajes y Sistemas Informáticos perteneciente al Departamento de Ingeniería Informática de la Universidad de Cádiz, en calidad de Directores de la Tesis titulada *Generación Automática de Casos en Procesamiento De Eventos Con EPL*, realizada por la doctorando Dª Lorena Gutiérrez-Madroñal dentro del Programa de Doctorado en Ingeniería y Arquitectura, para proceder a los trámites conducentes a la presentación y defensa de la Tesis Doctoral arriba indicada, informan que se autoriza la tramitación de la misma.

Los directores de Tesis:

Firmado: ______________________________

Firmado: ______________________________

Fecha: ______________________________

"Si supiese qué es lo que estoy haciendo, no le llamaría investigación, ¿verdad?"

Albert Einstein

Resumen

La aplicación en diversas áreas de Internet de las Cosas (IoT) ha ido en aumento en los últimos años. Uno de los principales inconvenientes que tienen los sistemas IoT es la cantidad de información que tienen que manejar. Esta información llega en forma de eventos, cuyo receptor ha de tomar las decisiones correctas, en tiempo real, según los datos recibidos.

A raíz de estos nuevos sistemas, surgen nuevas herramientas, dispositivos, lenguajes y mecanismos para obtener, procesar y transmitir esta información. Entre estas novedades se destacan los lenguajes de programación “Event Processing Language” (EPL), que se desarrollaron para detectar las situaciones de interés sobre un dominio concreto en tiempo real. Estos lenguajes utilizan patrones para describir las situaciones en las que se quiere filtrar la información, para realizar las actuaciones correctas. Los lenguajes EPL procesan y analizan grandes cantidades de datos (eventos) en tiempo real, por lo que cualquier fallo de programación podría afectar gravemente en la toma de decisiones. Viendo la relevancia que tiene el procesado de esta información, resulta fundamental analizar y probar programas escritos en estos lenguajes de programación, detectando los posibles fallos más comunes que los programadores pueden cometer.

Para poder hacer cualquier tipo de prueba en sistemas que utilicen EPL, se necesita una gran cantidad de eventos con estructuras y valores específicos. Conseguir estos eventos de forma manual puede ser una tarea muy costosa y propensa a errores. Esta tesis aborda el problema de la generación automática de los casos de prueba proponiendo y empleando un método para la generación automática de eventos, el cual se comparará con los sistemas de generación de eventos disponibles. El método incluye una propuesta de especificación para la definición de tipos de eventos y un generador de eventos basado en la especificación anterior. Con este método se obtienen casos de prueba, que van a permitir evaluar situaciones críticas de las aplicaciones, ya que los eventos pueden ser generados con valores concretos que ayuden a simularlas.

Adicionalmente, para validar el método propuesto, se escoge el lenguaje de consulta EPL de la empresa EsperTech y se aplica la prueba de mutaciones en programas que lanzan este tipo de consultas. Entre los motivos de la elección del lenguaje EPL de EsperTech se destaca que es de código abierto y que su sintaxis se aproxima bastante al lenguaje SQL (ampliamente conocido). Por otro lado, se escoge la prueba de mutaciones ya que esta ha sido aplicada con éxito a una gran variedad de lenguajes de programación y mostramos que es una técnica de prueba adecuada para verificar y validar estos programas. Se generan multitud de variantes del programa a probar que representan los fallos más

comunes de programación. Para poder aplicar la prueba de mutaciones se requiere de una gran cantidad de casos de prueba.

Dado que la prueba de mutaciones no había sido aplicada anteriormente a este lenguaje, esta tesis afronta el proceso completo de aplicación de esta técnica a un lenguaje: definición de operadores de mutación, desarrollo de una herramienta de generación y ejecución automática de los mutantes y evaluación de los operadores definidos (comprobando su frecuencia de aplicación en casos de estudio).

Abstract

Internet of Things (IoT) has been increasingly become popular in different areas. One of the main drawbacks of the IoT systems is the amount of information they have to handle. This information arrives as events that need to be processed in real time in order to make correct decisions.

As a consequence, new ways (tools, devices, mechanisms...) of obtaining, processing and transmitting information have to be put into action. It is worth mentioning the "Event Processing Languages" (EPL), which were created to detect, in real time, interesting situations in a particular domain. These languages use patterns to filter the information (a good knowledge of the EPL languages is needed). A huge amount of data is processed and analysed by EPLs, so any programmer error could seriously affect the outcome because of a poor decision making system. Given that processing the data is crucial, testing and analysing programs which run any EPL language is required. The most common mistakes that programmers could make have to be detected.

A large number of events with specific values and structures are needed to apply any kind of testing in programs which use EPL. As this is a very hard task and very prone to error if done by hand, this dissertation addresses the automated generation of test cases; a method to automatically generate events has been proposed and used. Moreover, it will be compared to existing event generation systems. This method includes a general definition of what is a event type and its representation is proposed. Additionally an event generator is developed based on this definition. Test cases to evaluate critical situations the EPL programs may suffer are obtained following this method. The proposed method allows the generation of events with specific values and structures, which will simulate these critical situations.

In order to validate the proposed method, the EPL of EsperTech company is selected to apply mutation testing to programs which run the EsperTech EPL queries. EsperTech EPL language has been selected because it is open source and its syntax is very similar to SQL language (very popular programming language). Mutation testing is the chosen testing technique because it has been successfully applied to various programming languages. Mutation testing is a suitable testing technique to verify and validate these programs. Several variations of the original program are generated, which include "typical" programming errors. In order to apply mutation testing, a huge amount of data is required.

Due to mutation testing has not been applied to EsperTech EPL language before, this dissertation faces the complete process of applying the mutation testing technique to

a programming language: mutation operators definition, a development of a tool which automatically generates and executes mutants and evaluation of the defined operators (checking their application frequency in studies cases).

Agradecimientos

"Cuando bebas agua, recuerda la fuente."

(Proverbio chino)

Desde estas líneas pretendo expresar mi más sincero agradecimiento a todas aquellas personas que durante estos años de trabajo han estado a mi lado, mi pareja, amigos, familia y compañeros que, de una u otra forma, han contribuido a que esta tesis haya llegado a buen fin.

- A Jose, por su apoyo y ayuda incondicional, por su afecto y ánimos, por las horas que te he robado... por todo, gracias.
- A mi padre por su apoyo y confianza, por escuchar e intentar entender lo que hace su hija, por desesperarse cuando yo me desesperaba, y ayudarme y animarme cuando más lo necesitaba.
- A mi madre por su apoyo, sus ánimos y por estar ahí (aunque fueran 5 minutos tras el teléfono o me hiciera una visita exprés).
- A mis tutores Inma y Juanjo, por su apoyo, dedicación y buenos consejos para que esta tesis pudiese llegar a buen puerto.
- A los miembros del grupo UCASE por sus ánimos y apoyo, en especial a Antonio y a Juan por su paciencia, dedicación y explicaciones (a pesar de estar a miles de kilómetros en más de una ocasión), a Pedro por su ayuda y escucha.
- A Ruediger por los más de 100 emails intercambiados, por sus detalladas contestaciones y por sus ánimos.
- A Hossain Shahriar y Mohammad Zulkernine por darme la oportunidad de realizar la estancia de investigación en Kingston, Canadá, y a todos los amigos que conocí allí de los que conservo muy buenos recuerdos.
- A Mercedes y Manolo por darme la oportunidad de realizar la estancia de investigación en Madrid, por ayudarme, por contar conmigo y por todos los buenos momentos pasados (y que pasarán).
- A mis compañeros del Departamento de Ingeniería Informática y los compañeros de la Facultad de Ciencias Sociales y de la Comunicación por sus ánimos, en especial a Manolo por sus consejos, por ser un gran apoyo y compañero y por contar siempre conmigo; a Antonio, Nuria y un largo etcétera donde incluyo aquellos que siempre me han dedicado su tiempo a escucharme e interesarse.

- A mis amigos, por escucharme y no pensar que estaba loca cuando les decía que estaba matando mutantes, por organizarme una fiesta donde "tenía que matar mutantes", por sus ánimos constantes, y por sacarme una sonrisa cuando me decían que era "famosa" por dar ponencias, salir en artículos... ¡os quiero!
- A todos aquellos (alumnos, compañeros, familiares, amigos y un largo etcétera) que me han escuchado decir "cuando termine la tesis..." gracias por vuestra paciencia y comprensión... ahora ya tengo hueco.

Muchas gracias a todos.

Agradecimientos Institucionales

Este trabajo ha sido financiado por el proyecto DArDOS TIN2015-65845-C3-3-R del Programa Nacional de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad.

Índice general

Conformidad II

Resumen VI

Abstract VIII

Agradecimientos X

Agradecimientos Institucionales XII

Índice de Figuras XIX

Índice de Tablas XXII

1. Introdución 1
- 1.1. Motivación 1
  - 1.1.1. Objetivos 4
- 1.2. Aportaciones 6
  - 1.2.1. Estructura de la Tesis 7

2. Fundamentos 11
- 2.1. Internet de las Cosas 11
- 2.2. Arquitecturas dirigidas por eventos 15
- 2.3. Procesamiento de eventos complejos 17
  - 2.3.1. Conceptos Fundamentales 17
  - 2.3.2. Lenguajes de Procesamiento de Eventos 22
- 2.4. Lenguaje EPL de EsperTech 23
  - 2.4.1. Sintaxis y cláusulas de EPL de EsperTech 24
  - 2.4.2. Patrones en EPL de EsperTech 38
- 2.5. Generadores de eventos y plataformas IoT 45
- 2.6. Técnicas de verificación y validación del software 47
  - 2.6.1. Prueba de software 50
- 2.7. Prueba de Mutaciones 53
  - 2.7.1. Técnicas de reducción de coste 55
  - 2.7.2. Técnicas de detección de mutantes equivalentes 57

3. Estado del arte **59**
3.1. Pruebas en sistemas IoT 59
3.2. Procesado de Eventos Complejos (CEP) 64
3.3. Pruebas en aplicaciones CEP 66
3.4. Expansión de los lenguajes de procesamiento de eventos 69
3.4.1. EPL orientados a flujos 69
3.4.2. EPL orientados a reglas 70
3.4.3. EPL imperativos 71
3.5. Generadores de eventos y plataformas IoT 72
3.6. Herramientas de validación de consultas EPL de EsperTech 75
3.7. Definición de operadores de mutación 77
3.8. Prueba de mutaciones en sistemas de tiempo real 79

4. Método para automatizar la generación de eventos para pruebas **81**
4.1. Motivación 81
4.2. Etapas del método 82
4.3. Conclusiones 83

5. Propuesta de especificación para la definición de tipos de eventos **85**
5.1. Estructura de la especificación 85
5.1.1. Bloque 87
5.1.2. Campo y Campos Opcionales 88
5.2. Tipos de Datos 91
5.2.1. Tipos Simples 92
5.2.1.1. Tipo Entero 93
5.2.1.2. Tipo Punto Flotante 94
5.2.1.3. Tipo Entero Grande 94
5.2.1.4. Tipo Cadena de Caracteres 95
5.2.1.5. Tipo Alfanumérico 96
5.2.1.6. Tipo Booleano 96
5.2.1.7. Tipo Fecha 97
5.2.1.8. Tipo Tiempo 98
5.2.2. Tipos Complejos 99
5.2.2.1. Tipos Anidados 100
5.2.2.2. Atributos opcionales para tipos complejos 103
5.3. Conclusiones 106

6. Generador de eventos IoT-TEG **109**
6.1. Introducción 109
6.2. Arquitectura de IoT-TEG 112
6.2.1. Componente de validación 112
6.2.2. Componente de generación 119
6.2.2.1. Tipos de datos simples 122
6.2.2.2. Tipos de datos complejos 129
6.3. Funcionalidad avanzada de lectura de consultas 134
6.3.1. Secuencia del proceso 136
6.3.2. Tratamiento de datos 140

6.3.2.1. Entero, Punto flotante y Entero grande 141
6.3.2.2. Cadenas de caracteres y Alfanuméricos 142
6.3.2.3. Booleanos 143
6.3.2.4. Fecha y Tiempo 143
6.4. Aspectos técnicos de la implementación del generador IoT-TEG 144
6.5. Conclusiones 145

7. Definición de operadores de mutación para EPL de EsperTech 149
7.1. Introducción 149
7.2. Operadores de expresiones de patrones 152
7.3. Operadores de reemplazo 156
7.4. Operadores de ventanas 163
7.5. Operadores de inyecciones de ataque de SQL 165
7.6. Conclusiones 165

8. Generador de mutantes MuEPL 167
8.1. Introducción 167
8.2. Arquitectura de MuEPL 168
8.2.1. Capturador 169
8.2.2. Analizador 171
8.2.3. Generador de mutantes 172
8.2.4. Sistema de ejecución 177
8.3. Criterios para matar mutantes 179
8.4. Conclusiones 180

9. Resultados 183
9.1. Aplicación de la especificación para definir tipos de eventos con tipos de eventos reales 184
9.2. Selección del criterio para matar mutantes 185
9.2.1. Tiempo de ejecución 186
9.2.2. Contenido y orden de los eventos 187
9.2.3. Contenido y número de eventos 189
9.3. Comparativas de ejecución 190
9.3.1. Caso de estudio - Self-Service Terminal 190
9.3.2. Caso de estudio - Transaction 192
9.3.3. Caso de estudio - Ecological Island 195
9.3.4. Caso de estudio - Domótica 198
9.3.5. Caso de estudio - DENMEvaP 200
9.3.5.1. Módulo - Brute Force 202
9.3.5.2. Módulo - Sniffer Flat 204
9.3.5.3. Módulo - Sniffer Flood 207
9.3.5.4. Módulo - Simple Sniffer 209
9.3.5.5. Módulo - Sniffer Congestion 211
9.4. Análisis de operadores de mutación definidos para EPL de EsperTech . . . 212
9.4.1. Análisis de mutantes potencialmente equivalentes 213
9.5. Reducción del tiempo de ejecución 216
9.6. Usabilidad del generador de eventos IoT-TEG 217

9.7. Conclusiones 219

10. Conclusiones y Trabajo Futuro **222**
10.1. Contribuciones y Conclusiones 222
10.1.1. Estado del Arte 222
10.1.2. Proponer un método para generar de forma automática eventos para pruebas 224
10.1.3. Proponer una especificación para la definición de los tipos de eventos 224
10.1.4. Implementar una herramienta para generar casos de prueba para lenguajes procesadores de eventos 224
10.1.5. Definir un conjunto de operadores de mutación adaptados a las características del lenguaje EPL de EsperTech 225
10.1.6. Implementación de una herramienta de generación y ejecución de mutantes EPL de EsperTech 226
10.1.7. Evaluar los resultados con diferentes casos de estudio 227
10.2. Líneas de Investigación Futuras 229
10.2.1. Ampliar la funcionalidad de lectura de consultas 229
10.2.2. Añadir la generación secuencial en IoT-TEG 229
10.2.3. Generación de varios tipos de eventos en IoT-TEG 230
10.2.4. Añadir la generación condicional en IoT-TEG 230
10.2.5. Generación de eventos distribuidos en el tiempo 230
10.2.6. Desarrollar un componente de validación de la salida 231
10.2.7. Conexión de MuEPL con la herramienta GAmera 231
10.2.8. Mejorar la definición de algunos operadores de mutación 231
10.2.9. Mejorar el componente Capturador de MuEPL 232
10.3. Publicaciones 232
10.3.1. Publicaciones Internacionales 232
10.3.2. Publicaciones Nacionales 233

11. Conclusions and Future Work **234**
11.1. Contributions and Conclusions 234
11.1.1. State of the art 234
11.1.2. Proposing a method to automatically generate test events 235
11.1.3. Proposed specification to define event types 236
11.1.4. Tool implementation to generate event processing languages test cases 236
11.1.5. Defining a mutation operator set adapted to EPL characteristics . 237
11.1.6. EPL mutants generation and execution tool implementation 238
11.1.7. Evaluation of case studies results 238
11.2. Future Research Lines 239
11.2.1. Extending the queries reading functionality 239
11.2.2. Adding a sequential generation 240
11.2.3. Adding multiple event type generations 240
11.2.4. Adding the conditional generation 240
11.2.5. Time distribution between events generation 241
11.2.6. Output validation component implementation 241
11.2.7. MuEPL and GAmera connection 241
11.2.8. Improving some mutation operator definitions 241

11.2.9. Improving the Capturing component 242
11.3. Publications 242
11.3.1. International Publications 242
11.3.2. National Publications 243

A. XML Schema de la Propuesta de Especificación para la Definición de Tipos de Eventos 244
A.1. XML Schema 244

B. Casos de Estudio 250
B.1. Caso de Estudio - Self-Service Terminal 250
B.2. Caso de Estudio - Transaction 252
B.3. Caso de Estudio - Ecological Island 253
B.4. Caso de Estudio - Domótica 254
B.5. Caso de Estudio - DENMEvaP 256

C. Definición de Eventos de los Casos de Estudio 260
C.1. Caso de Estudio - Terminal 260
C.2. Caso de Estudio - Transaction 261
C.3. Caso de Estudio - Ecological Island 262
C.4. Caso de Estudio - Domotic 264
C.5. Caso de Estudio - DENMEvaP 265
C.5.1. Módulo - Brute Force 265
C.5.2. Módulo - Sniffer Flat 270
C.5.3. Módulo - Sniffer Flood 275
C.5.4. Módulo - Simple Sniffer 280
C.5.5. Módulo - Sniffer Congestion 285

Bibliografía 290

Índice de figuras

2.1. Ejemplo de tres flujos de CEP . . . 20
2.2. Etapas implicadas en CEP . . . 21
2.3. Salida de una declaración con una ventana de tiempo . . . 27
2.4. Salida de una declaración con una ventana de tiempo . . . 29
2.5. Declaraciones con un filtro de cadena de eventos . . . 30
2.6. Declaraciones con la cláusula where . . . 32
2.7. Los ocho componentes de una plataforma IoT . . . 47

4.1. Etapas del método . . . 82

5.1. Componentes y propiedades de la definición de tipo de evento . . . 87

6.1. Arquitectura del generador de eventos IoT-TEG . . . 113

7.1. Ejemplo PFP . . . 154
7.2. Ejemplo PGR . . . 154
7.3. Ejemplo PNR . . . 155
7.4. Ejemplo POC . . . 155
7.5. Ejemplo POM . . . 155
7.6. Ejemplo POM . . . 156
7.7. Ejemplo de mutante generado por RRO . . . 156
7.8. Ejemplo de mutante generado por RLO . . . 157
7.9. Ejemplo de mutante generado por RAF . . . 157
7.10. Ejemplo de mutante generado por RLM, RLA y RBW . . . 158
7.11. Ejemplo de mutante generado por RNO . . . 159
7.12. Ejemplo de mutante generado por RRR_2 . . . 159
7.13. Ejemplo de mutante generado por RSR_1 . . . 160
7.14. Ejemplo de mutante generado por RGR . . . 160
7.15. Ejemplo de mutante generado por RBR . . . 161
7.16. Ejemplo de mutante generado por RJR . . . 161
7.17. Ejemplo de mutante generado por RSC . . . 162
7.18. Ejemplo de mutante generado por RNW . . . 162
7.19. Ejemplo de mutante generado por ROM . . . 163
7.20. Ejemplo de mutante generado por RTU . . . 163
7.21. Ejemplo de mutante generado por WLM . . . 164
7.22. Ejemplo de mutante generado por WBT . . . 164
7.23. Ejemplo de mutante generado por IWR . . . 165
7.24. Ejemplo de mutante generado por ICN . . . 165

8.1. Arquitectura MuEPL 168
8.2. Consultas capturadas de Terminalsvc-jse 170
8.3. Consultas de Terminalsvc-jse y operadores localizados 171
8.4. Salida del analizador para Terminalsvc-jse 172
8.5. Codificación de un mutante 172
8.6. Mutante WBL de Terminalsvc-jse 174
8.7. Mecanismo de ejecución en paralelo 178

Índice de tablas

2.1. Funciones de agregación que incorpora EPL de EsperTech. 31
2.2. Operadores aritméticos. 35
2.3. Operadores lógicos y de comparación. 36
2.4. Patrones atómicos. 38
2.5. Patrones de operadores. 39
2.6. Operadores de patrones. 42
2.7. Controladores de patrones. 43
2.8. Observadores de patrones. 44

5.1. Patrones para los tipos Fecha y Tiempo. 98

7.1. Operadores de Mutación para EPL de EsperTech adaptados de SQL. . . . 150
7.2. Operadores de Mutación para EPL de EsperTech sin modificar de SQL. . 150
7.3. Operadores de Mutación para EPL de EsperTech de patrones y de reemplazamiento. 153
7.4. Operadores de Mutación para EPL de EsperTech de ventanas y de inyecciones de ataque de SQL. 154

8.1. Valores y atributos para los operadores de mutación. 173

9.1. Comparativas de los tiempos de ejecución. 186
9.2. Resultados tras aplicar el *analizador* con el caso de estudio Terminal. . . . 191
9.3. Resultados del caso de estudio Self-Service Terminal usando sus eventos originales. 192
9.4. Resultados del caso de estudio Self-Service Terminal usando eventos generados por IoT-TEG. 192
9.5. Resultados tras aplicar el *analizador* con el caso de estudio Transaction. . 193
9.6. Resultados del caso de estudio Transaction usando sus eventos originales, una transacción. 194
9.7. Resultados del caso de estudio Transaction usando sus eventos originales, diez transacciones. 194
9.8. Resultados del caso de estudio Transaction usando eventos generados por IoT-TEG, una transacción. 195
9.9. Resultados del caso de estudio Transaction usando eventos generados por IoT-TEG, diez transacciones. 195
9.10. Resultados tras aplicar el *analizador* con el caso de estudio Ecological Island. 196
9.11. Resultados del caso de estudio Ecological Island usando sus eventos originales. 197

9.12. Resultados del caso de estudio Ecological Island usando eventos generados por IoT-TEG. 197
9.13. Resultados tras aplicar el *analizador* con el caso de estudio Domótica. . . 199
9.14. Resultados del caso de estudio Domótica usando sus eventos originales. . . 199
9.15. Resultados del caso de estudio Domótica usando eventos generados por IoT-TEG. 200
9.16. Resultados tras aplicar el *analizador* con el caso de estudio Brute Force. . 202
9.17. Resultados del caso de estudio Brute Force usando sus eventos originales. 202
9.18. Resultados del caso de estudio Brute Force usando eventos generados por IoT-TEG. 203
9.19. Resultados del caso de estudio Brute Force usando eventos generados por IoT-TEG y la funcionalidad de lectura de consultas. 204
9.20. Resultados tras aplicar el *analizador* con el caso de estudio Sniffer Flat. . 204
9.21. Resultados del caso de estudio Sniffer Flat usando sus eventos originales. . 205
9.22. Resultados del caso de estudio Sniffer Flat usando eventos generados por IoT-TEG. 205
9.23. Resultados del caso de estudio Sniffer Flat usando eventos generados por IoT-TEG y la funcionalidad de lectura de consultas. 206
9.24. Resultados tras aplicar el *analizador* con el caso de estudio Sniffer Flood. 207
9.25. Resultados del caso de estudio Sniffer Flood usando sus eventos originales. 207
9.26. Resultados del caso de estudio Sniffer Flood usando eventos generados por IoT-TEG. 208
9.27. Resultados del caso de estudio Sniffer Flood usando eventos generados por IoT-TEG aplicando la funcionalidad avanzada. 209
9.28. Resultados tras aplicar el *analizador* con el caso de estudio Simple Sniffer. 209
9.29. Resultados del caso de estudio Simple Sniffer usando sus eventos originales. 210
9.30. Resultados del caso de estudio Simple Sniffer usando eventos generados por IoT-TEG. 210
9.31. Resultados tras aplicar el *analizador* con el caso de estudio Sniffer Congestion. 211
9.32. Resultados del caso de estudio Sniffer Congestion usando sus eventos originales. 211
9.33. Resultados del caso de estudio Sniffer Congestion usando eventos generados por IoT-TEG. 212
9.34. Análisis del tipo de mutantes generados en cada caso de estudio según el operador de mutación. 214
9.35. Resultados de las ejecuciones del módulo Sniffer Congestion usando y sin usar la paralelización. 217

B.1. Eventos para el caso de estudio *Self-Service Terminal*. 251
B.2. Eventos para el caso de estudio *Transaction 3-Event Challenge*. 252
B.3. Información sobre los atributos del tipo de evento utilizado en el caso de estudio de EcologicalIsland. 254
B.4. Información sobre los canales utilizados como fuentes de eventos en el caso de estudio de Domótica. 255
B.5. Información sobre los atributos de los tipos de eventos utilizados en el caso de estudio de Domótica. 256
B.6. Información sobre los atributos del tipo de evento "Sniffer_flat". 258

Capítulo 1

Introdución

Este capítulo describe los antecedentes y la motivación encontrada para la realización de esta tesis doctoral. A continuación se detallan los objetivos que se persiguen y las contribuciones que se alcanzan con su desarrollo. Por último se realiza una descripción de la estructura del resto de esta memoria.

1.1. Motivación

Hoy en día fluyen diariamente enormes volúmenes de datos por una gran cantidad de sistemas de información heterogéneos y distribuidos a nivel mundial [1]. Si nos centramos en el entorno empresarial, los expertos del negocio tendrán que obtener y gestionar eficientemente toda esta información obtenida en tiempo real, para dirigir decisiones y/o acciones de forma satisfactoria [2]. Dado que las herramientas de software tradicionales no pueden procesar esta gran cantidad de datos, *big data* es un enfoque emergente que permite gestionar y analizar esta vasta cantidad de datos, convirtiéndolos en información valiosa para la toma de decisiones.

No obstante, big data se centra fundamentalmente en datos ya obtenidos y almacenados en bases de datos. Luego para el procesamiento de datos de diversa naturaleza y provenientes de fuentes heterogéneas en tiempo real, puede no ser la mejor solución. En 2008 Haller et al. [3] definieron IoT (*Internet of Things*) como:

> "A world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes. Services are available to interact with these 'smart objects' over the Internet, query their state and any information associated with them, taking into account security and privacy issues."

Extrayendo de la definición, IoT es considerado como un mundo donde los objetos físicos que se integran en la red se convierten en participantes activos en los procesos de negocios. En este mundo se maneja, en tiempo real, un gran volumen de información que llega de diversas fuentes en forma de mensajes o eventos. Como consecuencia se desarrollan diversas herramientas, dispositivos y mecanismos para obtener, procesar y transmitir esta información.

Uno de los mayores problemas de estos sistemas es monitorizar la información y responder con la menor latencia. Para intentar solventarlo David Luckham propone el *Procesado de Eventos Complejos* (CEP) [4]. Esta tecnología permite detectar y responder ante situaciones relevantes o críticas en el negocio ya que procesa y analiza grandes cantidades de eventos correlacionándolos en tiempo real. Para ello, las condiciones que describen las situaciones que se quieren detectar se especifican usando *patrones de eventos*. Los patrones de eventos se añaden a un motor CEP incorporado en el software, que se encargará de analizar y correlacionar los eventos provenientes de las distintas fuentes de información, así como generar los eventos complejos tras la detección de estos patrones indicando qué situación o situaciones han acontecido. Es decir, que tras ser detectados, se generarán nuevos eventos más complejos y con mayor significado semántico, que ayudarán a la toma de decisiones.

Estos patrones de eventos son definidos utilizando lenguajes específicos de procesado de eventos desarrollados para este propósito, conocidos como EPL *(Event Processing Language)*. Para proceder a la definición de dichos patrones se requiere un gran conocimiento del lenguaje EPL en cuestión que se vaya a emplear. Ya que cualquier fallo en su definición podría no detectar una situación crítica en el negocio.

La aparición de nuevos lenguajes de programación que procesan y analizan grandes cantidades de datos en tiempo real y en los que cualquier fallo de programación podría provocar fallos graves en la toma de decisiones, requieren ser validados y verificados para que su funcionamiento sea el correcto. Para realizar pruebas en programas que emplean EPL, se necesita desarrollar un método que permita generar una gran variedad de datos de entrada para este tipo de software, eventos. Los primeros generadores de eventos que se desarrollaron no estaban enfocados al mundo IoT, sino para realizar experimentos para la simulación de eventos físicos. Las plataformas IoT aparecen por la necesidad de información por parte de las personas y empresas para una correcta toma de decisiones y para mejorar la calidad de nuestras vidas. Estas plataformas permiten al usuario elaborar, desarrollar, almacenar y analizar aplicaciones IoT. Todos los dispositivos conectados a estas plataformas pueden publicar, en los canales de las mismas, los eventos que captan del mundo real, permitiendo que cualquier usuario pueda utilizarlos en su aplicación. El problema aparece cuando se quieren usar estos eventos para realizar pruebas específicas;

es posible que sus valores e incluso su estructura, necesiten ser modificados de forma manual.

El método que se propone en la presente tesis, permite generar de forma automática eventos de prueba. Para validar los eventos generados, se acude a la literatura para seleccionar el tipo de prueba adecuada en la que se necesite una gran diversidad de eventos de prueba.

Según la guía SWEBOK 2014 [5]:

> "The purpose of V&V (Verification and Validation) is to help the development organization build quality into the system during the life cycle. V&V processes provide an objective assessment of products and processes throughout the life cycle. This assessment demonstrates whether the requirements are correct, complete, accurate, consistent, and testable. The V&V processes determine whether the development products of a given activity conform to the requirements of that activity and whether the product satisfies its intended use and user needs."

En resumen se indica que la verificación y validación (de software) persigue la calidad de un producto e indica si este cumple las necesidades y requisitos. La verificación y validación del software engloba técnicas estáticas y dinámicas. A diferencia de las técnicas estáticas, las técnicas dinámicas implican la ejecución del sistema a probar. Dentro de las técnicas dinámicas podemos destacar la *prueba del software*. La prueba de software es el proceso que permite evaluar la funcionalidad y corrección de un programa mediante su ejecución sobre un conjunto finito de casos de prueba. En 1972 Dijkstra [6] expuso:

> "Program testing can be used to show the presence of bugs, but never their absence."

Es decir, que la prueba de software únicamente alerta de la presencia de fallos, pero no puede demostrar su ausencia. Por otro lado en [7] se matiza que, a pesar de las inherentes limitaciones de la prueba de software, si se aplican las técnicas de prueba adecuadas de forma rigurosa y consistente estas pueden mejorar la calidad de los sistemas software.

Un aspecto fundamental de la prueba de software es la selección de los casos de prueba a utilizar en la prueba de un programa. Dado que el dominio de entrada de un programa puede ser muy grande, se necesita seleccionar un pequeño conjunto de casos de prueba representativo del dominio de entrada completo. Para ello distinguimos diferentes criterios: *métodos estructurales*, que generan casos de prueba que deben cubrir ciertos

elementos de la estructura del programa; los *métodos basados en fallos*, que miden la capacidad de un conjunto de casos de prueba para encontrar fallos en un programa, y los *métodos basados en errores*, que generan casos de prueba que comprueban el programa en ciertos puntos más propensos a errores. La *prueba de mutaciones* es la técnica más conocida dentro de los métodos basados en fallos. Esta técnica ha sido aplicada a multitud de lenguajes de programación {C, Fortran, Ada, C++, Java, SQL, ...}. A pesar de sus conocidas limitaciones [8] la prueba de mutaciones está aceptada como una de las técnicas más populares de prueba de software.

Por todo lo anteriormente expuesto, la motivación que guía la realización de esta tesis doctoral es:

- El auge de sistemas que manejan una masiva cantidad de información de la cual se espera poder tomar decisiones acertadas de manera inmediata, hace necesario disponer de un método para generar datos de prueba (eventos) para este tipo de aplicaciones.
- Para validar los eventos de prueba generados por el método propuesto es necesaria la aplicación de técnicas de prueba en el lenguaje de programación específico EPL. En concreto se aplicará la prueba de mutaciones en el EPL de la empresa EsperTech [9], un lenguaje de código abierto que se ejecuta en un motor CEP escrito en Java.

1.1.1. Objetivos

Esta tesis doctoral satisface los siguientes objetivos:

- **Objetivo 1**: Recopilar el estado del arte de los estudios sobre el Internet de las Cosas (IoT), el procesamiento de eventos complejos (CEP) y las pruebas aplicadas, la expansión de los lenguajes de procesamiento de eventos, los generadores de eventos existentes, la definición de operadores de mutación y su integración en herramientas de mutación, la aplicación de la prueba de mutaciones en sistemas de tiempo real y trabajos relacionados.

  Para alcanzar este objetivo, se realizará una exhaustiva revisión bibliográfica sobre dichos aspectos.
- **Objetivo 2**: Proponer un método para generar de forma automática eventos para pruebas. Debido a los problemas a los que se enfrentan los programadores a la hora de realizar pruebas en aplicaciones que procesan eventos, se propone un método

que permita generar eventos adaptados al programa y las pruebas que se vayan a realizar obteniendo eventos con la estructura y valores específicos que se requieran.

Este método se compone de tres etapas que se definen en los dos siguientes objetivos: una especificación para la definición de los tipos de eventos y una herramienta que valide el tipo de evento definido y genere los eventos (casos de prueba para las aplicaciones que procesan eventos).

- **Objetivo 2.1**: Proponer una especificación para la definición de los tipos de eventos. Dado que la naturaleza de los eventos es muy diversa, es necesario establecer unas normas para determinar cómo son los eventos, su estructura, tipo y posibles formatos.

  Se estudiarán los diferentes estándares propuestos para la tecnología CEP y se analizará una amplia muestra de tipos de eventos de diversa naturaleza. Tras observar las posibles estructuras y los posibles tipos de datos que pueden tener los eventos, se definirán las normas a seguir para la definición del tipo de evento a través de la especificación que se propone.

- **Objetivo 2.2**: Implementar una herramienta que valide la definición del tipo de evento y que a partir de esta genere de forma automática de casos de prueba para lenguajes procesadores de eventos. Los casos de prueba de este tipo de programas son los propios eventos. Debido a que existe una gran variedad de fuentes desde las que obtenemos los eventos: bases de datos, servicios, procesos de negocios, sensores, sistemas de mensajería intermedio... se estudiarán sus salidas (los formatos y los tipos de eventos) para crear un generador de eventos de prueba válido para cualquier aplicación que procese eventos. Este generador de eventos estará basado en la especificación que se propondrá en la presente tesis.

  Utilizando el lenguaje de programación Java se implementará una herramienta basada en la especificación propuesta para la definición de tipos de eventos, que incluirá un componente de validación para comprobar que la definición del tipo de evento sigue la especificación. Una vez analizada la estructura y los tipos de datos que definen el evento, se generarán tantos eventos como indique el usuario. La herramienta ofrecerá la posibilidad de generar los eventos en los formatos más comunes utilizados por los programas que procesan eventos.

- **Objetivo 3**: Evaluar el método propuesto para la generación automática de eventos de prueba, aplicando la prueba de mutaciones. Para aplicar el proceso completo de esta técnica de prueba en EPL de EsperTech, se han de seguir los puntos que se definen en los dos siguientes objetivos.

- **Objetivo 3.1**: Definir un conjunto de operadores de mutación adaptados al lenguaje EPL de EsperTech que haga posible aplicar la prueba de mutaciones en este lenguaje de programación. El conjunto de operadores ha de ser representativo para EPL de EsperTech, es decir, que los operadores de mutación definidos cubran los fallos más comunes que un programador puede cometer con este lenguaje.

  Para ello se analizará una amplia muestra de programas que usan consultas EPL de EsperTech. Debido a la similitud entre los lenguajes de programación EPL de EsperTech y SQL, se estudiarán los operadores de mutación definidos para SQL adaptándolos o redefiniéndolos para el lenguaje EPL de EsperTech. Además, tras estudiar la sintaxis de EPL de EsperTech y los programas que utilizan consultas de este lenguaje, se definirán nuevos operadores de mutación específicos de EPL de EsperTech para cubrir los posibles errores que podría cometer un programador. A partir de los operadores se incluirán errores en los programas a probar.

- **Objetivo 3.2**: Implementar una herramienta para la generación y ejecución de mutantes de EPL de EsperTech.

  Para alcanzar este objetivo se desarrollará una herramienta, con el lenguaje de programación Java, que contenga unos componentes adecuados para aplicar la prueba de mutaciones: analizador, generador de mutantes, sistema de ejecución.

- **Objetivo 4**: Evaluar los resultados del método propuesto.

  Los tipos de eventos de diferentes casos de estudio serán analizados y definidos siguiendo la especificación propuesta en la presente tesis. Estos se generarán con el generador de eventos de prueba y podrán ser comparados con los originales y así validar la herramienta. Además, se generarán eventos de prueba para programas en los que se aplicará la prueba de mutaciones, convirtiéndose en casos de prueba con valores específicos que ayudarán no solo a validar el método, sino también en confirmar la necesidad del mismo.

1.2. Aportaciones

En esta sección, se enumeran las principales aportaciones derivadas del trabajo desarrollado en esta tesis doctoral:

1. Proposición de un método para la generación automática de eventos de prueba.
2. Proposición de una especificación para la definición de tipos de eventos de diversa naturaleza.

3. Implementación de un generador de eventos basado en la especificación anterior y aplicado a varios casos de estudio de diversa índole. En este se incluye una funcionalidad que permite generar eventos con valores específicos según las consultas EPL de EsperTech que van a procesar dichos eventos.
4. Definición de un conjunto de operadores de mutación adaptados a las características del lenguaje EPL de EsperTech.
5. Evaluación del conjunto de operadores de mutación, para determinar la necesidad de su definición, según su frecuencia de aplicación.
6. Implementación de una herramienta de generación y ejecución de mutantes EPL de EsperTech.
7. Evaluación del método propuesto con diversos tipos de eventos.
8. Evaluación de las diferentes funcionalidades del generador de eventos.

1.2.1. Estructura de la Tesis

A continuación se describe brevemente el contenido del resto de los capítulos que componen esta memoria.

- En este capítulo 1 se especifica la motivación de este trabajo, se enumeran los principales objetivos de esta tesis, así como las principales contribuciones.
- El capítulo 2 presenta una recopilación de los conceptos fundamentales sobre todas las tecnologías y técnicas utilizadas tales como: Internet de las Cosas (IoT), arquitecturas dirigidas por eventos, procesamiento de eventos complejos (CEP), los lenguajes de procesamiento de eventos (en detalle sobre el lenguaje EPL de EsperTech), los generadores de eventos y plataformas IoT y técnicas de verificación y validación del software, en concreto la prueba de mutaciones; además se detallan los principales trabajos relacionados.
- En el capítulo 3 se presentan estudios relacionados con los diferentes aspectos que se abordan en esta tesis doctoral: estudios sobre las pruebas en IoT, el CEP y las pruebas aplicadas, la expansión de los lenguajes de procesamiento de eventos, los generadores de eventos y plataformas IoT existentes, definición de operadores de mutación para diversos lenguajes y sus respectivas herramientas, la aplicación de la prueba de mutaciones en sistemas de tiempo real y trabajos relacionados.

- El método para la generación automática de casos de prueba para aplicaciones que procesan eventos se expone en el capítulo 4. En el capítulo se indican cada una de las etapas que lo componen: definición, validación y generación.
- En el capítulo 5 se define y detalla la primera etapa del método propuesto, la definición. En esta etapa se utiliza una propuesta de especificación para definir los diversos tipos de eventos. En este mismo capítulo se introducen cada uno de los elementos de la especificación, los tipos de datos contemplados, así como sus propiedades. Además, se analizan una gran variedad de tipos de eventos para comprobar si estos pueden ser definidos por la especificación que se propone.
- Las etapas de validación y generación, así como los componentes de cada una, se exponen en el capítulo 6. En este capítulo se describe la arquitectura y funcionamiento del componente de validación y del componente de generación. Además se habla de la funcionalidad que permite obtener eventos con valores específicos. A lo largo de todo el capítulo se incluyen ejemplos de cada una de las características y funcionalidades de la herramienta.
- El capítulo 7 presenta el conjunto de operadores de mutación que se ha definido para EPL de EsperTech, para aplicar de forma efectiva la prueba de mutaciones a programas que contengan consultas escritas en este determinado lenguaje de consulta.
- En el capítulo 8 se describe la herramienta MuEPL. Este generador de mutantes se ha desarrollado para automatizar la aplicación de la prueba de mutaciones a programas con consultas EPL de EsperTech. Esta herramienta incorpora los operadores de mutación definidos en el capítulo 7, los genera de forma automática y los ejecuta frente a un conjunto de casos de prueba.
- En el capítulo 9 se analizan y comparan los eventos generados frente eventos reales de una plataforma IoT. También se genera un conjunto de casos de prueba para cada uno de los casos de estudio que se van a emplear en la presente tesis (véase apéndice B). Se aplica la prueba de mutaciones en los casos de estudio mencionados anteriormente, utilizando los casos de prueba originales de los mismos y los casos de prueba generados a través del método. Comparando y analizando los resultados obtenidos determinarán y verificarán algunos de los puntos y objetivos que se exponen en la presente tesis.
- El capítulo 10 recoge las contribuciones y conclusiones que se han extraído de esta investigación, finalizando el capítulo con un conjunto de propuestas de líneas de investigación futuras relacionadas con esta tesis doctoral.
- El capítulo 11 es la traducción al inglés del capítulo 10.

- En el apéndice A se recoge el XML Schema que sigue la propuesta de especificación para definir tipos de eventos.
- En el apéndice B se describen cada uno de los casos de estudio utilizados a lo largo de esta tesis.
- En el apéndice C están las definiciones, según la especificación propuesta, de los tipos de evento de los casos de estudio.
- Finalmente, las referencias bibliográficas empleadas a lo largo de esta disertación.

Capítulo 2

Fundamentos

En este capítulo se recopilan los conceptos fundamentales relacionados con todas las tecnologías y técnicas utilizadas en el desarrollo de esta tesis doctoral, tales como Internet de las Cosas (IoT), arquitecturas dirigidas por eventos, procesamiento de eventos complejos (CEP), los lenguajes de procesamiento de eventos (en concreto el lenguaje EPL de EsperTech) y los generadores de eventos y plataformas IoT. Dado que se va a aplicar una técnica de prueba para validar los eventos de prueba generados por el método propuesto, se exponen los conceptos fundamentales sobre las técnicas de verificación y validación del software, en concreto la prueba de mutaciones (la técnica a utilizar).

2.1. Internet de las Cosas

El Internet de las Cosas (*Internet of Things*, IoT) es un paradigma que está ganando rápidamente terreno en el mundo de las telecomunicaciones. La idea básica de este concepto es la presencia continua, alrededor de nosotros, de objetos o cosas que interactúan y cooperan entre ellas para conseguir un fin común [10].

Se estima que IoT "nació" entre los años 2008 y 2009, cuando el crecimiento explosivo de los smartphones y tablets elevó el número de dispositivos conectados a Internet. La fortaleza de la idea de IoT es el gran impacto que tiene en diversos aspectos de la vida diaria, así como en el comportamiento de sus usuarios potenciales [11]. IoT ofrece una gran cantidad de oportunidades a muchas aplicaciones que prometen mejorar la calidad de nuestras vidas. Desde el punto de vista de un usuario privado, la introducción de IoT afectaría tanto en su vida doméstica como laboral: elementos domotizados, asistentes virtuales, sanidad electrónica (e-health), son algunos ejemplos en los que este paradigma jugaría un papel importante en un futuro no muy lejano. Del mismo modo, desde una

perspectiva de usuarios de negocios, las consecuencias directas serían visibles en campos como: automatización, logística, fabricación industrial, gestión de proceso de negocios, transporte inteligente de personas y cosas... Un ejemplo específico sería la automatización del registro de compras de una tienda de productos electrónicos, que ofrece varios métodos de compra (físico, on-line, a través de smartphones...). El registro de las compras permitirá no solo almacenar la información generada de la misma, sino también detectar posibles errores o situaciones críticas.

IoT es la red de objetos físicos con tecnología embebida que les permite comunicar, percibir o interactuar, tanto con sus estados internos como con el entorno. El concepto de IoT se populariza gracias al software desarrollado por el conjunto de las nuevas y ya establecidas empresas así como al abaratamiento y perfeccionamiento de hardware de comunicaciones. En el año 2011, el número de dispositivos conectados a Internet superó al número de humanos en el planeta, y se estima que para el año 2020, el número de dispositivos conectados a la red, estará entre 26 y 50 miles de millones [12].

IoT proporciona acceso a la información, a los servicios y a los medios de comunicación a través de conexión por cable e inalámbrica. IoT utiliza la sinergia que se genera de las relaciones que se establecen entre los usuarios, las compañías y las industrias. Estas relaciones crean una red abierta y global que conecta a personas, datos y cosas; gracias a esto se le saca el máximo partido a *la nube*[1] conectando objetos inteligentes que perciben y transmiten una amplia gama de datos que ayudan a crear servicios que antes eran imposibles de ofrecer sin este nivel de conectividad. IoT y los servicios que se ofrecen, hacen posible crear redes que permiten transformar simples fábricas en entornos inteligentes. La red de IoT conecta cosas de forma global manteniendo su identidad on-line; como resultado se obtiene una red (accesible globalmente) de cosas, de información y de usuarios, a los que se les permite crear un negocio, contribuir con su contenido, generar y comprar servicios, etc [13].

Las características fundamentales de IoT son las siguientes [14]:

- **Interconectividad**: Gracias a IoT cualquier cosa puede ser interconectada con infraestructura mundial de la información y de la comunicación.
- **Servicios relacionados con cosas**: IoT es capaz de ofrecer servicios según las relaciones establecidas con las cosas que están conectadas; los servicios ofrecidos están dentro de las limitaciones de las propias cosas, como por ejemplo la protección de privacidad y la consistencia semántica entre las cosas físicas y su componente virtual asociado.

[1]Modelo de uso de los equipos informáticos que traslada parte de tus archivos y programas a un conjunto de servidores a los que se puede acceder a través de Internet

- **Heterogeneidad**: Los dispositivos que están conectados a IoT son heterogéneos, hay una gran variedad de dispositivos con diferente hardware y que además están conectados a diferentes tipos de redes. A pesar de esto pueden interaccionar entre otros dispositivos y servicios a través de las redes.
- **Cambios dinámicos**: No solo el estado de los dispositivos está en constante cambio: apagado, encendido, conectado y/o desconectado; también varía como el contexto de los mismos (localización y velocidad). Por otro lado, hay que tener en cuenta que el número de dispositivos puede cambiar de un momento a otro de forma dinámica.
- **Gran escala**: El número de dispositivos a controlar es mayor que el número de dispositivos conectados actualmente a Internet. Pero lo que es aún más crítico es controlar la información generada e interpretarla para su uso en las aplicaciones, lo cual se consigue con la semántica de los datos y un manejo eficiente de los mismos.

Las tecnologías de apoyo para IoT pueden agruparse en tres categorías [15]:

1. Tecnologías que permiten a las cosas adquirir información contextual.
2. Tecnologías que permiten a las cosas procesar la información contextual.
3. Tecnologías que mejoran la seguridad y la privacidad.

Las dos primeras categorías pueden entenderse como dos bloques que se necesitan para construir la ”inteligencia” de las cosas, que son en sí las características que las diferencian del Internet tradicional. La tercera categoría no es funcional, pero es un requisito *de facto*, sin el cual la introducción de la idea de IoT se vería muy reducida.

El desarrollo de IoT implica que los entornos, ciudades, edificios, vehículos, ropa, dispositivos portátiles y otros objetos tengan más y más información asociada a ellos y/o la habilidad de percibir, comunicar, conectarse y producir nueva información. Además, la red de las tecnologías tiene que hacer frente a los siguientes retos: datos a muy alta velocidad, densas aglomeraciones de usuarios, baja latencia, poca energía, bajos costes, y una gran cantidad de dispositivos.

Actualmente existen otros retos que necesitan alcanzarse, tanto a nivel tecnológico como social, para que la idea de IoT sea aceptada. IoT se enfrenta a nuevos problemas referentes a los aspectos de las redes de comunicación, entre ellos destacamos que las cosas que componen IoT se caracterizan por ser recursos de baja capacidad de computación y energía. Las soluciones a estos problemas con las redes de comunicación se tendrán

que enfocar en la eficiencia de los recursos, en la monitorización y la escalabilidad de los datos, y en obtener respuestas con la menor latencia posible. Como consecuencia se han desarrollado y diseñado diversas herramientas, dispositivos, mecanismos y arquitecturas para obtener, procesar y transmitir estos datos intentando solventar los problemas anteriormente comentados [10].

Friess e Ibañez [13] especifican que para lograr la aceptación de IoT se tienen que hacer validaciones de los modelos tecnológicos y de negocio, así como desarrollar proyectos pilotos de aceptación a gran escala que solventen la problemática de la seguridad y confianza de forma integrada. Los proyectos pilotos de IoT, deben de solventar la siguiente lista de objetivos:

- **Solventar las barreras tecnológicas pendientes**, enfocándose en ofrecer una fuerte seguridad. Algunos aspectos de ingeniería tienen que solventarse, como acelerar el proceso de ingeniería para concebir, diseñar, probar y validar sistemas IoT. En cuanto al software, es muy importante controlar un gran número de dispositivos que no pueden ser controlados de forma individual y tienen que ser ejecutados automáticamente.

- **Explorar el potencial de integración** de las arquitecturas IoT y sus componentes con *la nube* y el Big Data. Estas tecnologías (IoT, *la nube* y Big Data) están en continuo desarrollo explotando sus propios dominios.

- **Validar la aceptación del usuario** en las aplicaciones que no están en funcionamiento todavía y que necesitan estudiarse.

- **Promocionar la innovación de las plataformas de sensores y objetos**. Los proyectos pilotos en Internet han fomentado la creación de proyectos en los que se les daba la oportunidad a los usuarios para desarrollar aplicaciones innovadoras que recogen datos de sensores. Se necesita dar un paso más e innovar para que los usuarios no experimentados puedan llegar a comunicarse con objetos inteligentes.

- **Demostrar problemas de casos de uso cruzados** para validar los conceptos de tecnologías genéricas que pueden dar servicio a multitud de entornos y que implican la cooperación de varios operadores (por ejemplo, para entornos de casas inteligentes, pueden cruzarse con entornos de cadenas de alimentación inteligente). Además es esencial ejecutar proyectos piloto desplegando aplicaciones y probando los sistemas en espacios físicos en los que interactúan diferentes cantidades de personas.

Según Thibaut Kleine [13], miembro de la Comisión Europea dentro del “Günther Oettinger’s team”, la Comisión Europea está planteándose dar un mejor soporte a los trabajos

de investigación e innovación en IoT. Para Kleiner, posibles trabajos de investigación para el paradigma IoT serían: desarrollar proyectos pilotos para probar el despliegue de grandes cantidades de sensores relacionados con aplicaciones Big Data, o el lanzamiento de proyectos pilotos a gran escala para probar en la vida real la entrega de soluciones IoT integradas. La seguridad de extremo a extremo es otro de los retos a los que claramente se enfrenta IoT para ser aceptada y adoptada por los usuarios.

2.2. Arquitecturas dirigidas por eventos

Una arquitectura dirigida por eventos (EDA) se refiere a aplicaciones y servicios que tienen la habilidad de reaccionar a cambios en condiciones, independientemente de que el cambio sea un fallo en un sistema de alcantarillado o bien un cambio repentino en la bolsa, como una crisis en Wall Street [16]. Un **evento** se define como un cambio *destacable* en el *estado* de un sistema o su entorno, donde este puede producirse rápidamente o no [17]. Un evento podría significar un problema o un problema inminente, una oportunidad, una amenaza, o una variación.

Según [18] existen dos tipos de eventos: atómicos y complejos.

Eventos atómicos Un evento es atómico si este no puede representarse como un conjunto de sub-eventos esenciales para el dominio de nuestro interés.

Eventos complejos Un evento es complejo si se puede representar como un conjunto de sub-eventos que son esenciales para el dominio de nuestro interés.

Teniendo en cuenta estos conceptos se definen los tres estilos generales del procesamiento de eventos: simple (*simple*), en cadena (*stream*) y complejo (*complex*) [19].

Procesamiento de Eventos Simples *(Simple Event Processing)*, se basa en la sugerencia de que cada evento está directamente relacionado con cambios de condiciones específicos y medibles que pueden ser procesados de forma independiente. Cada evento que provoca cambios destacables inicia *actividades descendentes*. Es decir, actividades que se llevan a cabo como consecuencia de los eventos procesados. Según el ejemplo de la tienda de productos electrónicos, si un cliente realiza un pedido on-line de un producto (evento simple), la acción descendente sería comprobar el inventario de dicho producto.

Procesamiento de Eventos en Cadena *(Stream Event Processing)*, tienen lugar tanto eventos que provocan cambios destacables como eventos ordinarios (que no provocan cambios destacables). Los eventos (órdenes, transmisiones RFID) son evaluados y, si son destacables, son transmitidos a los *suscriptores de información* (personas o máquinas que reciben la información). Siguiendo el ejemplo de la tienda; cuando un sensor RFID (*Radio Frequency IDentification*) emite eventos cada vez que un producto sale del almacén (flujo de eventos) y el distribuidor de productos electrónicos (suscriptor) quiere saber si algún producto de gama alta sale del mismo (evento destacable).

Procesamiento de Eventos Complejos *(Complex Event Processing)*, se ocupa de la evaluación de una confluencia de eventos a partir de los cuales se tomarán una serie de actividades y medidas. Los eventos podrían ser de diferentes tipos, destacados u ordinarios, y pueden ocurrir durante un largo periodo de tiempo. La correlación de los eventos podría ser ocasional, temporal, o espacial. Para este tipo de procesamiento se requiere el empleo de sofisticados intérpretes de eventos, definiciones de patrones de eventos y técnicas de correlación. Un ejemplo en la tienda de productos electrónicos sería el siguiente: si por cada venta se genera un evento ordinario, y el sistema detecta que se han realizado muchas ventas por el mismo cliente, a través de su tarjeta de crédito, en un periodo corto de tiempo (10 minutos) y en diferentes lugares, se activa la alarma de fraude. En la sección 2.3.1 se explica gráficamente el ejemplo expuesto junto con otros más.

Las cuatro capas lógicas de EDA son las siguientes [19]:

Canal de eventos Normalmente es una red troncal de mensajería que transporta eventos con formato estándar (dependerá del sistema), entre generadores de eventos, motores de procesamiento de eventos y suscriptores.

Generador de eventos Cada evento es generado de una fuente. La fuente puede ser una aplicación, un "almacén de datos", un servicio, un proceso de negocios, un sensor, un transmisor, o una herramienta de colaboración (Instant Messaging IM, emails). Según Luckham [4] hay tres principales fuentes de eventos:

1. Capas IT (Information Technology): Los componentes del sistema como bases de datos, receptores de eventos, sistema de mensajería intermedio...
2. Instrumentos: Hardware, monitores software, eventos de sistemas de latidos, software de monitorización de redes...
3. CEP: Eventos complejos creados por el filtrado, agregación y/o unión de eventos simples en el sistema.

Dada la variedad de fuentes de eventos, no todos tendrán el formato que requiere el sistema para que sean procesados. En esos casos, los eventos necesitan transformarse al formato requerido para ser depositados en el *canal de eventos*.

Procesamiento de eventos En la capa de procesamiento de eventos, tras la recepción, los eventos son evaluados frente a unas reglas de procesamiento de eventos (directrices que se comprueban si cumplen o no los eventos), y como consecuencia se inician una serie de *actividades descendentes*. Las reglas de procesamiento de eventos y las *actividades descendentes*, son definidas de acuerdo a las necesidades de las partes interesadas. Las *actividades descendentes* pueden ser desde invocar un servicio, inicializar un proceso de negocio, publicar el evento a un centro de suscripción, dar una notificación directamente a los sistemas o personas, generar un nuevo evento, y/o guardar el evento en un historial, etc.

Actividades descendentes Un simple evento, o una correlación de ellos, podría iniciar una gran cantidad de *actividades descendentes* (actividades que se llevan a cabo como consecuencia de un suceso). La invocación de la actividad se puede dar por un aviso del motor o por una petición de los suscriptores. Los suscriptores pueden ser humanos, aplicaciones, procesos activos de negocio, almacenes de datos y/o agentes automatizados.

2.3. Procesamiento de eventos complejos

A continuación, se define en qué consiste la tecnología CEP, cuáles son sus principales ventajas, así como algunos de los dominios más relevantes donde se puede aplicar. Seguidamente, se analizan los tipos de lenguajes específicos que existen para hacer uso de esta tecnología.

2.3.1. Conceptos Fundamentales

Como se ha comentado previamente, CEP es una tecnología emergente que permite capturar, analizar y correlacionar una ingente cantidad de eventos heterogéneos con el fin de detectar situaciones críticas o relevantes en tiempo real. Esta tecnología se basa en el filtrado de eventos irrelevantes y en el reconocimiento de los eventos que sí son relevantes para un dominio en particular. Para ello se utilizan los **patrones de eventos**, unas plantillas en las que se especifican cuáles son las condiciones que deben cumplirse para detectar dichas *situaciones relevantes*, así como las actividades que deberán de ser ejecutadas tras su detección (*actividades descendentes*). Entendemos como *situación*

relevante una ocurrencia de un evento o una sucesión de eventos que requiere alguna reacción inmediata [20].

Según Luckham [21] los **patrones de eventos** son plantillas que describen eventos o algunos de sus parámetros, a través de operaciones relacionales y variables. Un patrón de evento puede encajar un conjunto de eventos relacionados reemplazando las variables con valores.

En el ejemplo de la tienda de productos electrónicos, cuando se están realizando multitud de compras a nombre de un cliente en un periodo de 10 minutos y en diferentes lugares, la reacción inmediata es activar la alarma de fraude. A estas situaciones de mayor complejidad semántica se les denominan *eventos complejos*. Por tanto, un evento complejo es una abstracción de otros eventos simples o complejos que contiene la información necesaria para describir la situación acontecida [21].

Los eventos complejos procesados mediante tecnología CEP pueden ser identificados e informados en tiempo real, lo cual ayuda a reducir la latencia en la toma de decisiones. Los eventos complejos pueden ser estructurados mediante una *jerarquía de eventos*, donde cada evento de un nivel superior se habrá inferido a partir de uno o más eventos de un nivel inferior. Gracias a esta jerarquía, se logrará reducir el número de eventos que deberá procesarse según el nivel de abstracción requerido por el usuario y/o sistema en cuestión en cada momento.

Para llevar a la práctica este tipo de procesamiento de eventos en los sistemas de información, es necesario hacer uso de un software específico conocido como motor CEP. En la actualidad existen varios motores CEP desarrollados por distintas empresas y grupos de investigación. Cada motor proporciona al programador un lenguaje concreto para que implemente los patrones de eventos que desee detectar en tiempo real; a este tipo de lenguaje se le denomina lenguaje de procesamiento de eventos o EPL; en la siguiente subsección se proporcionan detalles adicionales sobre los motores CEP más relevantes, y sus respectivos EPL.

En la figura 2.1 se muestra un ejemplo de esta tecnología en el que se visualizan las cuatro capas lógicas explicadas en la sección 2.2: generador de eventos, canal de eventos, procesamiento de eventos y actividades descendentes. El ejemplo presenta tres flujos de eventos complejos para una tienda de productos electrónicos que ofrece varios métodos de compra (físico, on-line, a través de smartphones...). En el primer flujo (arriba a la izquierda), un *gateway*[2] para comunicación entre negocios (business-to-bussines, *B2B*), transfiere las operaciones de la tienda. Estas operaciones las emite un "sistema de latidos"

[2]Gateway o puerta de enlace es el dispositivo que permite interconectar redes de ordenadores con protocolos de comunicaciones y arquitecturas diferentes a todos los niveles de comunicación.

que va generando eventos cada 15 minutos, la ausencia de eventos indica un fallo, el gateway ha dejado de funcionar y los clientes podrían realizar un pedido a una tienda competidora.

El motor de eventos hace un seguimiento de la fecha y hora del último evento recibido del "sistema de latidos". Si transcurren 15 minutos y no ha llegado ningún evento, entonces se inicializan las actividades descendentes asociadas a la no llegada de eventos. En este caso, la persona responsable del gateway es avisada y se genera un nuevo evento de "fallo en el gateway". Este nuevo evento es depositado en el canal de eventos. Cuando este es recibido por el motor de procesado de eventos, se lleva a cabo una acción de publicación. La empresa suscribe este evento para el seguimiento de la resolución.

El segundo y el tercer tercer flujo muestran variaciones de detecciones de fraudes. Ambos flujos son originados con la aplicación "punto de venta" (abajo a la izquierda), que tiene un comportamiento similar al primer flujo. Para cada tienda se genera un evento ordinario (Evento B, ocurrencias 1-3). Estos eventos son evaluados por un router de eventos local, produciendo eventos complejos "Valor alto" (Evento C, ocurrencia 1) para transacciones de más de 1,500$. Todos los eventos (simples y complejos), son depositados en el canal de eventos.

En el primer flujo de detecciones de fraudes (Evento B, ocurrencias 1-3), el motor de eventos comprueba si existen múltiples transacciones (los eventos simples) realizadas por un mismo cliente (se identifica por su tarjeta de crédito) en un periodo de tiempo corto (10 minutos) en diferentes localizaciones en una larga distancia (más de 30Km). Si estas condiciones se dan, se establece el estado de la cuenta a "cuenta fraudulenta".

En el segundo flujo de detecciones de fraudes (Evento C, ocurrencia 1), cuando se recibe un evento "Valor alto", el motor de eventos complejos realiza una investigación en el historial de compras del cliente para determinar si esta compra debería ser marcada como sospechosa. Si el precio de la compra se desvía más del 50 % del largo historial de compras, el evento (C, ocurrencia 1) es publicado como sospechoso y se llaman a los titulares de las tarjetas.

Este proceso lo podemos explicar de manera generalizada en tres etapas que están recogidas en [22]. Para una mayor comprensión, estas se expresan gráficamente en la figura 2.2:

1. *Captura de eventos*: consiste en la recepción de los eventos que se analizarán con la tecnología CEP.

2. *Análisis*: a partir de los patrones de eventos definidos previamente en el motor CEP, este se encargará de procesar y correlacionar toda la información, en forma de eventos, con el fin de detectar situaciones críticas o relevantes en tiempo real.

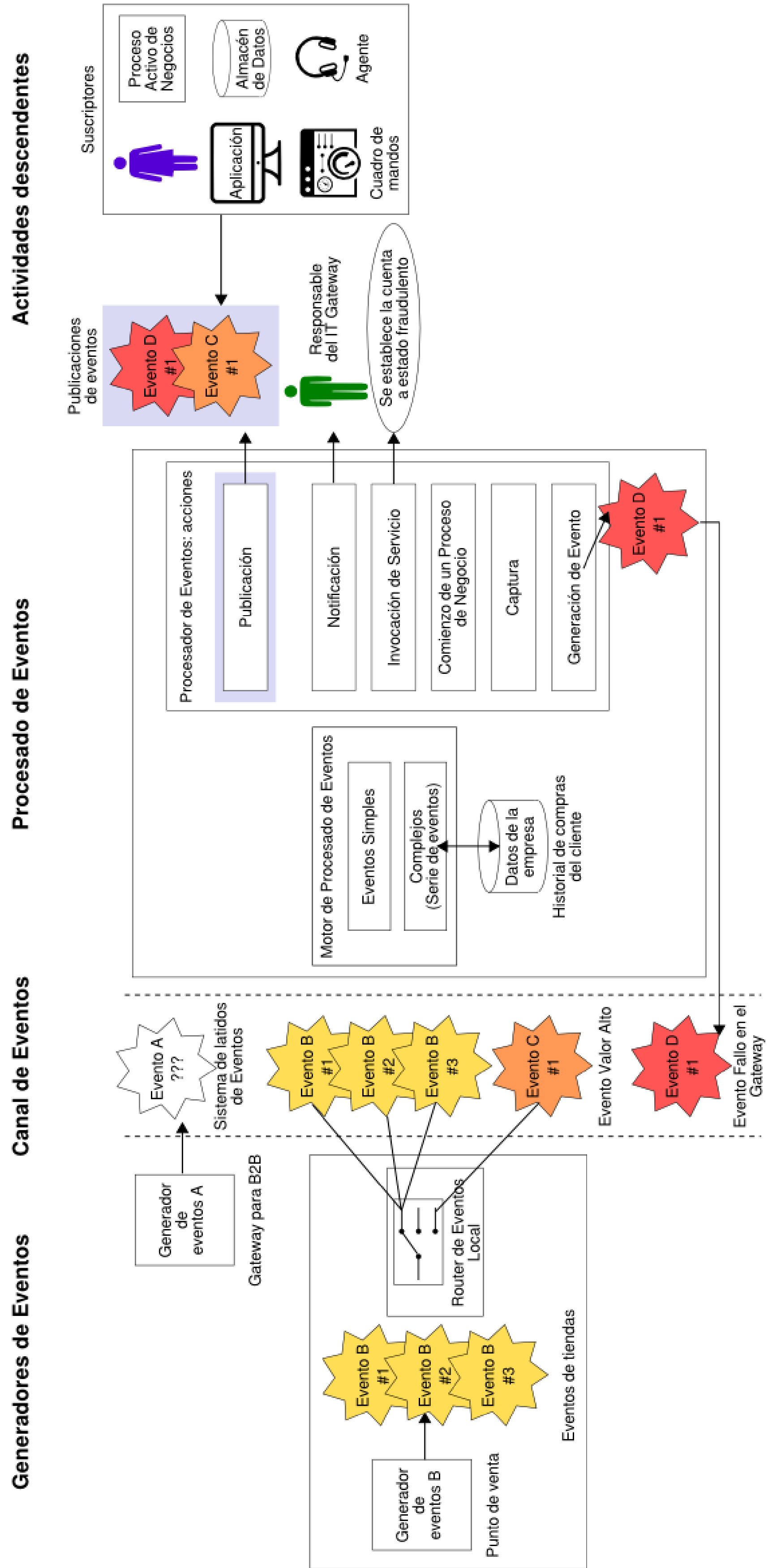

Figura 2.1: Ejemplo de tres flujos de procesamiento de eventos complejos.

3. *Respuesta*: tras la detección de una determinada situación, se actuará en consecuencia notificándoselo al sistema, software, servicio o dispositivo en cuestión.

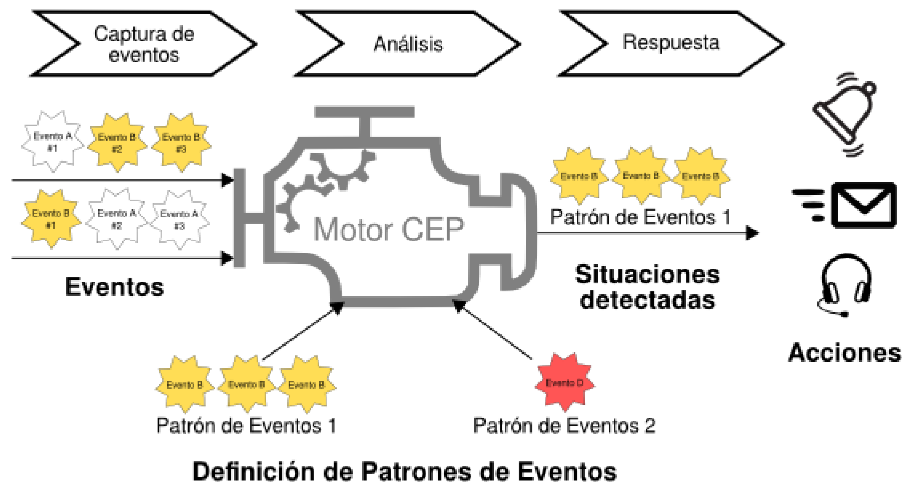

FIGURA 2.2: Etapas implicadas en el procesamiento de eventos complejos.

Siguiendo el estudio [22], las ventajas más relevantes que ofrece esta tecnología son:

- *Mejora de la calidad en las decisiones*: los ordenadores son capaces de gestionar muchísima más información por segundo y de contemplar muchos más factores a la hora de tomar una decisión frente a las capacidades de que disponen los seres humanos.
- *Respuesta en tiempo real*: el uso de un motor CEP que permite realizar automáticamente tanto el análisis de los eventos como la realización de la toma de decisiones, sin que sea necesario la intervención de ningún usuario, conlleva a una respuesta en tiempo real.
- *Prevención de sobrecarga de información*: debido a la naturaleza inherente de los sistemas CEP, se reduce enormemente la cantidad de información que debe ser notificada a las personas, puesto que normalmente solo se mostrarán a los usuarios finales aquellas situaciones críticas o relevantes (eventos complejos) en las que estén interesados; estas situaciones se habrán detectado a través de los patrones de eventos y tras un análisis exhaustivo de toda la información recibida.
- *Reducción del esfuerzo humano*: puesto que se utilizan sistemas informáticos, se reduce el esfuerzo y el trabajo necesarios por parte de las personas para analizar toda la información.

Son muchos y varios los dominios en los que la detección en tiempo real de situaciones críticas o relevantes es una necesidad; en estos es donde la tecnología CEP puede y

debe aplicarse. Algunos ejemplos de los dominios de aplicación para esta tecnología son: detección de fraude y seguridad informática [23, 24], domótica [25, 26], salud [27], servicios basados en localización [28], etc.

2.3.2. Lenguajes de Procesamiento de Eventos

Gracias al uso de patrones de eventos en sistemas CEP es posible la detección de situaciones de interés sobre un dominio concreto en tiempo real. Estos patrones de eventos se definen utilizando unos lenguajes de programación desarrollados para tal fin, denominados EPL. Estos lenguajes pueden clasificarse en tres categorías [20]:

- *EPL orientados a flujos*: estos lenguajes están basados en SQL (Structured Query Language) y el álgebra relacional, pero extendidos para que puedan manejar ventanas temporales de datos. A diferencia de los modelos relacionales convencionales en los que una consulta se ejecuta sobre una tabla de datos, este modelo de consulta continua y en tiempo real deberá ejecutarse sobre un conjunto limitado de eventos (normalmente agrupados mediante ventanas temporales). Algunos de estos EPL orientados a flujos son los siguientes: EPL de EsperTech [9], CQL (Continuous Query Language) de Oracle [29], StreamSQL [30] y CCL (Continuous Computation Language) [31].
- *EPL orientados a reglas*: este tipo de EPL se clasifica a su vez en tres subtipos:
    - *Reglas de producción*: estas reglas son del tipo *si condición entonces acción*, esto es, cuando la condición se satisface, entonces se ejecutará la acción. El EPL del motor Drools Fusion [32] es uno de los más conocidos entre los lenguajes de esta categoría.
    - *Reglas activas*: también denominadas reglas de evento-condición-acción o ECA (Event-Condition-Action) están inspiradas en las bases de datos activas que siguen el siguiente funcionamiento: cuando un evento ocurre, se evalúan las condiciones y, si se satisfacen, entonces se lanza la acción correspondiente. Entre los EPL de reglas activas se encuentra, por ejemplo, el proporcionado por IBM Operational Decision Management [33].
    - *Reglas de programación lógica*: estos lenguajes están basados en aserciones lógicas y en bases de datos deductivas, que permiten hacer deducciones a través de inferencias, a partir de reglas y un conjunto de hechos concreto. Como ejemplo, destacar ETALIS (Event TrAnsaction Logic Inference System) [34], un sistema CEP implementado en Prolog que ofrece dos lenguajes basados en

reglas: ELE (ETALIS Language for Events) y EP-SPARQL (Event Processing SPARQL).

- *EPL imperativos*: lenguajes que permiten definir, mediante programación imperativa, el conjunto de operadores que debe aplicarse sobre los eventos, en el que cada operador especifica una transformación sobre los eventos recibidos. El EPL de Apama [35] es el lenguaje más representativo de los EPL imperativos.

En el estudio realizado por Cugola y Margara [36] se presenta de manera más exhaustiva información sobre los diferentes tipos de EPL.

Es importante destacar que en esta tesis doctoral se ha optado por utilizar el EPL de EsperTech (EPL orientado a flujos). Entre los motivos de la elección destacamos que su sintaxis se aproxima bastante a la del lenguaje SQL, ampliamente conocido a nivel mundial, por lo que la curva de aprendizaje no es tan elevada. Por otro lado, este lenguaje se ejecuta en Esper, un motor CEP escrito en Java y de código abierto, y uno de los más utilizados en la actualidad (en cuanto procesamiento de eventos se refiere), capaz de procesar en torno a 500.000 eventos/s en una estación de trabajo, y entre 70.000 y 200.000 eventos/s en un portátil, según confirma EsperTech [9], la empresa desarrolladora.

2.4. Lenguaje EPL de EsperTech

El lenguaje EPL de EsperTech se ejecuta en Esper, un motor CEP escrito en Java. Es un lenguaje parecido al lenguaje de programación SQL, pero que proporciona una serie de reglas para procesar eventos tales como: filtrado, correlación, aplicación de condiciones y agregaciones. Este conjunto de reglas ayudan a derivar información que proviene de una cadena de eventos y fusionarlos todos.

El lenguaje EPL de EsperTech posee las cláusulas `select`, `from`, `where`, `group by`, `having` y `order by`. Al igual que las tablas del lenguaje SQL, EPLde EsperTech proporciona vistas integradas para colocar los datos. Adicionalmente contiene patrones, operadores, funciones y vistas.

La consulta más simple de EPL de EsperTech contiene al menos una cláusula `select` y una sola definición cadena de eventos. Pero las consultas complejas pueden llegar a crearse incluyendo diferentes condiciones de búsqueda en las cláusulas `where`, o con una cláusula `select` elaborada, etc. En el código 2.1 se muestra un ejemplo la sintaxis de una consulta EPL de EsperTech con la cláusula `select`.

```
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[order by order_by_expression_list]
```

Código 2.1: Ejemplo de sintaxis de una consulta EPL.

En la consulta anterior, la cláusula `select` hace mención a qué atributo hay que seleccionar, mientras que la cláusula `from` especifica el nombre de la cadena de eventos de la cual hay que seleccionar dicho atributo. La cláusula `where` se utiliza para especificar la condición de búsqueda, para filtrar las salidas, además la cláusula `group by` divide las salidas filtradas anteriormente en grupos. La cláusula `having` acepta o rechaza los eventos definidos en la cláusula `group by`, basándose en las condiciones de agrupamiento de la búsqueda. La cláusula `order by` ordena la salida de los eventos según sus *atributos*. Entendiendo como **atributos de un evento** a los componentes de la estructura del mismo.

2.4.1. Sintaxis y cláusulas de EPL de EsperTech

En esta sección se explica la sintaxis de algunas de las diferentes cláusulas de EPL de EsperTech (orientado a flujos). Este lenguaje de programación se ejecuta en Esper, un motor CEP escrito en Java que tiene una gran variedad de especificaciones en las cláusulas, diversos operadores, funciones, palabras claves... Por esto rescatamos solo aquellos conceptos que ayuden a comprender la complejidad y comportamiento de este lenguaje, así como los que son utilizados en la presente tesis. Esto ayudará a comprender mejor las definiciones de los operadores de mutación que se presentan en el capítulo 7.

Cláusula `SELECT`

Se requiere en todas las declaraciones EPL de EsperTech y se usa para seleccionar eventos o atributos de eventos. El carácter comodín `*` se utiliza para seleccionar todos los atributos o una lista específica de atributos.

Selección de todos los atributos de los eventos

La sintaxis para seleccionar todos los atributos puede verse en el código 2.2.

```
select * from stream_def
```

Código 2.2: Seleccionando todos los atributos.

En el ejemplo que aparece en código 2.3, se seleccionarían los eventos de tipo `StockTick` y se mostrarían todos sus atributos. La definición de este tipo de evento la incluiría el desarrollador de la aplicación Java donde se lance esta consulta. Suponiendo que los atributos de StockTick son: symbol, volume y price, aparecerían todos los valores de los atributos de los eventos StockTick.

```
select * from StockTick
```

Código 2.3: Seleccionando todos los atributos.

Elección de atributos específicos

En el código 2.4 puede verse la sintaxis para seleccionar atributos específicos.

```
select event_property [, event_property] from stream_def
```

Código 2.4: Selecionando atributos específicos.

En el ejemplo que aparece en código 2.5, se seleccionarían los atributos `symbol` y `price` de los eventos `StockTick`.

```
select symbol, price from StockTick
```

Código 2.5: Seleccionando todos los atributos.

Expresiones

La cláusula `select` puede estar compuesta por una o más expresiones.

```
select expression [, expression] [, ...] from stream_def
```

Código 2.6: Múltiples expresiones en la cláusula select.

Siguiendo con el ejemplo, en el código 2.7, se seleccionaría el volumen multiplicado por el precio de los eventos `StockTick`.

```
select volume * price from StockTick
```

Código 2.7: Seleccionando todos los atributos.

Renombrando atributos de eventos y expresiones

Siguiendo la siguiente sintaxis se pueden renombrar los atributos de los eventos y expresiones.

```
select [event_property | expression] as identifier [,...]
```

Código 2.8: Renombrando de atributos de eventos.

En el código 2.9, se seleccionaría el volumen multiplicado por el precio de los eventos `StockTick`, pero se especifica que el nombre de la columna resultante es `volPrice`.

```
select volume * price as volPrice from StockTick
```

Código 2.9: Seleccionando todos los atributos.

Ventanas de tiempo y de longitud

Una de las principales diferencias con SQL es el almacenamiento de los datos. En SQL se utilizan tablas y en EPL de EsperTech se emplean *vistas*, las cuales contienen los eventos. Las vistas representan las diferentes operaciones necesarias para estructurar la información en una serie de eventos y para derivar la información de una serie de eventos. Las *ventanas* son vistas que retienen los eventos entrantes hasta que se indica a través de su único parámetro; un valor de tamaño o un valor temporal (dependiendo del tipo de ventana). Las ventanas en Esper pueden ser de tiempo o de longitud, véase su sintaxis en el código 2.10.

A continuación se explica el comportamiento de las ventanas de tiempo.

```
win:length(size_expression) //ventana de longitud
win:time(time_period) //ventana de tiempo
```

Código 2.10: Sintaxis de las ventanas de longitud y de tiempo.

Las ventanas de tiempo son ventanas "movibles" que amplían el intervalo de tiempo especificado en el último en el que se basó la hora del sistema. Las ventanas de tiempo nos permite limitar el número de eventos a considerar por una consulta (al igual que

las ventanas de longitud). El siguiente ejemplo selecciona los eventos `Withdrawal` en los últimos 4 segundos.

```
select * from Withdrawal.win:time(4 sec)
```

Código 2.11: Ejemplo de las ventanas de tiempo.

La siguiente figura 2.3 comienza con un tiempo `t` y muestra el contenido de la ventana de tiempo en `t + 4`, `t + 5`, etc.

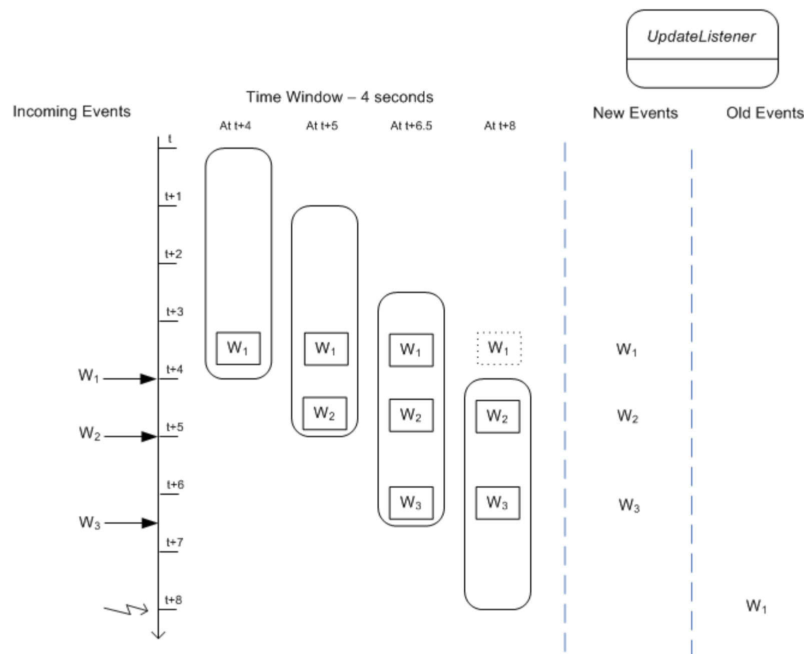

Figura 2.3: Salida de una declaración con una ventana de tiempo.

La actividad que se muestra en la figura es la siguiente:

1. En `t + 4` el evento W1 llega y entra en la ventana de tiempo. El motor envía este nuevo evento para actualizar los *listeners*[3].
2. En `t + 5` el evento W2 llega y entra en la ventana de tiempo. El motor envía este nuevo evento para actualizar los *listeners*.
3. En `t + 6.5` el evento W3 llega y entra en la ventana de tiempo. El motor envía este nuevo evento para actualizar los *listeners*.
4. En `t + 8` el evento W1 deja la ventana de tiempo. El motor envía un aviso sobre este evento indicando que es viejo para actualizar los *listeners*.

[3]Interfaces Java para la notificación de eventos

Las ventanas de longitud son ventanas "movibles" que amplían el número de eventos que se hubiesen especificado anteriormente. El único parámetro que reciben es para definir el tamaño de la ventana.

Ventanas por lotes

Al igual que existen ventanas de tiempo y de longitud, existen ventanas por lotes de tiempo y de longitud, véase su sintaxis en el código 2.12. En este caso las ventanas mantienen un conjunto determinado de eventos que colocan en los *listeners* una vez que tengan una cantidad de eventos determinados, o se cumpla un tiempo específico (depende del tipo de ventana utilizado). A continuación se explica el funcionamiento de las ventanas por lotes a través de las ventanas por lotes de tiempo.

```
win:length_batch(size_expression) //vent. por lotes de long.
win:time_batch(time period) //vent. por lotes de tiempo
```

Código 2.12: Sintaxis de las ventanas por lotes de longitud y de tiempo.

Las ventanas por lotes de tiempo se usan para mantener en el *buffer* eventos para liberarlos pasado un intervalo específico de tiempo, y así actualizar todo de una sola vez.

La siguiente figura 2.4 es para ilustrar la funcionalidad de las ventanas de lotes de tiempo. Para la figura asumimos la consulta del código 2.13:

```
select * from Withdrawal.win:time_batch(4 sec)
```

Código 2.13: Ejemplo del tiempo bash.

La figura comienza con un tiempo `t` y muestra el contenido de la ventana de tiempo en `t + 4`, `t + 5`, etc.

La actividad que se muestra en la figura:

1. En `t + 1` el evento W1 llega y entra en el lote. No se produce ningún informe para actualizar los *listeners*.
2. En `t + 2` el evento W2 llega y entra en el lote. No se produce ningún informe para actualizar los *listeners*.
3. En `t + 4` el motor procesa los eventos que están en el lote y comienza un nuevo lote. El motor envía un informe de los eventos W1 y W2 para actualizar los *listeners*.
4. En `t + 6.5` el evento W3 llega y entra en el lote. No se produce ningún informe para actualizar los *listeners*.

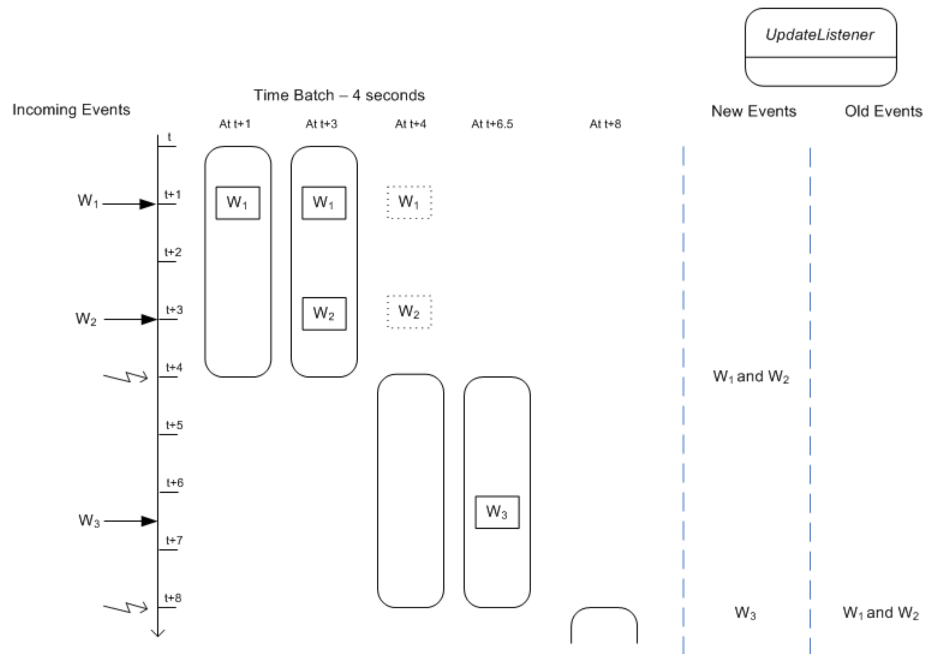

FIGURA 2.4: Salida de una declaración con una ventana de tiempo.

5. En `t + 8` el motor procesa los eventos que están en el lote y comienza un nuevo lote. El motor envía un informe del evento W3 (nueva información) para actualizar los *listeners*. El motor envía un aviso sobre los eventos W1 y W2 indicando que son viejos para actualizar los *listeners*.

Las ventanas por lotes de longitud reúnen un número determinado de eventos, una vez alcanzado ese número específico de eventos, estos son colocados como un lote en los *listeners*.

Palabras reservadas de la cláusula select

Existen también palabras reservadas como `istream` (flujos de eventos a introducir), `rstream` (flujos de eventos a borrar) o `irstream` (flujos de eventos a introducir o borrar), que se utilizan para introducir o eliminar flujos de eventos.

```
select [istream|irstream|rstream] * | expression_list...
```

CÓDIGO 2.14: Palabras reservadas de la cláusula select.

En el código 2.15 se seleccionan solo los eventos que abandonan la ventana de tiempo de 30 segundos.

```
select rstream * from StockTick.win:time(30 sec)
```

CÓDIGO 2.15: Palabras reservadas de la cláusula select.

Filtros

Los filtros para los flujos de eventos, permiten filtrar eventos antes de que estos entren en la ventana. En el siguiente ejemplo, solo se seleccionan aquellos eventos `Withdrawal` que tienen un valor en `amount` mayor que 200.

```
select * from Withdrawal(amount>=200).win:length(5)
```

CÓDIGO 2.16: Selección de eventos usando filtros.

En la figura 2.5 puede verse cómo los eventos W2, W4 y W5 no pueden entrar en la ventana de longitud ya que el valor de `amount` es menor de 200.

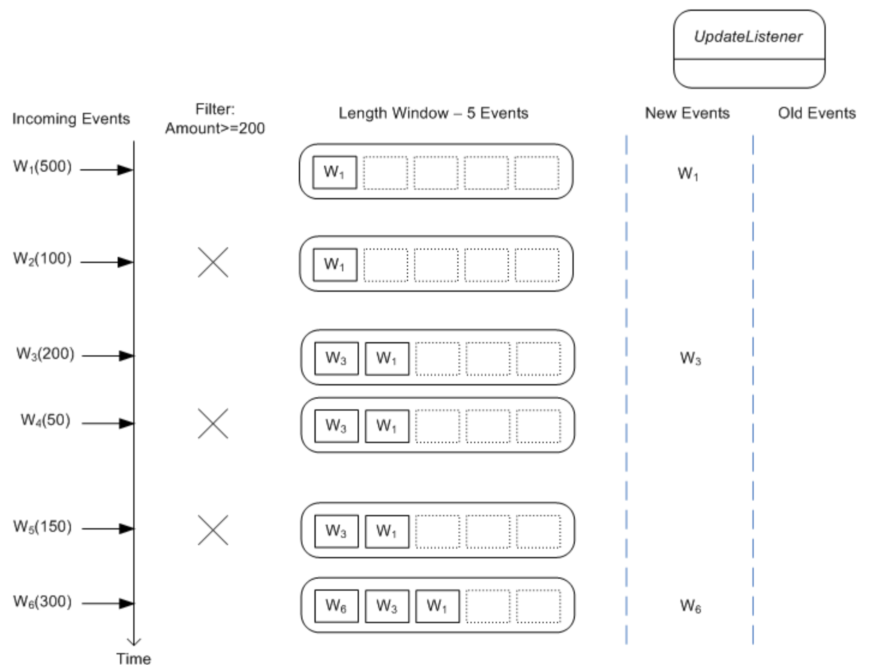

FIGURA 2.5: Declaraciones con un filtro de cadena de eventos.

Funciones de agregación

Las funciones de agregación pueden ser utilizadas para calcular y resumir los datos de los atributos de los eventos. Por ejemplo, se quiere encontrar el precio total de todos los eventos `StockTick`.

```
select sum(price) from StockTickEvent
```

CÓDIGO 2.17: Selección de eventos usando filtros.

Las funciones de agregación pueden ser aplicadas a todos los eventos que estén en una ventana de flujos de eventos, o en cualquier vista, o a uno o más grupos de eventos. Por cada conjunto de eventos para los que se aplica una función de agregación, Esper genera un único valor. La tabla 2.1 recoge algunas de las funciones de agregación de EPL de EsperTech.

Función	Descripción
sum([all|distinct] expression)	Devuelve la suma de todos (o de los valores distintos) de la expresión.
avg([all|distinct] expression)	Devuelve el promedio de todos (o de los valores distintos) de la expresión.
count([all|distinct] expression)	Devuelve el número total de valores no null de todos (o de los valores distintos) de la expresión.
max([all|distinct] expression)	Devuelve el máximo de todos (o de los valores distintos) de la expresión.
min([all|distinct] expression)	Devuelve el mínimo de todos (o de los valores distintos) de la expresión.
stddev([all|distinct] expression)	Devuelve la desviación estándar de todos (o de los valores distintos) de la expresión.
mean([all|distinct] expression)	Devuelve la mediana de todos (o de los valores distintos) de la expresión.

TABLA 2.1: Funciones de agregación que incorpora EPL de EsperTech.

Cláusula `WHERE`

La cláusula `where` es opcional en las declaraciones EPL de EsperTech. A través de esta cláusula se pueden unir y correlacionar flujos de eventos. Cualquier expresión puede colocarse en una cláusula `where`. Normalmente se utilizan operadores de comparación {`=, <, >, >=, <=, is null, is not null`}, operadores lógicos o una comparación de estos utilizando `and` y `or`. De esta forma se unen, comparan y se correlacionan los eventos.

En EsperTech se denominan eventos nuevos a aquellos eventos que entran en la ventana (de longitud o de tiempo), y eventos viejos a aquellos eventos que ya han estado en la ventana (de longitud o de tiempo) y que salen de la misma. La cláusula `where` se aplica tanto a los eventos entrantes (nuevos) como los salientes (viejos). Tal y como muestra la siguiente figura 2.6, los eventos nuevos entran en la ventana de longitud pero solo aquellos que cumplen la cláusula `where` son los que actualizan los "*listeners*". Por otro

lado, de los eventos que abandonan la ventana, solo los que cumplen la cláusula `where` son colocados en los "*listeners*" como eventos viejos. Por ejemplo, en la siguiente consulta solo actualizan los "*listeners*" aquellos eventos cuyo valor `amount` es mayor que 200.

```
select * from Withdrawal.win.length(5) where amount >=200
```

CÓDIGO 2.18: Ejemplo de la cláusula where.

En la figura 2.6 los eventos W2, W4 y W5 no se contabilizan en los "*listeners*" como nuevos eventos ya que el valor de `amount` es menor que 200.

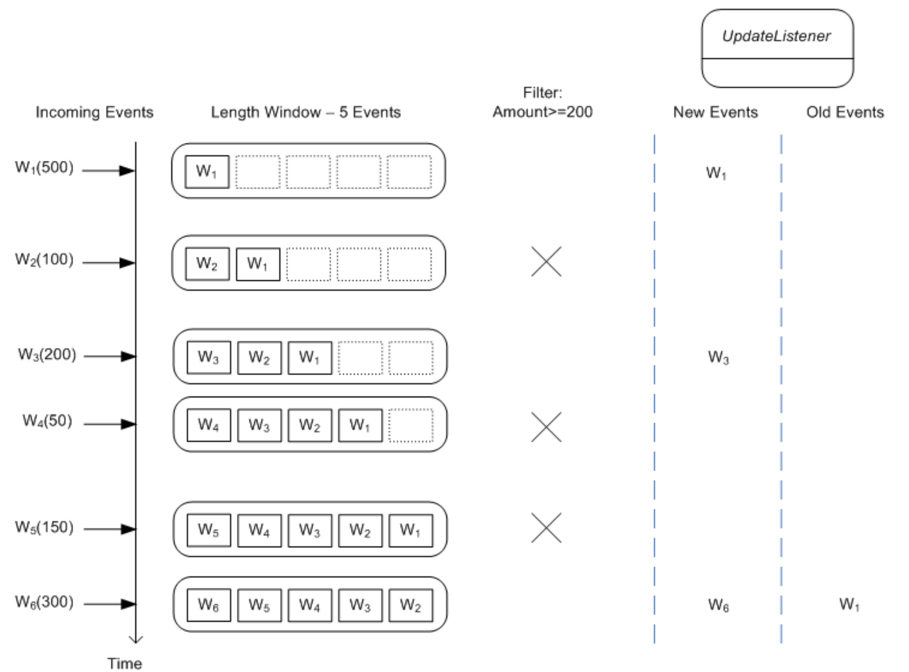

FIGURA 2.6: Declaraciones con la cláusula where.

Cláusula `FROM`

La cláusula `from` ha de aparecer en cada declaración EPL de EsperTech. En esta cláusula pueden aparecer muchas cadenas de eventos y ventanas. La sintaxis general de la cláusula `from` es la siguiente:

```
from stream_def [as name]
[, stream_def [as stream_name]] [, ...]
```

CÓDIGO 2.19: Sintaxis cláusula from.

Indicando un evento de un tipo específico

Tal y como se ha visto en la cláusula `select`, sección 2.4.1, cuando se quieren listar los eventos de un tipo específico, hay que indicar el nombre del mismo. En el código 2.20 se muestra un ejemplo en el que se listan todos atributos de los eventos RfidEvent.

```
select * from RfidEvent
```

Código 2.20: Ejemplo de listado de los eventos del tipo de evento específico RFIDEvent.

Especificando los criterios para un filtro

Los criterios para filtrar eventos con ciertos valores específicos en sus atributos se han de colocar entre paréntesis después del nombre del tipo de evento. En el código 2.21 se muestra un ejemplo en el que se listan todos atributos de los eventos RfidEvent cuyo atributo `category` es igual a `Perishable`.

```
select * from RfidEvent(category="Perishable")
```

Código 2.21: Ejemplo de filtrado de eventos con valores específicos en sus atributos.

Puede usarse cualquier tipo de expresión para hacer los filtrados. Las comas simples separan las expresiones y son interpretadas como operadores lógicos `AND` (véase código 2.22).

```
select * from RfidEvent(zone=1, category=10)
... es equivalente a ...
select * from RfidEvent(zone=1 and category=10)
```

Código 2.22: Ejemplo de filtrado de eventos con valores específicos en sus atributos.

Los siguientes operadores pueden ser utilizados para filtrar un alto volumen de flujos de eventos:

- Operadores de comparación {`=, !=, <, >, >=, <=`}
- Rangos {`(), (], [), []`}, con la posibilidad de usar las palabras reservadas {`between, in, not`}
- Lista de valores usando las palabras reservadas {`in, not in`}

El operador `in` puede utilizarse para filtrar tanto en listas como en rangos. Este operador, junto con otros operadores de interés, serán explicados y se mostrarán ejemplos de su uso en la sección 2.4.1.

Cláusula `GROUP BY`

La cláusula `group by` es opcional en las declaraciones EPL de EsperTech. Esta cláusula divide las salidas en grupos, pudiéndose agrupar por uno o varios atributos o por los resultados de expresiones ya calculadas. Suelen usarse con funciones de agregación, aunque pueden utilizarse sin ellas (algo que produce, generalmente, resultados confusos).

La siguiente consulta devuelve el precio total por cada atributo de evento `symbol` de todos los eventos `StockTick` en los últimos 30 segundos:

```
select symbol, sum(price) from StockTickEvent.win:time(30 sec)
group by symbol
```

Código 2.23: Ejemplo de cláusula group by.

Si de la expresión del `group by` se obtienen valores null, entonces los valores null se convierten en su propio grupo, es decir todos los valores null son agregados en el mismo grupo.

Cláusula `HAVING`

La cláusula `having` es para aceptar o rechazar los eventos definidos en la cláusula `group by`. La cláusula agrupa las condiciones para la cláusula `group by` del mismo modo que la cláusula `where` agrupa condiciones para `select`, con la excepción que `where` no puede incluir funciones de agregación.

La siguiente consulta devuelve el precio total por cada `symbol` de todos los eventos `StockTick` en los últimos 30 segundos, para aquellos cuyo precio total supere los 1000.

```
select symbol, sum(price) from StockTickEvent.win:time(30 sec)
group by symbol having sum(price) > 1000
```

Código 2.24: Ejemplo de cláusula having.

Se pueden incluir más de una condición en la cláusula `having` combinando las condiciones con {`and, or, not`}. En el código 2.25 se muestra una consulta que selecciona solo los grupos con un precio total mayor que 1000 y un volumen de media menor que 500.

```
select symbol, sum(price), avg(volume)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000 and avg(volume) < 500
```

Código 2.25: Ejemplo de cláusula having.

Cláusula `ORDER BY`

La cláusula `order by` también es opcional y se utiliza para ordenar los eventos de la salida por sus atributos, o por expresiones involucradas en los atributos. Sus palabras reservadas son `asc` (para ordenar de forma ascendente, valor por defecto) y `desc` (para ordenar de forma descendente).

La siguiente consulta devuelve el volumen total por `symbol`, ordenados de forma ascendente, de aquellos eventos con volumen mayor que cero de todos los que se encuentren en la ventana `OrdersNamedWindow`.

```
select symbol, sum(volume) from OrdersNamedWindow
group by symbol having volume > 0 order by symbol
```

Código 2.26: Ejemplo de cláusula order by.

OPERADORES

La gramática de EPL de EsperTech contiene una gran variedad de operadores, se rescatan solo aquellos que se consideran relevantes para el estudio que se presenta en esta tesis.

Operadores aritméticos

Los operadores aritméticos siguen la precedencia de los operadores aritméticos del estándar de Java. La siguiente tabla 2.2 expone los operadores aritméticos disponibles:

Operador	Descripción
+, -	Como operadores unarios denotan expresiones positivas o negativas. Como operadores binarios suman o restan.
*, /	Como operadores binarios multiplican o dividen.
%	Como operador binario realiza el módulo.

Tabla 2.2: Operadores aritméticos.

Operadores lógicos y de comparación

Al igual que los operadores aritméticos, los operadores lógicos siguen la precedencia de los operadores lógicos del estándar de Java. A continuación la tabla 2.3 muestra los operadores lógicos y de comparación que existen.

Operador	Descripción
NOT	Devuelve `true` si la condición que le sigue es falsa, devuelve `false` si esta es verdadera.
OR	Devuelve `true` si cualquiera de los componentes de la condición es verdadero, devuelve `false` si ambos son falsos.
AND	Devuelve `true` si ambos componentes de la condición son verdaderos, devuelve `false` si cualquiera es falso.
=, !=, <, >, >=, <=	Operadores de comparación.

TABLA 2.3: Operadores lógicos y de comparación.

PALABRAS RESERVADAS

A continuación se especifican algunas de las palabras reservadas que contempla la gramática de EPL de EsperTech.

Palabra reservada `IN`

La palabra reservada `in` determina si un valor dado coincide con algún valor de una lista. Su sintaxis es la siguiente:

```
test_expression [not] in (expression [,expression...] )
```

CÓDIGO 2.27: Sintaxis de palabra reservada in.

Su resultado es de tipo booleano, si devuelve `true` es que *test_ expression* es igual a cualquier expresión del listado evaluado, en caso contrario devolverá `false`. En el siguiente ejemplo se muestra cómo seleccionar los eventos de tipo `RfidEvent` cuyo `command` se encuentre en la lista:

```
select * from RFIDEvent
where command in ('OBSERVATION', 'SIGNAL')
```

CÓDIGO 2.28: Ejemplo de uso de palabra reservada in.

Esta palabra reservada puede usarse para especificar rangos de cualquier tipo {`()`, `(]`, `[)`, `[]`}. En el siguiente ejemplo se seleccionan eventos de tipo OrderEvent, cuyo precio esté en el rango de 0 a 1000 (ambos inclusive):

```
select * from OrderEvent where price in [0:10000]
```

Código 2.29: Ejemplo de uso de palabra reservada in en rangos.

Palabra reservada `BETWEEN`

Esta palabra reservada especifica el rango a analizar, su sintaxis es la siguiente:

```
test_expression [not] between
begin_expression and end_expression
```

Código 2.30: Sintaxis de palabra reservada between.

Su resultado es de tipo booleano. Si el valor de *test_expression* es mayor o igual que el valor de *begin_expression* y menor o igual que el valor de *end_expression*, entonces devuelve `true`. En el siguiente ejemplo se muestra cómo se seleccionan eventos StockTickEvent cuyo precio esté entre los valores 50 y 60 (ambos incluidos):

```
select * from StockTickEvent where price between 55 and 60
```

Código 2.31: Ejemplo de la palabra reservada between.

Palabra reservada `REGEXP`

La palabra reservada `regexp` es una forma de coincidencia de patrones basada en expresiones regulares. Su sintaxis es la siguiente:

```
test_expression [not] regexp pattern_expression
```

Código 2.32: Sintaxis de palabra reservada regexp.

El valor de *test_expression* es cualquier expresión válida que sea de tipo cadena o numérico. El valor de *pattern_expression* es una expresión regular. El resultado es de tipo booleano; si el valor de *test_expression* coincide con la expresión regular, el resultado es `true`. En caso contrario, el resultado es `false`.

```
select * from PersonLocationEvent where name regexp '.*Jack.*'
```

Código 2.33: Ejemplo de la palabra reservada regexp.

El funcionamiento de las expresiones regulares en Esper sigue la API de Java [4].

[4] `http://www.regular-expressions.info/refflavors.html`

2.4.2. Patrones en EPL de EsperTech

En esta sección se expone cómo usar los patrones del lenguaje EPL de EsperTech. Los patrones de eventos coinciden con un evento o varios cuando estos cumplen la definición del patrón. Las expresiones de los patrones consisten en “patrones atómicos” y “operadores”:

1. Los patrones atómicos son los bloques básicos de construcción de los patrones. Los átomos son expresiones de filtrado, observadores de eventos basados en tiempo y observadores personalizados para eventos externos que no están bajo el control del motor.

2. Los operadores de patrones controlan el ciclo de vida de las expresiones y combinan los patrones atómicos de forma temporal y lógica.

En la tabla 2.4 se muestra un ejemplo de los diferentes tipos de patrones atómicos.

Patrón atómico	Ejemplo
Expresiones de filtrado para buscar un evento	`StockTick(symbol='ABC', price >100)`
Observadores de eventos basados en tiempo que especifican un intervalo de tiempo u horarios	`timer:interval(10 seconds)` `timer:at(*, 16, *, *, *)`
Observadores personalizados para eventos específicos de aplicaciones	`myapplication:myobserver("http://resource")`

Tabla 2.4: Patrones atómicos.

Las expresiones de filtrado dentro de los patrones tienen la misma funcionalidad que los filtros de la cláusula `select`, vistos en la sección 2.4.1. Estos hacen un filtrado de los eventos según las condiciones y valores que se indiquen en el mismo. Los observadores de eventos basados en tiempo, observan eventos basados en tiempo cuyo hilo de control se origina por el temporizador del motor Esper o por un evento temporal externo (estos se verán con más detalle en la sección 2.4.2). Por otro lado, los observadores personalizados pueden ser creados para observar eventos de tiempo o eventos de cualquier otra aplicación con un motor externo.

Hay cuatro tipos de operadores de patrones. Estos pueden verse reflejados en la tabla 2.5. A excepción de los operadores lógicos, el resto de operadores de patrones son exclusivos de EPL de EsperTech. Cada uno de ellos será explicado con detalle en la sección 2.4.2.

La creación de los patrones se lleva a cabo a través de la interfaz `EPAdministrator` (se recuerda que este lenguaje de programación se ejecuta en Esper, un motor CEP escrito

Operador de patrón	Ejemplo
Operadores que controlan repeticiones de sub-expresiones de patrones	`every, every-distinct,` `[num]` and `until`
Operadores lógicos	`and, or, not`
Operadores temporales que se aplican en el orden del evento	`->` (*followed by*)

Las expresiones llamadas *guards* son condiciones `where` que controlan el ciclo de vida de las sub-expresiones	`timer:within`
	`timer:withinmax`
	`while-expression`

Tabla 2.5: Patrones de operadores.

en Java). Esta permite crear patrones de dos formas: patrones que quieren hacer uso de la cláusula `select` o cualquier constructor EPL de EsperTech usando el método `createEPL` que puede incluir una o más expresiones; o declaraciones del lenguaje EPL de EsperTech que usan patrones. Los patrones pueden aparecer en cualquier sitio de la cláusula `from` y pueden combinarse con cláusulas `where`, `group by`, `having`, `insert into`, así como limitar la tasa de salida.

OPERADORES DE LOS PATRONES

Dentro de los operadores que se pueden utilizar en los patrones, existe uno de gran importancia, sin él carecería de sentido la existencia del patrón. Por esto la siguiente sección se dedica por completo a la explicación de este operador.

Operador `EVERY`

El operador `every` indica que la subexpresión del patrón (parte de la expresión que define el patrón) debe reanudarse una vez que la subexpresión ha sido evaluada como verdadera o falsa. Sin el operador `every` la subexpresión del patrón se para tras la primera evaluación, es decir, se deja de buscar coincidencias de eventos con dicha subexpresión. Cuando la subexpresión del patrón se lanza, esta busca coincidencia de eventos una vez; el operador `every` indica al patrón que debe reanudarse la subexpresión para seguir buscando más ocurrencias de los mismos eventos o conjunto de eventos.

En el ejemplo que se muestra en el código 2.34, se considera un patrón genérico en el cual este debe coincidir que por cada evento de tipo A, este va seguido por un evento de tipo B y este seguido por un evento de tipo C, con la condición de que los eventos de tipo B y C deben llegar en un rango de 1 hora después del evento A.

```
every A  -> (B -> C) where timer:within(1 hour)
```

Código 2.34: Ejemplo del operador every.

Si se considera la siguiente secuencia de eventos A_1, A_2, B_1, C_1, B_2, C_2. Lo primero que hay que tener en cuenta, es que este patrón nunca deja de buscar/esperar eventos de tipo A debido al operador `every`.

Cuando A_1 llega, el patrón comienza una nueva subexpresión que mantiene A_1 en memoria y busca eventos de tipo B. De forma paralela, el patrón sigue buscando más eventos de tipo A.

Cuando llega A_2, el patrón comienza una nueva subexpresión que mantiene A_2 en memoria y busca eventos de tipo B. Del mismo modo, el patrón sigue buscando más eventos de tipo A.

Tras la llegada de A_2, hay 3 subexpresiones activas:

1. La primera subexpresión activa contiene en memoria A_1, esperando un evento de tipo B.
2. La segunda subexpresión activa contiene en memoria A_2, esperando un evento de tipo B.
3. La tercera subexpresión activa está esperando el siguiente evento de tipo A.

En el patrón que se ha puesto en el código 2.34, se ha especificado que hay una hora de vida para las subexpresiones que buscan eventos de tipo B y C. En el caso de que no llegasen estos eventos B y C después de una hora de la llegada de A_1, la primera subexpresión desaparece. Del mismo modo, si no llegan eventos de tipo B y C una hora más tarde de la llegada del evento A_2, la segunda subexpresión desaparece. Sin embargo, la tercera subexpresión permanece buscando más eventos de tipo A.

El patrón del ejemplo, según la secuencia de eventos expuesta, con la llegada del evento C_1 (después de la llegada de B_1) puede tener las siguientes combinaciones de eventos $\{A_1, B_1, C_1\}$ y $\{A_2, B_1, C_1\}$, si se considera que B_1 y C_1 llegan en el intervalo de una hora tras la llegada de A_1 y A_2.

En el caso de querer que el patrón, dado un evento A, coincida con unos eventos específicos B y C, se tendrán que usar los atributos `id` de los mismos. Véase su codificación en 2.35:

```
every a=A -> (B(id=a.id -> C(id=a.id))
where timer:within(1 hour)
```

Código 2.35: Ejemplo del operador every con atributo id.

En el patrón del código 2.35, la llegada de C_1 coincidirá para la combinación $\{A_1,\ B_1,\ C_1\}$ y con la llegada de C_2 se realizará la combinación $\{A_2,\ B_2,\ C_2\}$.

Una vez explicado el operador `every`, se recogen en la tabla 2.6 el resto de operadores que pueden ser utilizados en los patrones de EPL de EsperTech.

Operador	Sintaxis	Descripción	Ejemplo	Explicación
`every-distinct`	`every-distinct (distinct_value_expr [, distinct_value_expr [...][,expiry_time_period]`	Similar al operador `every`, la única diferencia es que este elimina los resultados duplicados.	`every-distinct(a.aprop) a=A`	Se mantiene la búsqueda de eventos A con un valor distinto a su propiedad `aprop`.
`repeat`	`[match_count] repeating_subexpr`	Este operador sigue buscando coincidencias un numero determinado de veces, cuando la subexpresión del patrón evalúa `true`.	`[5] A`	El patrón se activa cuando llega el último de los cinco eventos A.
`repeat-until`	`[range] repeated_pattern_expr until end_pattern_expr`	Este operador tiene en consideración un control extra sobre las coincidencias a repetir, una segunda subexpresión que termina la repetición.	`A until B`	El patrón continúa buscando eventos A hasta que llegue un evento B.
`and`	`A and B`	Ambas expresiones tienen que ser `true`.	`A and B`	El patrón coincide cuando llegan ambos eventos (A y B), en el momento de llegada del último de los dos eventos.
`or`	`A or B`	El patrón coincide si cualquiera de las expresiones es `true`.	`A or B`	El patrón busca independientemente eventos A o eventos B.
`not`	`(A ->B) and not C`	Niega el valor verdadero de una expresión.	`(A ->B) and not C`	El patrón coincide cuando se encuentra un evento A seguido de un evento B, y no se ha encontrado antes de la llegada de cualquiera de ellos un evento C.
`followed by`	`A ->B`	Especifica que solo cuando el primer valor de la izquierda sea `true` entonces se evaluará el valor de la derecha.	`every A ->(timer:interval(5 min) and not B)`	El patrón toma todos los eventos A que no son seguidos por eventos B en un intervalo de 5 minutos.

TABLA 2.6: Operadores de patrones.

Guardas de patrones

Los guardas de patrones (*pattern guards*), son condiciones `where` que controlan el ciclo de vida de las subexpresiones. Estas expresiones de control no tienen relación con la cláusula `where` que filtra un conjunto de eventos. En la tabla 2.7 se muestran ejemplos de los diferentes tipos de guardas de patrones.

Guarda	Sintaxis	Descripción
`timer:within`	`timer:within (time_period_expression)`	Sirve como un cronómetro. Si ninguna expresión del patrón se vuelve `true` en un tiempo especificado en el `timer:within`, se para la búsqueda de coincidencias y permanece como `falso`.
`timer:withinmax`	`timer:withinmax (time_period_expression, max_count_expression)`	Similar a `timer:within`, pero incluye un contador adicional para contabilizar el número de coincidencias. La subexpresión termina cuando el cronómetro logra el valor máximo.
`while`	`while (guard_expression)`	La subexpresión del patrón sigue buscando coincidencias mientras que el valor de `guard_expression` sea `true`. En cuanto el valor sea `false`, la subexpresión termina.

TABLA 2.7: Controladores de patrones.

Los guardas de patrones pueden combinarse en un mismo patrón usando paréntesis alrededor de cada subexpresión. En el patrón que se muestra en el código 2.36, se busca la coincidencia de que para cada evento A que se reciba, este tiene que tener un tamaño mayor que 0, y llegar en un rango de 20 segundos.

```
((every a=A) while (a.size > 0)) where timer:within(20)
```

CÓDIGO 2.36: Ejemplo de combinación de guardas de patrones.

OBSERVADORES DE PATRONES

Los observadores de patrones observan eventos basados en tiempo cuyo hilo de control se origina por el temporizador del motor Esper o por un evento temporal externo. La tabla 2.8 recoge los tipos existentes en EPL de EsperTech.

<table>
<tr><th>Observador</th><th>Sintaxis</th><th>Descripción</th><th>Ejemplo</th><th>Explicación</th></tr>
<tr><td><code>timer:interval</code></td><td><code>A -> timer:interval (period_time)</code></td><td>Espera que se cumpla el tiempo definido antes de que el valor del observador se vuelva <code>true</code>.</td><td><code>A ->timer:interval(10 seconds)</code></td><td>Después de la llegada de un evento A, se esperan 10 segundos para indicar la coincidencia del patrón.</td></tr>
<tr><td><code>timer:at</code></td><td><code>timer:at (minutes, hours, days of month, months, days of week, seconds, time zone)</code></td><td>Establece cualquier expresión a <code>true</code> después del tiempo especificado.</td><td><code>every timer:at(5, *, *, *, *)</code></td><td>El patrón coincide con cada hora y 5 minutos (1:05, 2:05, 3:05...).</td></tr>
<tr><td><code>timer:schedule</code></td><td><code>timer:schedule(iso: | [repetitions: | date: | period:]), schedule_expr)</code></td><td>Es un observador muy flexible para establecer horarios. Contiene varios parámetros para establecer los horarios.</td><td><code>timer:schedule(repetitions: 2, date: ’2008-03-01T13:00:00Z’, period: 1 year 2 month 10 days 2 hours 30 minutes)</code></td><td>El patrón organiza dos devoluciones de llamada (callbacks); la primera el 01-03-2008 a la 13:00:00 y la segunda el 11-05-2009 a las 15:30:00</td></tr>
</table>

TABLA 2.8: Observadores de patrones.

2.5. Generadores de eventos y plataformas IoT

Los primeros generadores de eventos que se han desarrollado son bibliotecas para generar eventos que simulan partículas físicas de alta energía. En los trabajos [37, 38] se recogen los generadores de eventos existentes según el tipo de simulación. En estos generadores los eventos se generan de forma aleatoria tal y como se producirían en aceleradores de partículas, experimentos de colisiones o en los inicios del universo.

Los experimentos en los que se emplean estos generadores de eventos son para simular eventos físicos. El estado final de las partículas generadas por los generadores de eventos lo capta un *detector* integrado en la simulación. Esto permite que el sistema de configuración del experimento tenga una predicción y verificación muy precisa. Por otro lado, se utilizan de forma adicional, técnicas de análisis de eventos más sencillas sobre los resultados de los generadores de eventos para completar los resultados del *detector* de la simulación.

Estos generadores de eventos se desarrollaron para experimentos específicos relacionados con la física de partículas. Hoy en día las personas, empresas, negocios... tienen la necesidad de controlar y monitorizar las cosas que los rodean. La información que reciben de ese control y monitorización de las cosas les llega en tiempo real en forma de eventos. Esta lectura de eventos les permite tomar decisiones o realizar acciones a consecuencia de la información recibida. Este es el motivo por el que nacen las plataformas IoT, las cuales son la clave del desarrollo de las aplicaciones y servicios IoT que conectan el mundo real y el virtual entre objetos, sistemas y personas. Las plataformas IoT representan una nueva área compleja y cambiante que no existía hasta hace unos pocos años. A continuación se describen algunas ideas que hay que tener en cuenta sobre las plataformas IoT [39].

No todas las plataformas denominadas IoT son en realidad plataformas IoT: Cuando se habla de una plataforma IoT, se refiere a una plataforma que permite elaborar, desarrollar, almacenar y analizar aplicaciones IoT. Existen otros tipos de plataformas a las que *erróneamente* se les denomina "plataforma IoT":

- **Plataformas (Machine-to-machine, M2M)**: Se centran en la conectividad de los dispositivos IoT a través de las redes, pero no en el procesamiento del conjunto de datos que recogen los sensores.
- **Servidores (Infrastructure-as-a-service, IaaS)**: Servidores que ofrecen su infraestructura para proporcionar espacio de almacenamiento y potencia de procesado para aplicaciones y servicios.

- **Plataformas software para un hardware específico**: Algunas compañías venden dispositivos que se conectan a una plataforma, ofrecida por la misma compañía, que contiene un servidor con un software propietario a la que únicamente pueden conectarse sus clientes.
- **Extensión del software de empresas**: Algunos paquetes software y sistemas operativos ya existentes, están poco a poco permitiendo la integración de dispositivos IoT. A día de hoy, estas extensiones software todavía tienen que avanzar para poder clasificarlas como una plataforma IoT.

La arquitectura de una plataforma IoT: La arquitectura de una plataforma IoT contiene ocho bloques (véase figura 2.7):

1. **Conectividad y normalización**: Se manejan diferentes protocolos y diferentes formatos de información en una interfaz software, donde se tiene que asegurar una transmisión de datos segura, así como una correcta interacción entre todos los dispositivos.
2. **Administración de dispositivos**: Se encarga de asegurar que las cosas conectadas funcionen correctamente.
3. **Base de datos**: El almacenamiento cada vez mayor de los dispositivos provoca llevar a otro nivel las bases de datos híbridas que usan la *nube*, en términos de mayor volumen, variedad, velocidad y veracidad.
4. **Procesado y administración de acciones**: Según los eventos procesados, se lanzan acciones que permiten la ejecución de acciones "inteligentes".
5. **Análisis**: Realiza una gran cantidad de análisis para sacarle el máximo partido a los flujos de datos de IoT.
6. **Visualización**: Permite visualizar patrones y observar tendencias desde un panel de control donde la información se presenta usando gráficos.
7. **Herramientas adicionales**: Permite a los desarrolladores IoT realizar prototipos, probar y comercializar casos de prueba IoT creando plataformas para ecosistemas de aplicaciones permitiendo la visualización, gestión y control de dispositivos conectados.
8. **Interfaces externas**: Permite la integración de terceros y el resto de ecosistemas IT a través del uso de APIs[5], SDK[6] y gateways.

El código abierto permite la interoperabilidad entre las plataformas IoT: Para crear ecosistemas IoT donde los sistemas interaccionen y generen valores de diversos

[5]La interfaz de programación de aplicaciones, (del inglés: Application Programming Interface)
[6]Un kit de desarrollo de software o SDK (del inglés de software development kit)

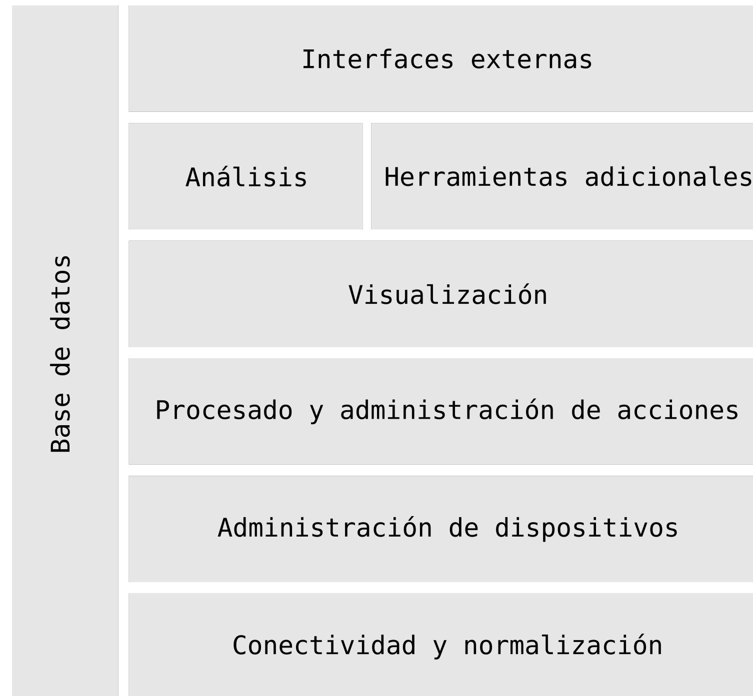

Figura 2.7: Los ocho componentes de una plataforma IoT.

flujos de datos, la interoperabilidad es indispensable. Hoy en día algunas plataformas IoT de código abierto que trabajan de forma conjunta, por el bien común, están incrementando sus beneficios.

2.6. Técnicas de verificación y validación del software

El estándar ISO/IEC/IEEE 29119 2013 [40] define la V&V (Verification & Validation) como el proceso de determinar si los requisitos para un sistema o componentes están completos y correctos, los productos de cada fase de desarrollo cumplen con los requisitos o condiciones impuestas en la fase previa y el sistema final o componente cumple con los requisitos especificados.

El proceso de V&V implica la utilización de técnicas de prueba, las cuales suelen dividirse atendiendo a los siguientes criterios:

- Según si ejecutan o no código. Las técnicas estáticas son las que se realizan sin ejecutar el programa a probar, mientras que las dinámicas sí lo hacen.
- Dependiendo de si se usa información del código fuente del programa o no, se pueden dividir en técnicas de caja blanca o de caja negra. Las técnicas de caja blanca, son aquellas que se realizan accediendo al código fuente del programa. Las de caja negra se hacen sin tener acceso al código fuente del programa. Por lo

general, las pruebas de caja blanca son más completas que las de caja negra. Sin embargo, estas segundas son las únicas que se pueden realizar si solo se dispone del código binario del programa a ejecutar.

- También se pueden clasificar según el tipo de validación que realicen sobre el programa. La prueba funcional comprueba el comportamiento correcto de un software a nivel funcional (que opere correctamente). Mientras que la prueba no funcional, como su nombre indica, no comprueba qué hace el programa, sino cómo lo hace. Esto suele implicar aspectos de usabilidad, portabilidad, restricciones temporales, facilidad de mantenimiento, etc.

De los diferentes criterios, el más extendido es el que se basa en la ejecución o no del código, por lo que se detallará más este tipo de técnica.

La técnicas estáticas se basan en el examen manual o automatizado de la documentación del proyecto, información relacionada con los requisitos y el diseño, el código, etc. Por tanto, estas técnicas pueden emplearse a lo largo de todas las etapas del desarrollo y su uso temprano es muy aconsejable. A continuación se describen algunas técnicas estáticas tradicionales:

- **Inspección del software**: Consiste en el examen visual de los diferentes documentos producidos para detectar errores, violaciones de los estándares de desarrollo y otros problemas.
- **Revisión del software**: Se refiere al proceso por el cual diferentes aspectos del producto se presentan al personal que interviene en el proyecto: directores, usuarios, cliente, etc., y a otros interesados, para que realicen comentarios o para dar su aprobación.
- **Lectura de código**: Se trata del análisis del código producido para detectar errores de tecleo típicos que no violen la sintaxis.
- **Análisis y traza de algoritmos**: Es el proceso en el cual se puede determinar la complejidad de los algoritmos empleados, se analiza el peor caso y el caso promedio, etc.

Los procesos implicados en algunas de las técnicas anteriores son altamente manuales, propensos a errores y costosos en tiempo.

La técnicas dinámicas obtienen información de interés sobre el programa observando su comportamiento durante un número limitado de ejecuciones. El análisis dinámico estándar incluye la prueba y el perfilado (*profiling*). El perfil de un programa registra

fundamentalmente el número de veces que aparecen ciertas entidades de interés durante un conjunto de ejecuciones controladas. Las herramientas de perfilado se utilizan para obtener medidas de cobertura, por ejemplo para identificar dinámicamente invariantes de flujo de control, así como medidas de frecuencia, denominadas espectros, que son diagramas que proporcionan frecuencias de ejecución relativas de las entidades monitorizadas.

Para que las técnicas dinámicas resulten efectivas, el programa a ser analizado se debe ejecutar con los suficientes casos de prueba como para producir un comportamiento interesante. Las técnicas dinámicas proporcionan distintos criterios para generar casos de prueba que provoquen fallos en los programas. Los criterios dependen de la técnica a utilizar, las cuales se agrupan en:

- **Técnicas de caja blanca**: Se necesita conocer la lógica del programa, ya que hay que desarrollar pruebas de forma que se asegure que la operación interna se ajusta a las especificaciones, y que todos los componentes internos se han probado de forma adecuada. Este método se centra en cómo diseñar los casos de prueba atendiendo al comportamiento interno y la estructura del programa. Se examina así la lógica interna del programa sin considerar los aspectos de rendimiento. Las pruebas de caja blanca intentan garantizar que:
  - Se ejecutan al menos una vez todos los caminos independientes de cada módulo.
  - Se utilizan las decisiones en su parte verdadera y en su parte falsa.
  - Se ejecuten todos los bucles en sus límites.
  - Se utilizan todas las estructuras de datos internas.
- **Técnicas de caja negra**: Se llevan acabo sobre la interfaz del software, entendiendo por interfaz las entradas y salidas del programa. No es necesario conocer la lógica del programa, únicamente la funcionalidad que debe realizar. Estas pruebas pretenden demostrar que:
  - Las funciones del software son operativas.
  - La entrada se acepta de forma correcta.
  - Se produce una salida correcta.
  - La integridad de la información externa se mantiene.

En las técnicas de caja blanca, puede ser impracticable realizar una prueba exhaustiva de todos los caminos de un programa. Por ello se han definido distintos criterios de cobertura lógica, que permiten decidir qué sentencias o caminos se deben examinar con los casos de prueba. Estos criterios son:

- *Cobertura de sentencias*: Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute, al menos, una vez.
- *Cobertura de decisión*: Se escriben casos de prueba suficientes para que cada decisión en el programa se ejecute una vez con resultado verdadero y otra con el falso.
- *Cobertura de condiciones*: Se escriben casos de prueba suficientes para que cada condición en una decisión tenga una vez resultado verdadero y otra falso.
- *Cobertura decisión/condición*: Se escriben casos de prueba suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.
- *Cobertura de condición múltiple*: Se escriben casos de prueba suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.
- *Cobertura de caminos*: Se escriben casos de prueba suficientes para que se ejecuten todos los caminos de un programa. Entendiendo camino como una secuencia de sentencias encadenadas desde la entrada del programa hasta su salida.

Dentro de las técnicas dinámicas podemos distinguir la prueba del software. La prueba de software es el proceso que permite evaluar la funcionalidad y corrección de un programa mediante su ejecución sobre un conjunto finito de casos de prueba. En la siguiente sección se hablará de forma más extensa y detallada sobre esta técnica.

2.6.1. Prueba de software

Los autores de la guía SWEBOK [5], Bourque et al., definen la prueba de software como:

> "Software Testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior."

Por tanto se trata de una técnica de prueba *dinámica*, que verifica el comportamiento de un programa sobre un conjunto *finito* de casos de prueba, *seleccionados* de forma adecuada de entre el dominio de ejecución, frente al comportamiento *esperado*.

En esta definición se han enfatizado aquellos términos que aluden a aspectos clave de la prueba de software:

- **Dinámico**: Significa que la prueba siempre implica la ejecución del programa sobre ciertas entradas.
- **Finito**: Indica que durante la prueba solo pueden ejecutarse un número limitado de casos de prueba, elegidos del conjunto completo de casos de prueba que generalmente puede considerarse infinito.
- **Seleccionado**: Se refiere a la técnica utilizada para seleccionar los casos de prueba.
- **Esperado**: Debe ser posible, aunque no siempre es fácil, decidir si las salidas observadas de la ejecución del programa son aceptables o no.

El estándar ISO/IEC/IEEE 29119 2013 [40] define un *caso de prueba* como un conjunto de entradas de prueba, condiciones de ejecución y resultados esperados que se desarrollan con un objetivo particular, como por ejemplo la ejecución un determinado camino de un programa, o para conseguir la conformidad con un requisito específico. Tal como establece la definición, un caso de prueba consta de tres elementos: entradas de prueba, también denominadas datos de prueba, resultados esperados y condiciones de ejecución.

Un *oráculo* es un agente (humano o mecánico) que decide si un programa se ha comportado o no correctamente al ejecutarse con una entrada determinada; de alguna forma, el oráculo debe determinar cuál debe ser la salida del programa para cada dato de prueba. Uno de los principales problemas de la prueba del software es el problema del oráculo, que aparece cuando no es posible predecir la salida para cada entrada [41].

Otro aspecto fundamental de la prueba de software es la selección de los casos de prueba que se van a utilizar para probar el programa. Para que la prueba sea efectiva hay que llegar a un compromiso entre dos necesidades opuestas. Por un lado, está la necesidad de hacer una prueba lo más minuciosa posible, para lo que se necesitaría un gran número de casos de prueba, pero por otro lado tenemos que reducir el tiempo y el coste de la prueba, siendo para ello mejor tener un reducido número de casos de prueba. El procedimiento que se suele utilizar consiste en seleccionar un pequeño conjunto de datos de prueba que se considere representativo del dominio de entrada completo. Lo ideal sería poder elegir los datos de prueba de forma que al ejecutar el programa sobre ese conjunto se descubrieran todos los errores, garantizándose que si un programa produce resultados correctos para el conjunto de datos de prueba también los producirá para cualquier otro punto del dominio de entrada. Sin embargo, descubrir ese conjunto de datos de prueba ideal es, en general, una tarea imposible [42, 43].

Dada la importancia de descubrir ese conjunto de datos de prueba ideal, Goodenough y Gerhart [44] abrieron una línea de investigación para definir qué es un criterio de prueba (o de suficiencia); es decir, el criterio que define lo que constituye una prueba adecuada.

A finales de los noventa, Zhu y col. [45] realizaron una revisión de los criterios de suficiencia, clasificándolos atendiendo a dos aspectos:

1. La fuente de información utilizada para generar los datos de prueba.
2. El enfoque de prueba subyacente.

Si tenemos en cuenta la fuente de información utilizada para generar los datos de prueba, podemos distinguir los siguientes métodos:

- **Métodos basados en la especificación**: Generan los datos de prueba sin necesidad de conocer el código del programa a probar, sino que se puede utilizar la descripción de la interfaz de usuario, una especificación de diseño, una lista de requisitos, etc. Estos métodos también se denominan de caja negra.
- **Métodos basados en el código**: Generan los datos de prueba a partir del código del programa a probar, por lo que se suelen denominar métodos de caja blanca o de caja de cristal.
- **Métodos combinados** Utilizan las ideas de los dos métodos anteriores, se suelen denominar también métodos de caja gris.
- **Métodos no basados ni en la especificación ni en el código** Generan los datos de prueba basándose en la información que se conoce sobre otros sistemas similares. Dado que en estos métodos no se necesita el código del programa a probar, pueden considerarse de caja negra.

Otra posible clasificación es la que tiene en cuenta el enfoque de prueba subyacente, en este caso podemos distinguir los métodos siguientes:

- **Métodos estructurales**: Los datos de prueba generados deben cubrir un conjunto de elementos de la estructura o de la especificación del programa a probar.
- **Métodos basados en fallos**: Miden la capacidad de encontrar fallos que tiene el conjunto de datos de prueba generado.
- **Métodos basados en errores**: Están dirigidos a generar datos de prueba que comprueben el programa en ciertos puntos más propensos a errores.

En la presente tesis se ha optado por emplear una técnica basada en fallos, más en concreto la *prueba de mutaciones*. Esta técnica introducida por Hamlet [46] y DeMillo [47],

consiste en generar programas denominados mutantes, que contienen uno o más cambios con respecto al programa que vamos a probar. El objetivo que persigue es conseguir que el conjunto de casos de prueba generado a través del método propuesto en la presente tesis, nos permita localizar el mayor número posible de mutantes frente al programa original. El apartado 2.7 se dedica a realizar un estudio detallado de esta técnica. Son varios los motivos que han llevado a elegir esta técnica: la experiencia en esta técnica de algunos de los miembros del grupo de investigación de la doctorando, así como la trayectoria de trabajo de la doctorando. Por otro lado, dado que lo que se quiere es probar la validez y efectividad de los eventos generados por el método que se propone en la presente tesis, y ya que la prueba de mutaciones necesita un gran conjunto de casos de prueba (eventos en este tipo de aplicaciones), esta técnica se presenta como la más adecuada para probar los eventos de prueba generados con valores y estructuras específicas.

2.7. Prueba de Mutaciones

La prueba de mutaciones es una técnica de prueba basada en fallos [48, 49] que consiste en introducir pequeños cambios sintácticos en el programa a probar mediante la aplicación de los denominados *operadores de mutación*. Los programas resultantes reciben el nombre de *mutantes*. Cada operador de mutación representa un error típico que puede cometer el programador.

Si un programa contiene la instrucción $a > 2000$ y le aplicamos el operador de mutación de expresiones relacionales (que reemplaza un operador relacional por otro del mismo tipo), uno de los mutantes que se genera podría contener en su lugar la instrucción $a >= 2000$. Cuando introducimos en un programa una sola mutación se produce un *mutante de primer orden*, en el caso de que se produzcan dos o más mutaciones hablamos de *mutantes de orden superior*.

Una vez generados los mutantes, estos se ejecutan frente a un conjunto de casos de prueba. Si un caso de prueba es capaz de distinguir al programa original del mutante, es decir, la salida del mutante y la del programa original (considerada correcta) son diferentes, se dice que dicho caso de prueba mata al mutante. Si por el contrario ningún caso de prueba es capaz de diferenciar al mutante del programa original, es decir, la salida del mutante y del programa original es la misma para todos los casos de prueba de los que disponemos, se habla de un mutante vivo para el conjunto de casos de prueba empleado.

Los *mutantes equivalentes* son uno de los principales inconvenientes de la prueba de mutaciones. Un mutante equivalente produce la misma salida que el programa original, presentan siempre el mismo comportamiento. Estos mutantes no deben confundirse con

los *mutantes resistentes (stubborn non-equivalent mutants)*, que se deben a que el conjunto de casos de prueba no tiene la suficiente calidad para poder detectarlos. Llegados a este punto es necesario aclarar que cuando hablamos de programa en la prueba de mutaciones estamos refiriéndonos a software que está siendo analizado, este puede ser un programa completo o un trozo de código (como una consulta).

La prueba de mutaciones puede ser utilizada de dos formas. Podemos utilizarla para medir la calidad de un conjunto de casos de prueba, calculando la *puntuación de mutación (mutation score)*, que se define como la relación entre el número de mutantes muertos frente al número de mutantes no equivalentes generados. Más formalmente, tenemos:

$$mutation\ score = \frac{K}{M - E} \tag{2.1}$$

donde K es el número de mutantes muertos, M es el número total de mutantes, y E es el número de mutantes equivalentes. Un problema que plantea esta fórmula es que el valor de E, en general, no es conocido, por lo que es necesario inspeccionar manualmente los mutantes vivos para identificar a los equivalentes. El conjunto de casos de prueba de mejor calidad es el que obtiene la mayor puntuación de mutación posible, es decir, consigue matar al mayor número de mutantes. Tal y como Ayari y col. indican en [50], dados dos conjuntos de casos de prueba con la misma puntuación de mutación será preferible el que tenga un menor número de casos de prueba.

El segundo uso de la prueba de mutaciones es la generación de nuevos casos de prueba que maten a los mutantes que permanecen vivos, mejorando así la calidad del conjunto inicial de casos de prueba. Fue a finales de los ochenta cuando Offutt propuso la utilización de la prueba de mutaciones para generar casos de prueba, en [51] se recogen las tres condiciones que se debe cumplir un caso de prueba para que mate a un mutante:

1. La sentencia mutada debe ejecutarse (*reachability*).
2. Una vez ha sido ejecutada la sentencia mutada, el estado de ejecución del mutante debe ser diferente al del programa original (*necessity*).
3. El estado de ejecución tras la sentencia mutada debe propagarse a la salida final del mutante (*sufficiency*).

Lo ideal sería que la puntuación de mutación fuera 1, esto indicaría que tenemos un conjunto de casos de prueba adecuado para detectar todos los fallos modelados por los mutantes.

Los resultados de comparar el comportamiento de los mutantes y el programa original frente a los casos de prueba se representan en una *matriz de ejecución*. Si |M| es el número de mutantes generados y |T| el número de casos de prueba en el conjunto, la matriz de ejecución se representa mediante la fórmula 2.2, donde m_{ij} es 1 o 0 para la mayor parte de los mutantes, dependiendo de si el mutante i-ésimo fue matado o no por el j-ésimo caso de prueba, respectivamente.

$$m_{ij} = \begin{pmatrix} 0 & 1 & 1 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2 & 2 & 2 & \dots & 2 \\ 1 & 0 & 0 & \dots & 1 \end{pmatrix} \tag{2.2}$$

Algunos mutantes pueden fallar al ser ejecutados frente al conjunto de casos de prueba, debido a que violan alguna restricción estática definida por el lenguaje; es decir, estos mutantes pueden fallar al ser compilados o desplegados. Estos son los mutantes no válidos y sus filas en la matriz de ejecución se establecen a $(2, 2, \dots 2)$.

Así, a partir de la matriz de ejecución se puede determinar qué mutantes han muerto y qué casos de prueba los matan, qué mutantes han quedado vivos y cuáles de los mutantes generados son no válidos.

La prueba de mutaciones presenta dos problemas. Por un lado, la gran cantidad de esfuerzo humano que se necesita para detectar todos los mutantes equivalentes. La detección automática de mutantes equivalentes es imposible, porque la equivalencia de programas es un problema indecidible [52]. Por otro, el coste computacional necesario para ejecutar todos los mutantes generados y el programa original frente al conjunto de casos de prueba. En un estudio empírico realizado por Offut y col. [53] se indica que bajo ciertas circunstancias el número de mutantes tiene una complejidad cuadrática en el tamaño del programa. Debido a esto se han propuesto técnicas para la reducción del coste computacional del análisis de mutaciones y para la detección de mutantes equivalentes. Jia y Harman [54] por un lado, y Polo y Reales [55] por otro, examinan las técnicas que se han propuesto para resolver ambos problemas.

2.7.1. Técnicas de reducción de coste

En [54] Jia y Harman clasifican las técnicas de reducción de coste en dos tipos: reducción del número de mutantes y reducción del coste de ejecución.

Gran parte del elevado coste computacional inherente a la prueba de mutaciones es debido a la ejecución de un gran número de mutantes frente al conjunto de casos de prueba, por lo que se ha investigado mucho en encontrar técnicas que permitan reducir el número de mutantes generados sin que se produzca una pérdida significativa de la efectividad de la prueba.

Muestreo de mutantes (mutant sampling) Se trata de un enfoque simple en el que se selecciona al azar un pequeño conjunto de mutantes del conjunto completo. La idea fue propuesta inicialmente por Acree [56] y Budd [57]. Posteriormente se han realizado estudios empíricos que intentan determinar el valor adecuado para el porcentaje de mutantes a generar, obteniéndose valores del 10 %.

Agrupamiento de mutantes (mutant clustering) La idea de esta técnica fue propuesta inicialmente por Hussain [58] y consiste en seleccionar un subconjunto de mutantes utilizando algoritmos de agrupamiento en vez de elegirlos al azar. El proceso comienza generando todos los mutantes de primer orden, que posteriormente se clasifican aplicando un algoritmo de agrupamiento. Los mutantes se agrupan en función de los casos de prueba que los matan. Los mutantes que pertenecen al mismo grupo son matados por un conjunto similar de casos de prueba. Posteriormente se eligen unos pocos mutantes de cada grupo, descartándose el resto. Ji y col. [59] han desarrollado este enfoque utilizando una técnica de reducción de dominios para evitar tener que ejecutar todos los mutantes.

Mutación selectiva (selective mutation) Otra forma de reducir el número de mutantes generados es reducir el número de operadores de mutación a aplicar. Esta es la idea subyacente en la técnica de mutación selectiva, que pretende encontrar un conjunto pequeño de operadores que generen un subconjunto de todos los posibles mutantes sin que se produzca una pérdida significativa de efectividad en la prueba. La idea fue propuesta inicialmente por Mathur [60] y posteriormente fue extendida por Offutt [53]. Se han realizado diversos trabajos [61–63] que presentan técnicas que permiten seleccionar un conjunto de operadores de mutación *suficiente*[7]. Zhang y col. [64] han comparado recientemente la efectividad y estabilidad de la técnica de mutación selectiva frente al muestreo de mutantes, concluyendo que la mutación selectiva no es superior.

Mutación de orden superior (higher order mutation) Polo y col. [65] han propuesto una técnica que combina pares de mutantes de primer orden para conseguir un conjunto más pequeño de mutantes de segundo orden. Esta técnica permite

[7]Un conjunto de operadores de mutación es suficiente si los mutantes que generan pueden representar a todos los mutantes generados por todos los operadores de mutación.

reducir el número total de mutantes a ejecutar así como el número de mutantes equivalentes que se producen. Jia y Harman [66] han generalizado esta idea proponiendo la generación de mutantes de orden superior (HOM) en vez de solo mutantes de primer orden (FOM). Los mutantes de orden superior son aquellos que contienen varios cambios con respecto al programa original, resultado de haber aplicado los operadores de mutación varias veces. Por contra, los mutantes de primer orden contienen un solo cambio sintáctico respecto al programa original. En su trabajo, Jia y Harman introducen el concepto de HOM subsumiente, se trata de un HOM que es más difícil de matar que los FOM a partir de los cuales se ha construido. De esta forma, se pueden sustituir dichos FOM por el HOM correspondiente, disminuyendo así el número de mutantes.

Mutación evolutiva (evolutionary mutation testing) Esta técnica, introducida por Domínguez y col. [67], reduce el número de mutantes a ejecutar mediante la utilización de un algoritmo genético que guía la generación de los mutantes más fuertes (strong mutants). La fortaleza de los mutantes se mide mediante la aplicación de la función de aptitud del algoritmo genético. De este modo, se consigue generar solo aquellos mutantes que van a proporcionar información interesante para el proceso de prueba, reduciéndose el número total de mutantes generados y ejecutados.

2.7.2. Técnicas de detección de mutantes equivalentes

Gracias al estudio de Grün y col. [68], donde analizaban un conjunto de 20 mutantes de programas Java obtenidos por la herramienta JAVALANCHE[8], se han definido las 4 razones principales por las que un mutante puede ser equivalente:

1. La mutación se ha producido en una parte de código no necesaria. Por ejemplo, algunas partes del código duplican algún comportamiento por omisión, o bien establecen un valor que posteriormente es restablecido sin haber sido utilizado mientras.
2. La mutación suprime una mejora de la velocidad de ejecución pero los resultados de la ejecución son los mismos.
3. La mutación solo altera el estado interno pero el comportamiento final no se modifica.
4. La mutación no puede ser lanzada. Se trata de mutaciones que harían fallar el programa bajo unas condiciones específicas, pero si se cumplen dichas condiciones el programa falla antes de alcanzar la mutación.

[8]Javalanche es un framework de código abierto para la prueba de mutaciones de programas Java. Javalanche está enfocado a la eficiencia, efectividad y automatización de las pruebas

Dado que los mutantes equivalentes es uno de los principales problemas a los que se enfrenta la prueba de mutaciones, se ha investigado mucho en la detección automática estos.

Capítulo 3

Estado del arte

En este capítulo se discuten los trabajos relacionados con los diversos aspectos que se abordan en esta tesis doctoral, tales como los estudios sobre las pruebas en sistemas IoT, sobre el procesamiento de eventos complejos (CEP) y las pruebas aplicadas, la expansión de los lenguajes de procesamiento de eventos y los generadores de eventos existentes para CEP. Además, dado que la prueba de mutaciones no había sido aplicada anteriormente al lenguaje de programación EPL de la empresa EsperTech, se destacan varios trabajos donde se sigue el proceso para aplicar este tipo de prueba de software; trabajos en los que se definen operadores de mutación de diferentes tipos de lenguajes de programación y su integración en herramientas de mutación. También se hace un recorrido sobre algunos estudios que aplican la prueba de mutaciones en sistemas de tiempo real. Y, finalmente, se habla de trabajos relacionados con el método para la generación automática de eventos de prueba que se aporta en la presente tesis.

3.1. Pruebas en sistemas IoT

En los primeros años de IoT, fueron muchos los investigadores que consideraban que las pruebas en este tipo de sistemas tenían que hacerse del mismo modo que con cualquier otra aplicación web o de escritorio. A medida que se fue avanzando en esta tecnología y se observó la influencia de la naturaleza de las cosas conectadas a la red IoT, el número de personas que no estaban de acuerdo con la forma de hacer las pruebas como en las otras aplicaciones fue incrementándose.

Según Gerie Owen [69], los motivos por los que difieren las pruebas en sistemas IoT del resto de aplicaciones:

> "Mobile and embedded software and systems are more closely related to the user. Mobile and embedded devices, more than any other technology, are an integral part of our lives and have the potential to become a part of us. This generation of mobile and embedded devices interacts with us, not just awaits our keystrokes. They operate in response to our voice, our touch, and the motion of our bodies."

El software y los sistemas móviles embebidos están más relacionados con el usuario, y los dispositivos móviles y embebidos son parte de nuestras vidas, se han convertido en una extensión de nosotros. Esta generación de dispositivos móviles y embebidos interactúa con nosotros sin esperar a recibir nuestras órdenes desde el teclado, estos responden a nuestra voz, a nuestro tacto y al movimiento de nuestros cuerpos.

Como consecuencia, las pruebas en este tipo de dispositivos tienen que dar un paso más allá según Owen:

> "Since all of these devices actually function with us, testing how the human experiences these devices becomes imperative. If we do not test the human interaction, our assessments and judgments of quality will be lacking some of the most important information needed to determine whether or not the device is ready to ship. [...] The user, a human being becomes a part of the 'internet of things'."

Dado que todos estos dispositivos funcionan con nosotros, es crucial hacer pruebas de cómo interactúa el ser humano en ellos. Si esto no se hiciese, nuestras evaluaciones y juicios sobre la calidad de los mismos tendrían una gran carencia de información para determinar si el dispositivo está o no está listo para usarse. Como consecuencia, el usuario, el ser humano, se convierte en una parte del IoT.

Peter Varhol, de acuerdo con las afirmaciones anteriores, realiza la siguiente recomendación:

> "I would recommend that testers look at their own experiences with their Internet-connected devices. If you are in some way dissatisfied, you need to question if the dissatisfaction is due to the device itself, or how it works with you."

Observar las experiencias de uno mismo con los dispositivos que tenemos conectados a Internet. Si no estamos satisfechos con un dispositivo, tenemos que preguntarnos si esa insatisfacción se debe al dispositivo en si, o a cómo funciona con nosotros.

Es por esto por lo que se tienen que hacer pruebas de todo tipo: físicas (tamaño, forma y género de los usuarios), sensoriales (reacciones a la luz, el sonido y al tacto), de orientación, de interacción con los movimientos, geográficas, condiciones ambientales...

Es muy importante que todas estas nuevas formas de hacer pruebas consideren el contexto real donde se utilizarán estos sistemas de comunicación [13]. En esos contextos están interconectadas y distribuidas cosas, máquinas, objetos inteligentes... lo que impone que se avance hacia métodos de prueba distribuidos que sean muy cercanos al entorno real donde van a operar. En el contexto IoT, donde los objetos están conectados de diferentes formas, la comunicación podría ser poco fiable y no controlada si no se proporcionan conjuntos de prueba eficaces y precisos, así como metodologías de prueba asociadas a la interoperabilidad que ayuden a hacer pruebas en profundidad a los protocolos usados por las cosas, máquinas y objetos inteligentes y a los servicios y aplicaciones embebidas.

En el proyecto [70], donde se vela por la seguridad de los trabajadores, se realizan las pruebas en el contexto real donde se utilizará este sistema. En este trabajo se presenta un simulador a alto nivel con estándares web aplicados a un entorno industrial (interior) para controlar la seguridad de los trabajadores. Este proyecto, FASyS (Fábrica Absolutamente Segura y Sostenible), tiene como reto principal la monitorización continua de los trabajadores para una detección de riesgos proactiva. Para poder lograrlo han construido un simulador que representa a varios sensores de diferentes tipos y lo han integrado con una base de datos de sensores estándar. Además, han desarrollado una interfaz amigable para las personas que sirve para monitorizar todos los sensores, y una aplicación central de control para hacer pruebas de uno de los riesgos dentro de la fábrica; detección de colisiones. A pesar de que las redes de sensores se emplean normalmente en entornos exteriores, su uso les ha facilitado hacer pruebas mucho más rápidas, sin riesgo y reduciendo los costes de operación.

En 2001 la fundación OWASP [71] aparece en la red, una comunidad libre y abierta que permite a las organizaciones concebir, desarrollar, adquirir, operar y mantener aplicaciones de confianza. Todas las herramientas, documentaciones, foros y capítulos son libres y abiertos para cualquiera que esté interesado en mejorar la seguridad en las aplicaciones. En 2015 comienza el proyecto "OWASP Internet of Things Project" [72], donde se quiere definir una estructura para proyectos IoT tales como; guías para pruebas, lista de vulnerabilidades...

En su guía para realizar pruebas en IoT, presentan una lista básica para ayudar a los desarrolladores a hacer pruebas de seguridad en IoT. Aseguran que si todos los elementos de esta lista los cumple el producto IoT, este habrá mejorado su seguridad. La lista de consideraciones para asegurar un producto IoT es la siguiente:

1. **Inseguridad en la interfaz Web**: Hay que evaluar si se aceptan contraseñas débiles, la posibilidad de cambiar el usuario y contraseña, el uso de HTTPS, las vulnerabilidades de la web, etc.

2. **Insuficiencia en la autorización/autenticación**: Evaluar los mecanismos de recuperación de las contraseñas, las soluciones para las contraseñas fuertes, para la expiración de la contraseña después de un periodo, para la opción de cambio del usuario y contraseña que vienen por defecto, etc.

3. **Servicios de redes inseguros**: Hay que evaluar las soluciones que aseguran que los puertos no están presentes, las que indican si la red de servicios responde mal si hay sobrecarga en el búfer, etc.

4. **Falta de encriptado en el transporte**: Hay que evaluar las soluciones que determinan el uso de comunicación encriptada entre dispositivos, dispositivos e Internet, las que determinan si se utilizan las prácticas de encriptado, etc.

5. **Preocupaciones en la seguridad**: Hay que evaluar las soluciones que determinan la cantidad de información personal que ha de almacenarse, si esta información está debidamente protegida utilizando encriptación tanto a la hora de almacenarse como a la hora de transmitirse, etc.

6. **Inseguridad en la interfaz de *la nube***: Hay que evaluar las vulnerabilidades de la seguridad, la no aceptación de contraseñas débiles, la disponibilidad de contraseñas fuertes, la expiración de la contraseña después de un periodo, el uso de encriptación a la hora de transportar los datos, etc.

7. **Inseguridad en la interfaz de los móviles**: Hay que evaluar la no aceptación de contraseñas débiles, la disponibilidad de contraseñas fuertes, la expiración de la contraseña después de un periodo, el uso de encriptación a la hora de transportar los datos, la cantidad de información personal almacenada, etc.

8. **Insuficiente capacidad de configuración para la seguridad**: Hay que evaluar si las opciones de seguridad de la contraseña están disponibles, si las opciones de encriptado están disponibles, si las alertas y notificaciones de seguridad para el usuario están disponibles, etc.

9. **Inseguridad en el *firmware* del software**: Hay que evaluar si el dispositivo tiene capacidad de actualizarse rápidamente cuando las vulnerabilidades están visibles, si utiliza ficheros encriptados actualizados y si estos se transmiten utilizando la encriptación, etc.

10. **Pobre seguridad física**: Hay que evaluar el dispositivo para determinar si permite deshabilitar los puertos físicos no utilizados, si incluye la posibilidad de limitar capacidades administrativas a una interfaz local, etc.

En el siguiente trabajo, se presenta un prototipo IoT para el control médico que no cumple la lista básica de OWASP para las pruebas de seguridad. Laranjo et al. introducen en [73] una arquitectura IoT que usa etiquetas RFID (*Radio Frequency IDentification*) que permiten establecer un control médico sencillo y remoto, así como la implementación y análisis de un prototipo, con interfaz web, que permite evaluar el servicio propuesto. Los autores indican que el uso de este tipo de etiquetas en un contexto médico permite una identificación rápida y precisa de cada paciente y, para IoT, un acceso rápido a la información médica del paciente. La integración de RFID e IoT, promociona una mejor interacción entre médico y paciente ya que con este tipo de sistemas no solo se pueden mandar avisos a los médicos, enfermeras o personal sanitario, sino que también se puede mejorar la monitorización y control del bienestar de los pacientes de forma remota. La información que maneja este prototipo es confidencial, así que ha de cumplir ciertos aspectos de seguridad. Actualmente no se tiene acceso a la información si no se tiene autorización, pero reconocen que el prototipo actual no incluye ningún mecanismo de seguridad fuerte. Además, reconocen que tienen que utilizar la encriptación para la protección de datos en las bases de datos, en las claves de los usuarios y en los mecanismos de intercambio de información. Durante el proceso de prueba del prototipo se asumió que la información estaba encriptada.

Para que un sistema IoT tenga una seguridad basada en la lista propuesta por OWASP, se necesitan controlar una gran cantidad de dispositivos y cosas, lo que conlleva un gran coste en verificación en sistemas IoT. Cabe mencionar las conclusiones de Brian Bailey en [74] sobre las diferentes perspectivas de los desarrolladores y empresarios en cómo recortar los costes de verificación para IoT:

> "The verification problem needs to be looked at differently for IoT devices, but not everyone agrees on where to start or how it should be done."

En definitiva, el problema de verificación necesita mirarse de manera diferente para cada uno de los dispositivos IoT, pero nadie se pone de acuerdo por dónde se debe empezar o cómo debería llevarse a cabo.

3.2. Procesado de Eventos Complejos (CEP)

La tecnología CEP la propone David Luckham para solventar los principales problemas de los sistemas que manejan, en tiempo real, un gran volumen de información en forma de eventos: la monitorización de dicha información y responder con la menor latencia [4]. A raíz de trabajar con esta tecnología, Luckham plantea el uso de lenguajes de alto nivel para la detección de patrones de eventos y reglas. Años más tarde Schiefer et al. [75] presentan Event Processing Language (EPL), un lenguaje estandarizado para definir patrones de eventos y reglas, el cual se ha especializado dando lugar a varios EPL con diferentes propósitos y características.

Siguiendo la línea cronológica, fue en 2009 cuando Dunket et al. [76] definen nuevos conceptos para la tecnología CEP:

- **Instancias de eventos**: Toda situación que requiera una reacción por parte del sistema.
- **Tipos de eventos**: Clasificación de las instancias de evento según sus características, es decir, cada instancia de evento pertenece a un *tipo de evento*.

Este último concepto (tipo de evento) es recogido por Luckham [77] quien lo define con mayor detalle exponiendo que:

> "Events are processed by computer systems by processing their representations as event objects. The same activity may be represented by more than one event object; each event object might record different attributes of the activity."

La actividad a procesar puede representarse por más de un evento, y de cada uno se recogerán diferentes atributos. Sigue indicando que:

> "An event attribute is a component of the structure of an event."

El **atributo de un evento** es un componente de la estructura de un evento. Y concluye:

> "All events must be instances of an event type. An event has the structure defined by its type. The structure is represented as a collection of event attributes."

Todos los eventos deben ser instancia de un tipo de evento, cuya estructura está definida por su tipo. La estructura se representa como una colección de atributos de evento. Luego si un tipo de evento se crea dependiendo de los atributos que hemos guardado, la colección de los atributos de un evento determina su estructura.

A medida que se avanza en la tecnología CEP, se comprueba la importancia de los eventos, sus atributos y, como consecuencia, los tipos de eventos. Hoy en día existen varios trabajos en los que se resalta la relevancia de estos según el propósito del estudio.

En 2010, Robins [78] reconoce que una gran parte de CEP consiste en detectar situaciones relevantes a través de las condiciones de los patrones que han de cumplirse. La detección de situaciones a través de los patrones consiste no solo en el filtrado (para determinar eventos de interés, o para sistemas con alto tráfico para descartar los eventos que no son interesantes), sino también en la extracción de propiedades (atributos) de los objetos de los eventos que pueden ser combinados con eventos de un nivel superior (eventos complejos). En el trabajo dan como ejemplo el siguiente evento complejo; el tiempo de comienzo y finalización de una transacción podría ser una parte útil de un evento generado o, en una red de eventos con una fuente común, la dirección de esa fuente podría ser utilizada en un evento con un nivel superior.

Luckham et al. exponen en otros estudios [79, 80] que el procesamiento de eventos complejos opera tanto en conjuntos de eventos como en las diferentes relaciones existentes entre estos. Estas relaciones que pueden ser especificadas en patrones de eventos, mapas y filtros. Un ejemplo de relación es el *Tiempo*: el evento A ocurrió antes del evento B, donde el tiempo se representa por marcas de tiempo *timestamp* (un atributo).

Ahn y Kim definen en [81] el concepto de *contexto* como un conjunto de eventos interrelacionados (llamados *eventos componentes*) a través de relaciones lógicas y de tiempo. A través de un ejemplo sobre el *contexto* de detección de fuego, explican que este puede detectarse analizando dos *eventos componentes*: temperatura y la cantidad de monóxido de carbono en el aire. Estos *eventos componentes* están interrelacionados gracias a sus estructuras y sus atributos relacionados.

En el estudio [82] de Perea et al. mencionan que la tecnología CEP debería procesar solo una cantidad seleccionada de información que permita entender el contexto de una forma precisa. Este filtrado de información ayudaría, tanto a nivel de hardware como de software, a obtener la información importante. A nivel de hardware ayudaría a reducir los costes de comunicación entre redes, y a alto nivel se ahorraría energía con el procesado de la información relevante.

3.3. Pruebas en aplicaciones CEP

El objetivo de las pruebas en cualquier software es proporcionar información sobre determinados aspectos del mismo. Según el tipo de software que estemos analizando tendrá más importancia o relevancia la información que nos ofrezca un tipo de prueba u otra. A pesar de ser una misma tecnología la que se utilice en un conjunto de programas (en nuestro caso hablamos de la tecnología CEP), dependiendo del ámbito de aplicación del programa habrá pruebas que nos aporten más información que otras. Es por esto por lo que se pueden encontrar diferentes tipos de pruebas sobre aplicaciones que emplean la tecnología CEP.

Habibi et al. [83] introducen un procedimiento de pruebas en seis fases para aplicaciones web dirigidas por eventos. El comportamiento del Software Dirigido por Eventos (SDE) se produce con los eventos que "entran" en el sistema. Uno de los principales problemas que existen a la hora de probar los sistemas SDE es que se necesita generar una gran cantidad de eventos que cubran todas las fases del mismo. Las seis fases del procedimiento son:

1. Dividir la aplicación según la estructura de la misma: Dependiendo de la tecnología utilizada, Ajax o Flash, se podrán actualizar partes de la página web sin que afecte a toda la página. Con estas bases, se dividen los objetos de la web en: objetos internos de la página y objetos externos de la página.

2. Crear un grafo funcional por cada sección: Se crean grafos según las funcionalidades de cada sección; cada nodo representa un estado del programa y la relación entre cada estado lo describe el evento situado en los ejes que unen los nodos.

3. Crear grafos mutantes de los grafos funcionales: Utilizando como base la prueba de mutaciones basada por modelos[1], se generan grafos mutantes aplicando operadores de mutación: borrar nodo, añadir nodo, borrar eje, cambiar eje y añadir eje. Con este último se obtiene como resultado un nuevo eje al cambiar el origen y el destino de cada evento con otro estado del grafo.

4. Seleccionar el criterio de cobertura y producir las rutas para las pruebas: El objetivo es obtener las rutas para hacer las pruebas tanto en los grafos funcionales como en los mutantes. Para evitar los bucles en las rutas para hacer pruebas y para cubrir todos los criterios de cobertura se utilizan diferentes herramientas ya desarrolladas.

5. Elegir los candidatos para la unión de la secuencia de eventos: Se escogen los candidatos según prioridades; si son representativos de cada sección tendrán más

[1] La estructura sintáctica del programa se modela usando una máquina de estados.

prioridad que si no son representativos. La prioridad se calcula: se ordena de mayor a menor basándose en el número de cobertura de los requisitos de prueba (NC), la ruta que obtenga el número NC mayor y además tenga un número bajo de pre-requisitos tiene más prioridad, y aquellos con un número NC menor se seleccionan en el último paso.

6. Derivar y ejecutar los casos de prueba: Todas las rutas de pruebas son entradas a la fase final de generar los casos de prueba. Los casos de prueba se ejecutan y se comparan resultados.

En sus conclusiones destacan tanto las ventajas como los inconvenientes encontrados, entre los que se destaca que debido al uso de la prueba de mutaciones la mayoría de las fases en su propuesta tuvieron que hacerse de forma manual (pasos 1, 2, 3 y 6) y, como consecuencia, el tiempo de ejecución de las pruebas se vio afectado. Por otro lado se reducen los casos de prueba, sin verse afectadas las funcionalidades esenciales, gracias al uso de grafos funcionales y las condiciones impuestas en la creación de mutantes en el operador que ”añade ejes”.

El sistema de pruebas QCEP-TS (de código abierto) y el lenguaje EPTDL (*Event Processing Test Definition Language*) [84] han sido creados para entender el desarrollo dirigido a pruebas de aceptación ejecutables (EATDD) de sistemas CEP. Los autores consideran que el nombrar de manera efectiva los eventos y sus atributos, ayuda a que las pruebas de aceptación sean más expresivas. Es por esto por lo que han diseñado e implementado el lenguaje EPTDL y el sistema de pruebas QCEPS-TS, los cuales se explican a través de una aplicación CEP aplicada al campo de análisis de las redes sociales.

Belli y Linschulte [85] han desarrollado una propuesta basada en eventos para modelar y hacer pruebas de comportamientos funcionales para servicios web en SOA a través de grafos de secuencias de eventos (GSE). Cada uno de los nodos representan a los eventos (de petición o de respuesta) y los arcos representan la secuenciación de los mismos. Para representar los valores de los parámetros, los GSE se aumentan convirtiéndose en tablas de decisiones. Esta propuesta se valida a través de un caso de estudio de un sistema comercial web (con SOA).

Las interfaces gráficas de usuario (GUI) son difíciles de testear ya que las entradas son interactivas, y las salidas pueden ser gráficos o pueden ser eventos. Por otro lado si se quiere hacer algún cambio y el programador no se sabe cómo es el diseño de software de la GUI, este necesitará un método para probarla. En [86] White presenta un test de regresión automática que puede utilizarse para probar las interacciones en la GUI: interacciones de eventos dinámicas y estáticas.

En un estudio posterior [87] se desarrolla una familia de criterios de cobertura para hacer pruebas en GUI basada en las pruebas de interacción combinatoria. Utilizan técnicas combinatorias ya que estas permiten incorporar ”contextos” en los criterios en función de la combinación de eventos, de la longitud de la secuencia y de la inclusión de todas las posiciones posibles de cada evento. Este estudio demuestra que se han detectado muchos más fallos (previamente no detectados con otras técnicas) gracias al incremento de la combinación de los eventos y al control de las posiciones relativas de los eventos.

FINCoS [88] es un framework que puede ser utilizado como benchmark[2] en sistemas CEP. FINCoS se creó para resolver algunos de los problemas que existen a la hora de utilizar la técnica de benchmark: la falta de estándares, los diversos dominios donde se utiliza CEP, las métricas. Los autores consideran que FINCoS ayudará a la creación de nuevos benchmarks, así como a acelerar las mejoras en los motores CEP. En un trabajo posterior [89], tras observar que el rango de escenarios donde se aplica CEP es muy extenso y que además presenta diferentes requisitos operacionales en términos de rendimiento, tiempo de respuesta, tipos de evento, patrones, número de fuentes, escalabilidad... los autores no tienen claro cuál de estos requisitos operacionales tiene más necesidad de los motores CEP, o qué ocurre cuando los parámetros están cambiados o si el rendimiento va poco a poco degradándose. Para solventar esta falta de información con respecto al rendimiento del procesamiento de eventos se presentan micro-benchmarks para estresar operaciones fundamentales: selección, agregación, unión, patrones de detección, etc. Además, realizan una evaluación experimental extensiva de tres productos CEP, dos comerciales (de los cuales no revelan los nombres por motivos de licencia) y uno de código abierto (Esper [9]).

Otros estudios que se centran en usar la técnica de benchmark en sistemas CEP [90, 91] presentan CEPBen, que ha sido diseñado para evaluar el comportamiento funcional de sistemas CEP. El benchmark CEPBen fue implementado en la plataforma Esper [9] y usado para explorar tanto los factores de rendimiento como nuevas métricas en la gestión del rendimiento en sistemas CEP. Los resultados demuestran que el rendimiento de estos sistemas está influenciado por el comportamiento de las funcionalidades CEP.

Saboor y Rengasamy presentan en [92] un trabajo en el que se dan nociones sobre cómo debe diseñarse y desarrollarse aplicaciones CEP, se proponen unos puntos muy útiles para la realización de pruebas, desde diferentes perspectivas, para este tipo de aplicaciones. Entre otros: que el generador de eventos debe tener la habilidad de adaptarse (igual que la aplicación se adapta a la nueva fuente de eventos), los eventos de entrada deben ser generados bajo una aleatoriedad controlada (entre un rango con valores máximos y mínimos) y los ficheros con los eventos de entrada tienen que tener el formato `csv` o `xml`.

[2]Técnica utilizada para medir el rendimiento de un sistema o componente del mismo

Los autores recomiendan, para la realización de pruebas, que el generador de eventos contemple los puntos anteriormente citados y que sea genérico, es decir, que genere datos según un tipo de evento dado.

El método presentado en la presente tesis, a diferencia de los estudios analizados, puede utilizarse para realizar una gran variedad de pruebas en programas que procesen eventos. Esto es posible ya que el método permite generar eventos personalizados y adaptados para un gran rango de tipos de prueba. La propuesta de especificación para definir tipos de eventos que también se presenta, ayuda a definir, de una forma personalizada, los eventos para las pruebas. Para estas dos aportaciones, se siguen no solo las nociones que se indican en el estudio [92], sino también las directrices de Luckham [93].

3.4. Expansión de los lenguajes de procesamiento de eventos

A raíz de los estudios publicados a principios del siglo XXI (véase 3.2), se desarrollaron lenguajes de programación para detectar las situaciones de interés sobre un dominio concreto en tiempo real. Ese domino de aplicación es el que los categoriza y clasifica (véase 2.3.2). A continuación se detallan los orígenes y los estudios relevantes de algunos de los Lenguajes de Procesamiento de Eventos.

3.4.1. EPL orientados a flujos

Dentro de esta categoría se encuentra el EPL de EsperTech [9], un lenguaje aplicado de forma industrial que, gracias a su condición de carácter libre, es analizado en la gran mayoría de los estudios que se irán citando en esta tesis para hacer comparativas, basarse en su sintaxis, o para aplicarlo en diferentes áreas. Este lenguaje elaborado por la compañía EsperTech (2006) se ejecuta en Esper, un motor CEP escrito en Java. La compañía tiene detrás una comunidad de código abierto que está continuamente actualizando y mejorando todos sus productos.

El lenguaje CQL de Oracle se basa en SQL pero añade constructores que soportan cadenas de datos. Nace a raíz de la necesidad de ejecutar consultas sobre un continuo e ilimitado conjunto de datos que varían de forma constante. Este lenguaje es específico de Oracle y se adapta a la arquitectura Oracle CEP también diseñada por esta compañía. A diferencia de EPL de Esper, este lenguaje no incluye el reloj que permitiría soportar un procesado periódico y no es código abierto. En [94] utilizan el lenguaje CQL de Oracle junto con StreamSQL para diseñar un modelo que le da al usuario el control sobre el

nivel de detalle en el que expresar la simultaneidad. StreamSQL tampoco es de código abierto a pesar de su origen (una investigación académica).

Otro de los lenguajes de esta categoría es Continous Computational Language (CCL) desarrollado para Coral8, un sistema que soporta una gran variedad de tipos de formato de entrada y salida. La mayoría de las características de este lenguaje provienen de SQL, aunque el flujo de trabajo y el entorno de ejecución de este sistema es significativamente diferente de las bases de datos relacionales. En [95] se presenta una arquitectura EPS (Sistemas de Procesados de Eventos) en un contexto *e-market*, que permite integrarse con diferentes EPS (llamados EPS *esclavos*) bajo una interfaz de usuario con un dominio específico y unificado. En el estudio ponen como EPS esclavos a Coral8 y SMARTS (The Securities Markets Automated Rsearch Trading and Surveillance). El primero muy adaptable y flexible y el segundo diseñado específicamente para ajustarse a los requisitos de los dominios del mercado financiero. SMARTS es un sistema software comercial de vigilancia de los mercados financieros, las transacciones se almacenan en bases de datos que solo son accesibles por módulos de software SMARTS. Los patrones de eventos usados en SMARTS se describen con ALICE, el lenguaje de programación propietario de SMARTS.

3.4.2. EPL orientados a reglas

Tal y como se describió en la sección 2.3.2, este tipo de EPL se clasifica a su vez en 3 subtipos: producción, reglas activas y reglas de programación lógica.

En [96] encontramos un estudio donde se utiliza el motor Drools Fusion (EPL de producción). Este se emplea para analizar los datos de un framework para *Entornos Inteligentes* (IE) a través de dos motores de reglas: uno Drools y otro Esper. El motivo de usar dos motores diferentes es para demostrar que la arquitectura que proponen permite el reemplazamiento de uno de los motores por el otro de forma sencilla, sin tener que modificar componentes. Además demuestran la facilidad de expresar reglas complejas en diferentes lenguajes de programación.

Aunque existen publicaciones previas sobre el modelo del sistema ETALIS [97] (EPL de programación lógica), no es hasta 2011 cuando Darko lo presenta en su Tesis [98] como un sistema que permite la especificación y monitorización de cambios casi en tiempo real. Los cambios pueden especificarse con patrones de eventos complejos y ETALIS los detecta en tiempo real mientras que evalúa el conocimiento básico. Se determina que ese conocimiento básico proporciona el contexto o dominio en el que los eventos son interpretados. Tanto en su Tesis como en [99] Darko presenta el sistema ETALIS utilizando dos lenguajes (desarrollados en el mismo periodo de tiempo) para la especificación de los

patrones de eventos: ETALIS Language for Events y Event Processing SPARQL (ambos de código abierto, pero con distintos objetivos). En el libro de Darko [100] se presenta con detalle el lenguaje ETALIS Language for Events (ELE), un lenguaje declarativo basado en reglas para procesar eventos. EP-SPARQL se propone para cubrir el hueco existente en cuanto a herramientas semánticas. Estas herramientas pueden manejar de manera eficiente el conocimiento básico y realizar razonamientos sobre el mismo, pero no pueden soportar el rápido cambio de información que provocan las cadenas de eventos. Este lenguaje EP-SPARQL es para eventos complejos y razonamiento de cadenas, en [101] muestran su sintaxis, ejemplos de ejecución, etc.

IBM presenta, además de Operational Decision Management (lenguaje EPL de reglas activas), SPL (Streams Processing Language) [102]. SPL no es un lenguaje CEP ya que no esta enfocado principalmente en la coincidencia de eventos, sin embargo consideran que es posible implementar CEP en SPL envolviendo un motor de coincidencias de CEP en un operador.

3.4.3. EPL imperativos

La empresa Progress ha desarrollado el lenguaje Apama EPL, un lenguaje que ellos llaman "MonitorScript". Según Louis Lovas [103] este lenguaje imperativo proporciona un estilo más natural de desarrollo (similar a los lenguajes de programación tradicionales como Java y C++) y tiene los fundamentos básicos para que un lenguaje de programación sea satisfactorio [104]: modularidad, encapsulado, interfaces e instancias. Estos fundamentos permiten construir componentes y soluciones reutilizables para los frameworks. La compañía posee una comunidad activa que ayuda y mantiene informados a los usuarios de este lenguaje.

En esta tesis doctoral se ha optado por utilizar el EPL de EsperTech (EPL orientado a flujos). Entre los motivos de la elección destacamos que su sintaxis se aproxima bastante a la del lenguaje SQL, ampliamente conocido a nivel mundial, por lo que la curva de aprendizaje no es tan elevada. Por otro lado, este lenguaje se ejecuta en Esper, un motor CEP escrito en Java y de código abierto, y uno de los más utilizados en la actualidad (en cuanto procesamiento de eventos se refiere), capaz de procesar en torno a 500.000 eventos/s en una estación de trabajo, y entre 70.000 y 200.000 eventos/s en un portátil, según confirma EsperTech [9], la empresa desarrolladora. Tal y como se ha comentado, la compañía tiene detrás una comunidad de código abierto que está continuamente actualizando y mejorando todos sus productos. Y existe una gran cantidad de programadores

que han utilizado github[3] como plataforma donde publicar y compartir sus aplicaciones con consultas EPL de EsperTech.

3.5. Generadores de eventos y plataformas IoT

Un repaso sobre los generadores de eventos existentes nos revela que algunos son de propósito comercial [105, 106], otros solo pueden ser utilizados en aplicaciones específicas ofertadas por los autores [107], y otros se centran en la generación de eventos de áreas muy específicas [37, 38, 108], como condiciones ambientales.

Para algunos estudios los generadores de eventos son herramientas indispensables para simular procesos físicos complejos. En [38] los autores realizan un recorrido por las herramientas que generan eventos para aplicaciones específicas que recogen los datos de las colisiones de partículas (alta energía producida en la simulación).

La biblioteca de datos HepSim se presenta en [108] e incluye una serie de muestras de eventos que recogen los valores de la alta energía producida en las simulaciones. En otro estudio [37] se hace una revisión sobre los generadores de eventos que simulan colisiones de hadrones.

Tal y como se comentó en la sección 2.5, los primeros generadores se centraban en experimentos de un ámbito específico, pero la necesidad de información por parte de las personas y empresas dio lugar al IoT y sus plataformas. Los tres siguientes generadores de eventos, son extensiones de software ya existentes; y que son, erróneamente llamados, plataformas IoT.

Timing System [105] proporciona un sistema distribuido de tiempo que incluye la generación de señales de tiempo. Su generador de eventos es el responsable de crear y mandar eventos de tiempo al componente "Receptor de eventos". Los eventos de tiempo están normalmente sincronizados a un reloj principal, en este sistema se trata de un reloj de radio frecuencia externo que envía tramas de eventos en serie.

La compañía Starcom System [106] ha creado un generador de eventos que intenta resolver las necesidades de los desarrolladores de sistemas IoT en cuanto a los eventos que se necesitan las aplicaciones que están desarrollando, la cantidad de eventos y las características que han de cumplir. La compañía Starcom System ha localizado estos problemas y como consecuencia han desarrollado un generador de eventos on-line, que según describen, se puede monitorizar de forma paralela o de forma secuencial, lo que

[3] Una de las plataformas de desarrollo colaborativo más utilizadas a nivel mundial. Website: `http://www.github.com`

permite un uso amplio y variado de la misma según los requisitos que necesita el cliente final (según la aplicación a desarrollar). Starcom System ha añadido a su generador de eventos la capacidad de controlar el final de la acción del evento. Controlando la acción final del evento se pueden filtrar los requisitos exactos que el cliente necesita. Esto se consigue a través de una condición incluida en la monitorización (de forma paralela o secuencial).

La consola WebLogic Integration Solutions [107] permite manejar y monitorizar las entidades y fuentes necesarias para las aplicaciones WebLogic que desarrolle el usuario. Este sistema contiene un módulo para el generador de eventos que se integra en las aplicaciones WebLogic implementadas por el usuario. El generador de eventos permite definir el sistema de eventos que lanzará mensajes a los canales del usuario. El generador de eventos publica mensajes en los canales del "Agente de mensajería" en respuesta a los eventos del sistema. En WebLogic, diferencian los eventos según el contenido de los mismos: ficheros, emails, JMS (*Java Message Service*), tiempo, MQ (*Message Queue*, mensajes en cola), HTTP y RDBMS (*Relational Database Management System*, mensajes que indican que un proceso está esperando). Para cada uno de estos diferentes eventos se debe de incluir, en el sistema desarrollado, un módulo con el generador de eventos correspondiente.

Al igual que los generadores de eventos, algunas plataformas IoT son públicas [109–116] y otras de propósito comercial [117, 118], y aunque la mayoría permiten al usuario gestionar el tipo de actividad a monitorizar, también las hay especializadas en áreas.

Entre las plataformas públicas se encuentra ThingSpeak [109], una plataforma que se utiliza como fuente de eventos para el caso de estudio "Ecological Island" (véase sección B.3) de la presente tesis. Esta plataforma permite coleccionar y almacenar información, recogida con sensores, en *la nube*. Esta plataforma proporciona una serie de aplicaciones que permiten analizar y visualizar la información en MATLAB. La información que se recoge con los sensores puede ser enviada a través de Arduino, Raspberry Pi, BeagleBone Black y cualquier otro hardware. Esta plataforma utiliza canales para almacenar la información enviada, estos pueden hacerse públicos y así compartir la información.

Otra plataforma pública es Lelylan [110], que provee de una alta elasticidad y escalabilidad para construir, emplear y manejar aplicaciones IoT basada en microservicios. La arquitectura de los microservicios consiste en un nuevo enfoque para el desarrollo de una aplicación software como una serie de pequeños servicios, cada uno ejecutándose de forma autónoma y comunicándose entre sí. Hay un número mínimo de servicios que gestionan las cosas comunes (como el acceso a la base de datos), pero cada microservicio es pequeño y corresponde a un área de negocio de la aplicación. Este enfoque de desarrollo ha sido utilizado en los últimos años por compañías como Netflix y Amazon para el reparto satisfactorio de sus productos, Lelylan aplica esta estructura al Internet de las Cosas.

Particle [111] es una plataforma que permite desarrollar productos IoT que evolucionan rápidamente de su prototipo a producción. Es una plataforma de código abierto, con soluciones a todos los niveles para dispositivos conectados a *la nube*.

Buglabs [112] es una plataforma que se centra en dar soluciones y en el desarrollo de herramientas IoT. Utilizan un modelo de negocios que encaja perfectamente con la naturaleza en la que se basa la comunicación entre máquinas y los mercados IoT.

Construida sobre Node.js [119], Zettajs [113] es una plataforma de código abierto que se centra en la creación de servidores que funcionan en equipos distribuidos por el mundo y en la nube. Zettajs combina REST APIs, WebSockes y programación funcional reactiva[4] para la unir cualquier dispositivo conectado a Zettajs en tiempo real.

Grovestreams [114] es una plataforma abierta en la que cualquier organización, usuario o dispositivo puede utilizar. Esta plataforma ha sido desarrollada utilizando las últimas tecnologías para el manejo de grandes cantidades de información que emplean compañías como Google, Yahoo y Facebook. Esta plataforma permite crear, probar y desplegar de forma rápida, segura y sencilla las soluciones de sus usuarios evitando posibles costes y riesgos.

Plotly [115] contribuye con una gran cantidad de proyectos de código abierto y es compatible con una gran variedad de herramientas. Plotly permite analizar, visualizar y trabajar gráficamente con la información. Se auto-denomina como una plataforma colaborativa para la "ciencia de datos moderna".

Sensorcloud [116] es una plataforma que permite captar, visualizar, monitorizar y analizar la información. Su API está implementada como un servicio web usando HTTP y su SDK contiene librerías y código de ejemplo en Python, Java y C#, ambas accesibles.

Una plataforma que antes era abierta y que ha cambiado su política es Xively [117], esta se presenta como una plataforma simple, escalable y segura para el manejo de cualquier producto conectado. Según las características de su conectividad, esta permite procesar sobre unos 86 billones de mensajes por día.

Carriots [118] es una plataforma que funciona como un servicio, diseñada para IoT y la comunicación entre máquinas. Permite captar y almacenar cualquier tipo de información de los dispositivos, construir aplicaciones con su motor SDK y desplegar y escalar pequeños prototipos a miles de dispositivos.

[4]Su objetivo es permitir toda la potencia de la programación funcional moderna dentro de sistemas que requieren soluciones reactivas. La idea básica consiste en modelar las entradas y las salidas como valores que varían en el tiempo.

Tal y como se ha podido comprobar, la mayoría de plataformas IoT, permiten crear, desplegar, almacenar productos IoT, así como captar, guardar, enviar y analizar la información.

El método que se propone en la presente tesis para generar eventos de pruebas, tiene un comportamiento similar a los canales de información que incluyen las plataformas IoT. Los eventos que se ofrecen en las plataformas IoT, vienen de los sensores, dispositivos, herramientas u cosas que están conectadas a la red IoT de la plataforma. Para utilizar estos eventos en pruebas, se tendrían que modificar manualmente los valores de los atributos, o esperar a que se den las condiciones específicas y obtener los eventos con los valores deseados. El método que se presenta se centra exclusivamente en los eventos y sus atributos, permitiendo personalizarlos y adaptarlos según las necesidades del usuario y las pruebas. Una vez definido el tipo de evento se pueden obtener, de forma automática, tantos eventos personalizados como se necesiten.

3.6. Herramientas de validación de consultas EPL de EsperTech

El método que se propone en la presente tesis se compone de tres etapas (estas se verán con detalle en el capítulo 4): definición, validación y generación. Para la etapa de definición se presenta una propuesta de especificación para los tipos de evento. En la segunda etapa la definición realizada por el usuario será validada y, si esta es válida, se pasará a la etapa de generación de eventos.

Hoy en día existen diversas herramientas para pruebas que se encuentran en la red que realizan un proceso de validación y verificación de código. A continuación se comparan algunas herramientas que validan la correcta sintaxis de consultas escritas el lenguaje de programación estudiado en la presente tesis (EPL de EsperTech):

Simple EPL Query Tester [120] Es una aplicación basada en el motor CEP de Esper para iniciarse en la programación de consultas EPL. La herramienta ofrece una interfaz gráfica donde se pueden probar las consultas que esperan únicamente eventos de tipo "Transaction" con los atributos: price, symbol, buyer, seller y volume; los cuales tienen valores configurables. La herramienta posee un editor donde se escribe la consulta y otras secciones donde se visualizan los resultados de la misma, el historial y el estado.

Tal y como se ha comentado, esta herramienta es para probar si es correcta o no la sintaxis de consultas EPL de EsperTech. Luego la gran diferencia de Simple EPL

Query Tester, con el método que se propone en la presente tesis, es que este último es para comprobar la definición de tipos de eventos, no consultas EPL de EsperTech. Esta herramienta solo posee un tipo de evento "Transaction", que aunque pueden configurarse los valores de sus cinco atributos, no deja de ser muy limitada. A pesar de sus limitaciones, consideramos que este proyecto reciente, puede ser una buena forma de aprendizaje.

Esper EPL Online [121] Es una aplicación para comprobar el funcionamiento de consultas EPL ejecutando secuencias de eventos. Este servicio que ofrece la compañía EsperTech, permite configurar la secuencia de eventos y el tiempo de ejecución. Una vez configurados los parámetros, se incluyen todas las consultas EPL que se deseen probar y, tras la ejecución, se muestran los resultados en el denominado "Escenario de resultados".

Este servicio online de EsperTech tiene una finalidad similar a Simple EPL Query Tester, pero este es mucho más amplio. Se permiten definir los tipos de eventos necesarios, el rango del tiempo de ejecución y las consultas a probar son más complejas. A pesar de poder definir el tipo de evento que se desee, hay que definir manualmente cada uno de los valores de los eventos a procesar por la consulta. Dependiendo del tipo de prueba que se quiera realizar, este servicio puede no ser muy adecuado. En cambio, utilizando el método propuesto en esta tesis, se pueden generar de manera automática el conjunto de eventos necesarios con los valores que se deseen.

Norikra [122] es un servidor de código abierto escrito en JRuby que permite procesar flujos con consultas EPL de EsperTech pero con algunas limitaciones. Solo pueden ejecutarse consultas con la cláusula `select` (no soporta `insert into` ni `update`), tampoco soporta el acceso a datos externos, etc. Las consultas que se permiten son simples, "`select ... from ...`", sin ningún tipo de esquema adicional. Según su autor, Tagomori, el rendimiento de Norikra se puede ver afectado por el número de objetivos, el número de consultas, la complejidad de las mismas y la complejidad de las funciones definidas por el usuario. Actualmente se pueden ejecutar 10 consultas y procesar 2000 eventos por segundo haciendo un uso del 5 % de una CPU de cuatro núcleos. Los eventos que se envían tienen que tener formato `json` (uno por línea, y escritos manualmente por el usuario).

Al igual que las dos herramientas anteriores, este servidor se centra en las consultas EPL de EsperTech. A pesar de permitir utilizar más consultas en las pruebas, estas son más limitadas. Hablando de los eventos a procesar, estos los tiene que escribir en formato `json` el usuario, una tarea un tanto tediosa si se quieren utilizar los 2000 eventos que permite la herramienta.

Como puede observarse, las herramientas existentes para probar el lenguaje EPL de EsperTech, se centran en las consultas, no en los eventos y/o en la definición de los tipos de eventos. El método que se propone en esta tesis se podría integrar con ellas para facilitarle al usuario el proceso de generación de eventos para las pruebas que desee realizar.

3.7. Definición de operadores de mutación

Ammann y Offutt establecieron en [123], que un aspecto clave para la adecuada aplicación de la prueba de mutaciones es la elección de los operadores de mutación, que deben ser diseñados de forma específica para cada lenguaje de programación. Hoy en día están publicados diversos trabajos en los que se definen operadores de mutación para diferentes lenguajes de programación y aplicaciones de la prueba de mutaciones. Esta sección revisa algunos de los trabajos que aparecen en la bibliografía relacionados con la definición de operadores de mutación y las herramientas que los incorporan.

En 1989, Agrawal y col. [124] definen un conjunto de 77 operadores de mutación para el lenguaje C. Estos operadores modelan diversos errores que suelen cometer los desarrolladores al programar en C y proporcionan cobertura parcial de caminos. Posteriormente Delamaro y col. [125] (1996) han construido la herramienta de generación de mutantes Proteum que implementa todos los operadores definidos en el trabajo anterior.

King y Offutt [126] construyen en 1991 la herramienta Mothra para la aplicación de la prueba de mutaciones al lenguaje Fortran, que incorpora un conjunto de 22 operadores de mutación.

Cinco años después, 1996, Offut y col. [127] definen un total de 65 operadores de mutación para el lenguaje Ada, que los clasifican en cinco categorías: operadores de reemplazamiento de operandos, operadores de sentencias, operadores de expresiones, operadores de cobertura y operadores de tareas.

El lenguaje Java también ha sido objeto de estudio de diversos trabajos para definir sus operadores de mutación. En 1999, Kim y col. [128] aplican la técnica HAZOP (Hazard and Operability Studies) a la definición de la sintaxis de Java para identificar desviaciones de las construcciones del lenguaje y, así, definir un conjunto de operadores de mutación para este lenguaje. Tres años más tarde, 2002, Alexander y col. [129] definen un conjunto de operadores para introducir mutaciones en los objetos de Java; estos operadores son utilizados por la herramienta OME (Object Mutation Engine) para generar mutantes. Ese mismo año, Ma y col. [130] definen 26 operadores de mutación para el lenguaje Java

basándose en una lista exhaustiva de los fallos que pueden darse en los programas orientados a objetos. Estos operadores han sido integrados en la herramienta MuJava [131] (2005). Bradbury y col. [132] (2006) definen un conjunto de operadores específicos para el comportamiento concurrente de Java (J2SE 5.0). En trabajos más recientes, Reales y Polo [133, 134] han desarrollado la herramienta de mutación Bacterio para Java, que incorpora diversas técnicas de optimización de la prueba de mutaciones.

En 2007, Tuya y col. [135] definen un conjunto de operadores de mutación para SQL que se integran en la herramienta SQLMutation para la generación de mutantes.

En los años 2005 y 2007, Derezińska [136, 137] define un conjunto de operadores de mutación para C#, comparándolos con los definidos previamente para C++ y Java y desarrolla la herramienta CREAM (CREAtor of Mutants) para automatizar la aplicación de la prueba de mutaciones a programas escritos en C#.

En un trabajo más reciente Delgado-Pérez y col. [138] proponen un conjunto de operadores de mutación de clases relacionados con C++ y su particular característica orientación a objetos.

Una de las primeras aplicaciones de la prueba de mutaciones fuera del ámbito de los lenguajes tradicionales fue la mutación de interfaces. Delamaro y col. [139] proponen la mutación de interfaces, una técnica que se aplica a los sistemas compuestos por varios módulos que interactúan y que permite evaluar cómo han sido probadas las interacciones entre dos o más unidades. Posteriormente, Ghosh y Mathur [140] proponen un método basado en interfaces para la prueba de aplicaciones distribuidas constituidas por componentes, definiendo operadores de mutación de interfaces para CORBA.

En 1999, Ammann y Black [141] proponen la aplicación de la prueba de mutaciones a las especificaciones formales. Senger de Souza y col. [142] la aplican a la validación de especificaciones Estelle (Extended Finite State Machine Language) [143], definiendo un conjunto de 38 operadores de mutación.

Offutt y col. [144, 145] proponen en los años 2004, 2005 un nuevo enfoque denominado perturbación de datos, el cual lo han aplicado a la prueba de servicios web (WS). Este nuevo enfoque está relacionado con la prueba de mutaciones, pero en vez de mutar programas modifica los valores de los datos incluidos en los mensajes que intercambian los WS, definiéndose nuevos operadores de mutación para la modificación de estos valores.

En el año 2008, Estero y col [146] definen un conjunto de 30 operadores de mutación para composiciones de servicios web en WS-BPEL 2.0. En este estudio se aplica la prueba de mutaciones a estas composiciones para la generación de conjuntos de casos de prueba de calidad.

Más recientemente han aparecido numerosos trabajos que definen operadores de mutación para diferentes modelos de programación. Adra y col. [147] (2010) han definido un conjunto de clases de operadores de mutación específicas para los modelos basados en agentes. Jagannath y col. [148] (2010) han introducido un conjunto de operadores de mutación para los programas basados en actores, clasificándolos en tres categorías: comunicación, restricciones y creación/borrado.

Bertolino y col. [149] (2013) han propuesto 9 operadores de mutación para el lenguaje XACML 2.0, un estándar de facto para especificar políticas de control de acceso. Asimismo, presentan la herramienta de generación de mutantes XACMUT, que integra los 9 operadores propuestos en este trabajo, los 11 definidos por Martin y Xie en [150] y 4 de los 5 operadores propuestos por Mouelhi y col. en [151].

Hasta la realización de esta tesis doctoral no se habían definido operadores de mutación para el lenguaje EPL, por lo que una de las aportaciones de este trabajo es la definición de un conjunto de operadores de mutación para este lenguaje y una herramienta que integra los operadores definidos y permite aplicar la prueba de mutaciones de forma automática a las consultas EPL.

3.8. Prueba de mutaciones en sistemas de tiempo real

La prueba de mutaciones puede ser aplicada en diferentes niveles para hacer pruebas del software: nivel de unidad, nivel de integración y nivel de especificación [54]. Hoy en día existen en la literatura diversos estudios donde se aplica la prueba de mutaciones en diferentes lenguajes de programación. Algunos de estos lenguajes de programación se les denominan lenguajes de programación tradicionales, que han ido evolucionando y que actualmente se utilizan para implementar sistemas de tiempo real. Tal y como se describió en la sección 3.7, ya están publicados los correspondientes estudios de prueba de mutaciones para muchos de estos lenguajes de programación: Java [131], C [124], Ada [127].

A continuación se describen otros trabajos donde se utiliza la prueba de mutaciones en sistemas de tiempo real con un ámbito diferente.

R. Nilsson y col. realizaron un estudio [152] donde proponen un criterio de prueba de mutaciones para analizar la puntualidad de sistemas concurrentes en tiempo real. Este criterio de prueba define los requisitos que debe satisfacer un programa una vez que este es probado (la medida de un test de cobertura indica la profundidad de satisfacción del criterio de prueba). Para ello definieron un conjunto de operadores de mutación para un Autómata Temporizado con Tareas (TAT), el cual modela diferentes aspectos de sistemas

en tiempo real. Los operadores de mutación los dividen en 2 categorías de faltas de puntualidad: suposiciones incorrectas sobre el comportamiento durante el análisis y el diseño de su organización, y la habilidad del sistema para soportar imprevistos no contemplados y cambios en el comportamiento del entorno. Las definiciones de estos operadores son la base de la técnica para medir los problemas de tiempo en los sistemas de tiempo real. Los operadores de mutación cambian el autómata para crear los mutantes, los cuales son analizados por un comprobador de modelos que intenta producir trazas de ejecución en las que se producen problemas de tiempo. Estas trazas se convierten en casos de prueba y se utilizan para comprobar si los problemas de tiempo pueden observarse en la implementación. En [153], R. Nilsson y col. proponen un método basado en modelos para generar casos de prueba para el análisis de puntualidad, utilizando simulaciones conducidas de forma heurística. Siguiendo la misma línea de trabajo en [154] evalúan un framework basado en mutaciones, que realiza pruebas de forma automática para analizar la puntualidad en sistemas de tiempo real activados por eventos. En concreto, el framework utiliza un subconjunto del TAT para especificar los supuestos de las aplicaciones bajo análisis. Para lograrlo se guardan el comienzo y el final de cada tarea invocada, las cuales tienen el registro de sus propios eventos. Dado que existe un registro de los eventos con sus marcas de tiempo, estos pueden consolidarse y analizarse tras la ejecución de una prueba.

El uso de la prueba de mutaciones en la presente tesis es para validar el método propuesto para la generación de eventos de prueba. Se quiere mostrar que los eventos generados a través del método pueden aplicarse para hacer diferentes tipos de prueba de software en sistemas IoT. En los trabajos que se mencionan en esta sección, se emplea la prueba de mutaciones para analizar la puntualidad de sistemas concurrentes y para evaluar un framework que realiza pruebas de forma automática. Tal y como puede observar en esta sección como en la sección 3.7, la prueba de mutaciones es una técnica muy popular y ha sido aplicada a multitud de lenguajes de programación.

Capítulo 4

Método para automatizar la generación de eventos para pruebas

Este capítulo introduce un método para automatizar la generación de eventos para pruebas. Este método se divide en tres principales etapas: definición del tipo de evento, validación de la definición y generación de eventos según la definición.

4.1. Motivación

Los programadores que desean hacer pruebas en programas que empleen un lenguaje de procesado de eventos (EPL), se encuentran los siguientes problemas:

1. No hay una cantidad de eventos suficiente para pruebas.
2. Se necesitan eventos con valores específicos.
3. Hay que esperar la generación de eventos por parte de la fuente de eventos.

Los tres puntos anteriores son claros problemas para los programadores que quieren probar programas que procesan eventos, y es lo que ha llevado a plantear un método para generar de forma automática eventos para pruebas. Los eventos para pruebas que se generen a partir de este método podrán ser utilizados en cualquier tipo de prueba de software: funcionales, no funcionales, de regresión, negativas, etc.

El comportamiento de este método es similar a los canales de información que incluyen las plataformas IoT que existen hoy en día. Según [11, 155] las plataformas IoT permiten gestionar la información proveniente de sensores localizados a lo largo de todo el planeta,

permitiendo que los usuarios y las aplicaciones autorizados puedan acceder a esta información para analizarlas y procesarlas. Estas plataformas son ideales para la obtención de volúmenes masivos de datos heterogéneos proporcionados por una gran cantidad de usuarios, organizaciones y compañías a nivel mundial, sin el requerimiento de grandes inversiones ni en software ni en hardware por parte de los interesados en estos datos.

Las principales diferencias, y a su vez ventajas, entre el método propuesto y las plataformas IoT son las siguientes:

1. El usuario puede elegir (sin límite) la cantidad de eventos a obtener.
2. El usuario define el tipo de evento. Se puede personalizar el tipo de evento adaptándose a las necesidades del programador.
3. El usuario puede generar eventos con valores específicos.
4. El usuario puede obtener de forma inmediata los eventos para las pruebas gracias a la automatización del método.

Estas características permiten que, a través del método propuesto, se puedan generar eventos para cualquier tipo de prueba de software.

4.2. Etapas del método

El método que se propone y presentado en [156, 157], se divide en etapas (véase la figura 4.1), desarrolladas del siguiente modo: en la etapa de *definición* el *tipo de evento* del que se desea generar una cantidad de eventos se define de manera manual por el usuario siguiendo unas pautas. Esta definición será analizada en la etapa de *validación* y, si esta es correcta, en la última etapa de *generación de eventos* se obtendrá la cantidad de eventos solicitada, en el formato de salida indicado.

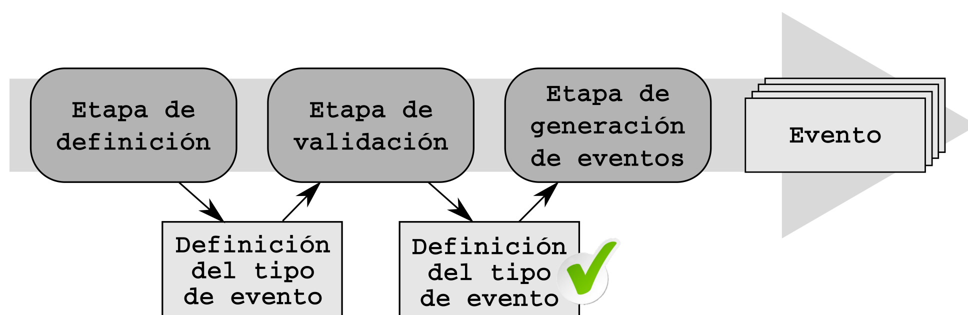

FIGURA 4.1: Etapas del método.

En la primera etapa, el usuario escribirá en un fichero la definición del tipo de evento cumpliendo las indicaciones de la propuesta de especificación para la definición de cualquier tipo de evento. La segunda etapa consiste en validar que la definición del tipo de evento que ha escrito el usuario, cumple todas las indicaciones explicadas en la propuesta de especificación. Se sigue un recorrido por toda la estructura de la definición del tipo de evento contemplando todos elementos y las propiedades utilizados en la misma. Finalmente, si la definición es válida, se pasa a la etapa de generación de eventos, donde se generan tantos eventos del tipo de evento definido por el usuario como este indique.

En los siguientes capítulos de la presente tesis se describirán con detalle cada una de las etapas, así como sus componentes, mostrando ejemplos de su uso. En el capítulo 5 se profundiza en la primera etapa, en la que se introduce una propuesta de especificación para la definición de cualquier tipo de evento. En el capítulo 6 se habla de las siguientes etapas: la *validación* de la definición y la *generación de eventos* según la definición, donde se presenta una herramienta que engloba ambas etapas. Tras la presentación del método y sus componentes, se realizan diversas pruebas con el objetivo de validar la efectividad del método propuesto (los resultados se muestran en el capítulo 9). En primer lugar se comprueba si a través del método pueden generarse eventos como los que se obtienen en las plataformas IoT estudiadas. En concreto se comparan los eventos generados a través del método con los eventos reales de la plataforma ThingSpeak [109]. Por otro lado, se utilizan los eventos generados a través de este método en varios casos de estudio aplicando la prueba de software prueba de mutaciones. Para poder aplicar la prueba de mutaciones se necesita definir un conjunto de operadores de mutación para el lenguaje de programación que se esté empleando (EPL de EsperTech), estos se encuentran en el capítulo 7. Finalmente, se tiene que desarrollar una herramienta que permita generar los mutantes, ejecutarlos y comparar sus salidas frente al programa original; herramienta presentada en el capítulo 8.

4.3. Conclusiones

Se ha presentado la propuesta de un método para automatizar la generación de eventos para pruebas. Según las características del mismo, este método permite generar eventos para cualquier tipo de prueba software que se quiera hacer. Las diferencias del método propuesto frente a los canales de información que incluyen las plataformas IoT son también sus principales ventajas: no hay límite en la generación de los eventos de prueba, el usuario puede definir la estructura del tipo de evento, se pueden asignar valores concretos a los atributos de los eventos y no hay que esperar a la fuente de eventos.

El método consta de tres etapas bien diferenciadas: definición, validación y generación de eventos, las cuales se detallarán en los siguientes capítulos de la presente tesis. Así mismo se mostrarán los resultados obtenidos tras la realización de una serie de pruebas con los eventos generados a través del método propuesto en el capítulo 9 de la presente tesis.

Capítulo 5

Propuesta de especificación para la definición de tipos de eventos

En este capítulo se presenta y se detalla la primera de las etapas del método presentado en el capítulo 4; una propuesta de especificación para la definición de cualquier tipo de evento que sea gestionado por un programa que emplee cualquier lenguaje de procesado de eventos (EPL). Esta propuesta de especificación facilita que cualquier programador defina la estructura del tipo de evento implicado en su programa, para que luego pueda ser estudiada, analizada y utilizada por cualquier otro programador o herramienta. Concretamente, se presenta una posible estructura, a través del lenguaje de marcado XML, para la definición de los tipos de eventos y cada uno de los atributos que lo conforman.

5.1. Estructura de la especificación

Luckham [93] en el año 2012 propone las siguientes pautas a seguir por un estándar para la representación de eventos:

- Todos los eventos tienen que ser instancia de un tipo de evento.
- La estructura de los eventos se define por el tipo al que pertenece.
- La estructura se representa como una colección de atributos del evento.

Por otro lado, se recomienda el uso de un lenguaje de programación fuertemente tipado (como XML Schema o Java). También se indica que cualquier estándar, a la hora de representar eventos, deberá especificar ciertos datos predefinidos (atributos), como por ejemplo:

- Un identificador único para el evento.
- El tipo de evento.
- La marca de tiempo de la creación del evento.
- La fuente de creación del evento.

Además, ha de tenerse en cuenta que los atributos de un evento pueden ser de tipo simple o de tipo complejo.

Siguiendo las pautas y recomendaciones de Luckham, se escoge XML como lenguaje para definir los tipos de eventos. XML es un lenguaje fuertemente tipado del cual existe una gran cantidad de herramientas y bibliotecas que simplifican su proceso de lectura y análisis.

En la figura 5.1 se presentan los diferentes componentes de la definición de tipo de evento, así como las propiedades que tendrán y que se proponen en esta tesis.

En concreto, se propone que cada tipo de evento (`event_type`) esté formado por un conjunto de bloques (`block`). Cada uno de ellos, a su vez, se componga de una serie de campos, pudiendo ser de tipo opcional (`optionalfields`) o no (`field`). Los campos de tipo opcional se componen de campos, y estos últimos pueden estar compuestos por atributos. Los elementos campos pueden contener elementos de tipo campo (solo hasta un nivel de recursividad). Estos elementos anidados (véase sección 5.2.2.1) no pueden definirse dentro de los campos opcionales.

Dado que el lenguaje utilizado para la especificación es XML, se utilizarán etiquetas y atributos para definir cada uno de los elementos del tipo de evento. Para que la estructura XML de la definición del tipo de evento pueda ser validada con cualquier validador de este lenguaje de marcado, la primera línea que ha de aparecer es la cabecera con la versión y codificación (véase código 5.1).

```
<?xml version="1.0" encoding="UTF-8"?>
```

Código 5.1: Cabecera de la especificación.

Tras la cabecera, tenemos que indicar el tipo de evento que vamos definir. Esto se hace a través de la etiqueta `<event_type>`, en la que se indica el tipo de evento a través del atributo `name` de la misma. En el siguiente ejemplo, código 5.2, se define el tipo de evento `EventoTerminal`.

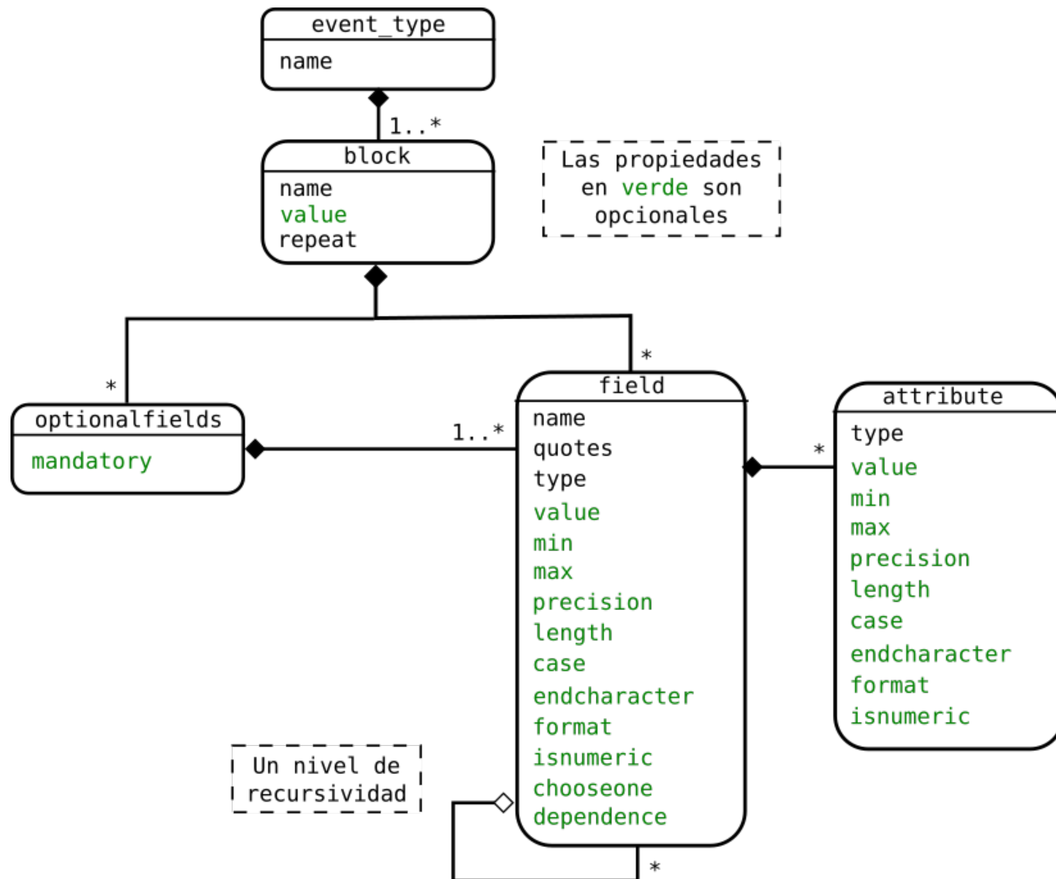

Figura 5.1: Componentes y propiedades de la definición de tipo de evento.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  ...
</event_type>
```

Código 5.2: Tipo de evento.

5.1.1. Bloque

La etiqueta `<event_type>` se compone de bloques, que se definen con la etiqueta `<block>`. Pueden utilizarse tantos bloques como se quiera, pero uno de ellos tiene que indicar el número de eventos que se desea de ese tipo. Esto se indica con el atributo `repeat` de la etiqueta `<block>`. Uno de los posibles usos de los bloques que no contengan el atributo `repeat`, es el de cabecera. Si se quiere indicar algún tipo de información adicional, esta puede especificarse con el atributo `value` de la etiqueta `<block>`, o a través del componente campo (véase 5.1.2).

Siguiendo con el ejemplo del tipo de evento `EventoTerminal`, el código 5.3 muestra cómo utilizar los atributos de la etiqueta `<block>` (`repeat` y `value`).

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="cabecera" value="Date:2016;Description:...">
  </block>
  <block name="valores" repeat="150">
  ...
  </block>
</event_type>
```

Código 5.3: Componentes del evento: bloques.

El atributo `value` puede contener cualquier tipo de información que se considere necesaria para la definición del evento: estructura, fecha de creación, identificador... El valor del atributo `repeat` será un número que determinará el número de eventos que se desea del tipo definido. Otro de los atributos de las etiquetas es `name`, que sirve para identificar el nombre de cada bloque.

5.1.2. Campo y Campos Opcionales

Los bloques se componen de campos, algunos de los cuales pueden ser opcionales, que se definen con las etiquetas `<field>` y `<optionalfields>`, respectivamente. Cada campo `<field>` define cada uno de los elementos de los que se compone el evento; es por esto por lo que el atributo `name` de esta etiqueta siempre debe de incluirse. Los valores de este atributo será cada uno de los nombres de los atributos de eventos que componen el evento. Del mismo modo, otro de los atributos que siempre ha de incluirse es `type`, con el cual se define el tipo del elemento: tipo complejo o tipo simple (véase sección 5.2). Y finalmente, otro de los atributos que deben incluirse es, `quotes` cuyos posibles valores son {`true`, `false`}, que indica si los valores del elemento que se define está entrecomillado o no.

En el ejemplo mostrado en el código 5.4 se definen los elementos del tipo de evento `EventoTerminal`, tanto de tipo complejo como de tipo simple.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="cabecera" value="Date:2016;Description:...">
  </block>
  <block name="valores" repeat="150">
    <field name="estado" quotes="true" type="ComplexType" ...>
    ...
    </field>
    <field name="terminal_id" quotes="false" type="Integer" ...></field>
    <field name="timestamp" quotes="false" type="Long" ...></field>
  </block>
</event_type>
```

Código 5.4: Definición de los elementos del tipo de evento: etiqueta field.

En el código 5.4, se indica que el tipo de evento `EventoTerminal` se compone de los elementos: `estado`, `terminal_id` y `timestamp`. El elemento `estado` es de un tipo complejo ("ComplexType") y su valor va entrecomillado, y los elementos `terminal_id` y `timestamp` son de tipo simple: Integer y Long respectivamente, y sus valores no van entrecomillados.

En algunos eventos, ciertos atributos de evento no aparecen siempre; este tipo de elementos se definen con las etiquetas `<optionalfields>`. Los componentes de `<optionalfields>` son campos, `<field>`, que pueden ser de tipo simple o de tipo complejo. Cuando se genere el evento, solo podrá aparecer uno de los campos definidos dentro de la etiqueta `<optionalfields>`. La etiqueta `<optionalfields>` posee un atributo opcional, `mandatory`, que indica la posibilidad de que no aparezca ningún campo de los definidos dentro de la etiqueta `<optionalfields>` cuando su valor es `false`, es decir, que su valor por defecto es `true`.

El código 5.5 es un fragmento de la definición del tipo de evento `EventoTerminal` en el que se utiliza la etiqueta `<optionalfields>`. Según la definición del tipo de evento `EventoTerminal`, este se compone de los elementos: `estado`, `terminal_id` y `timestamp`, y de los elementos opcionales: `responId`, `numequipaje` y `destinoId`. Según la definición, las posibles estructuras de los eventos de tipo `EventoTerminal` son:

1. {`estado`, `terminal_id`, `timestamp`}
2. {`estado`, `terminal_id`, `timestamp`, `responId`}
3. {`estado`, `terminal_id`, `timestamp`, `numequipaje`}

4. {estado, terminal_id, timestamp, destinoId}

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="cabecera" value="Date:2016;Description:...">
  </block>
  <block name="valores" repeat="150">
    <field name="estado" quotes="true" type="ComplexStatus" ...>
    ...
    </field>
    <field name="terminal_id" quotes="false" type="Integer" ...></field>
    <field name="timestamp" quotes="false" type="Long" ...></field>
    <optionalfields mandatory="false">
      <field name="responId" quotes="true" type="Integer" ...></field>
      <field name="numequipaje" quotes="false" type="Integer" ...></field>
      <field name="destinoId" quotes="true" type="String" ...></field>
    </optionalfields>
  </block>
</event_type>
```

Código 5.5: Etiqueta optionalfields.

Se puede utilizar la etiqueta `<optionalfields>` tantas veces como se necesite. En la definición del tipo de evento `EventoTerminal`, mostrada en el código 5.6, se puede ver otro ejemplo con la etiqueta `<optionalfields>`. Según la definición del tipo de evento `EventoTerminal`, las posibles estructuras de los eventos de tipo `EventoTerminal` son:

1. {estado, terminal_id, timestamp, responId}
2. {estado, terminal_id, timestamp, destinoId}
3. {estado, terminal_id, timestamp, responId, numequipaje}
4. {estado, terminal_id, timestamp, destinoId, numequipaje}

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="cabecera" value="Date:2016;Description:...">
  </block>
  <block name="valores" repeat="150">
    <field name="estado" quotes="true" type="ComplexStatus" ...>
    ...
    </field>
    <field name="terminal_id" quotes="false" type="Integer" ...></field>
    <field name="timestamp" quotes="false" type="Long" ...></field>
    <optionalfields>
      <field name="responId" quotes="true" type="Integer" ...></field>
      <field name="destinoId" quotes="true" type="String" ...></field>
    </optionalfields>
    <optionalfields mandatory="false">
      <field name="numequipage" quotes="false" type="Integer" ...></field>
    </optionalfields>
  </block>
</event_type>
```

Código 5.6: Otro uso de la etiqueta optionalfields.

5.2. Tipos de Datos

Como se menciona en [93], la estructura de los eventos se define por los tipos de datos de los atributos de evento que contienen. Los tipos de datos que se van a describir contienen una serie de atributos que se pueden dividir en dos grupos: los que *modifican la forma* del tipo de dato (los que indican si una cadena de caracteres se escribe en mayúsculas o en minúsculas, si la separación de los dígitos de una fecha son "-" o "/", etc) y los que *modifican el valor* del tipo de dato (los que indican si una cadena de caracteres va a tener una longitud de 3 o 6 caracteres, si un número estará entre el rango de números 15 y 25, etc). La funcionalidad de cada uno de ellos determina a qué grupo pertenece.

Primero se han descrito los tipos de datos básicos (o simples) y los atributos de los mismos, y luego los complejos, que son posibles combinaciones de los básicos. De ambos tipos de datos se muestra no solo cómo han de definirse sino también varios ejemplos para mostrar la potencialidad de la propuesta de especificación, presentada en esta tesis, para definir eventos.

5.2.1. Tipos Simples

Los tipos simples que se han contemplado son los tipos básicos más comunes que pueden aparecer en cualquier lenguaje de programación tradicional. Se han definido:

1. **Entero**, identificados por `Integer`.
2. **De punto flotante**, identificados por `Float`.
3. **Entero grande**, identificados por `Long`.
4. **Cadena de caracteres**, identificados por `String`.
5. **Alfanumérico**, identificados por `Alphanumeric`.
6. **Booleano**, identificados por `Boolean`.
7. **Fecha**, identificados por `Date`.
8. **Tiempo**, identificados por `Time`.

La definición de los tipos simples se realiza asignándole al atributo `type` del campo `<field>` el identificador correspondiente. En el código 5.7 se puede observar cómo se definen elementos de tipo simple, en concreto de tipo `Integer` y `String`.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true" type="Integer" ...></field>
    <field name="timestamp" quotes="false" type="String" ...></field>
  </block>
</event_type>
```

Código 5.7: Definición de un tipo simple.

En los siguientes apartados se presentan cada uno de los diferentes tipos de datos simples y los atributos que pueden tomar. Antes de profundizar en cada uno de los atributos, hay que indicar que un atributo común a todos es `value`. Si este parámetro tiene asignado un valor, este es el que tendrá asignado el elemento definido en `<field>`. En el ejemplo que se muestra en código 5.8 se utiliza este atributo.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true" type="Integer" value="95043">
    </field>
    <field name="destino" quotes="false" type="String" value="Madrid">
    </field>
  </block>
</event_type>
```

Código 5.8: Uso del atributo value.

El elemento `terminal_id` del ejemplo código 5.8, va a tener siempre el valor "95043" (obsérvese que `quotes` tiene valor `true`) y el elemento `destino` va a tener siempre el valor Madrid.

5.2.1.1. Tipo Entero

El valor que tomarían por defecto los elementos a los que se les asigne el tipo entero (sin incluir ningún atributo), es un número comprendido entre 0 y 9. Este valor puede modificarse si se indica con los atributos `min` y `max`. Con esos atributos definimos el rango numérico que puede tomar el elemento, ambos atributos deben aparecer conjuntamente.

En el ejemplo de código 5.9, el elemento `terminal_id` puede adquirir un valor numérico comprendido en el rango de valores [100,999].

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true"
                              type="Integer" min="100" max="999"></field>
  </block>
</event_type>
```

Código 5.9: Uso del tipo entero.

5.2.1.2. Tipo Punto Flotante

Al igual que los tipos enteros, los tipos de punto flotante tienen como valor por defecto un número comprendido en el rango [0,9.9]. Este valor puede modificarse con los atributos `min` y `max`. Con esos atributos definimos el rango numérico que puede tomar el elemento, ambos atributos deben aparecer conjuntamente.

Además, el tipo punto flotante cuenta con el atributo `precision` que define el número de decimales que se quiere asignar al elemento. El atributo `precision` puede utilizarse sin necesidad de que estén definidos los atributos `min` y `max`. Esto nos daría un número comprendido en el rango [0,9.9], pero con tantos decimales como se indique en `precision`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="temp" quotes="true" type="Float"
                            min="5.0" max="30.9" precision="2"></field>
    <field name="humd" quotes="true" type="Float" precision="3"></field>
  </block>
</event_type>
```

Código 5.10: Uso del tipo punto flotante.

El valor que tomaría el elemento `temp` en el ejemplo de código 5.10, sería un valor con dos decimales comprendido en [5,30.9] y el valor del campo `humd` sería un valor con tres decimales, comprendido en el rango de valores [0,9.9].

5.2.1.3. Tipo Entero Grande

Dado que el tipo entero tiene sus limitaciones en cuanto la longitud/capacidad que puede tomar, también se considera el tipo entero grande. Este tiene los mismos atributos que el tipo entero (véase sección 5.2.1.1), `min` y `max`. En cuanto a los valores por defecto, se obtiene un valor comprendido en el rango [0,9], al igual que el tipo entero.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="timestamp" quotes="false" type="Long"
                min="1000000000000" max="9999999999999"></field>
  </block>
</event_type>
```

Código 5.11: Uso del tipo entero grande.

El campo `timestamp` que se contempla en código 5.11, va a tener un valor comprendido en el rango [1000000000000,9999999999999].

5.2.1.4. Tipo Cadena de Caracteres

El tipo cadena de caracteres tiene el atributo `length`, que determina la longitud que debe tener la cadena. Si este atributo no se especifica, la longitud de la cadena será por defecto 10 caracteres. La cadena de caracteres que genera (por defecto) tiene las letras en mayúsculas. Para generar letras minúsculas se tendrá que utilizar el atributo `case` y asignarle el valor `low`. Por otro lado, si se quiere una cadena de caracteres hasta un determinado carácter, se puede determinar con el atributo `endcharacter`. A este atributo se le ha de asignar la letra correspondiente hasta la que se quiera obtener la cadena. En el ejemplo de código 5.12 se muestra un ejemplo del uso de todos los atributos.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoCoche">
  <block name="valores" repeat="150">
    <field name="pais" quotes="true" type="String" value="ES"></field>
    <field name="num_matr" quotes="true" type="Integer"
      min="1000" max="9999"></field>
    <field name="letr_matr" quotes="true" type="String" length="3"</field>
    <field name="id_matr" quotes="true" type="String" case="low"
                                                    length="4"></field>
    <field name="calidad" quotes="false" type="String" endcharacter="D"
                                                    length="1"></field>
  </block>
</event_type>
```

Código 5.12: Uso del tipo cadena de caracteres.

Según el ejemplo que se muestra en código 5.12, el valor del campo `pais` será siempre "ES", ya que el atributo `value` está definido y tiene asignado ese valor. Para el elemento `letr_matr` se obtendría una cadena de caracteres entrecomillada de longitud 3 y para `id_matr` la cadena de caracteres, también entrecomillada, será de longitud 4 pero con las letras en minúsculas. Finalmente, para el campo `calidad` se obtiene una letra que esta en el rango [A,D].

5.2.1.5. Tipo Alfanumérico

Aunque este tipo se podría haber obviado dado que se puede formar un tipo complejo con la combinación de los tipos entero y cadena de caracteres, tras analizar varios tipos de eventos se observó que su uso era frecuente. Para facilitar al usuario la definición del tipo de evento se incluye este tipo de dato como un tipo de dato básico.

Este tipo simple tiene los mismos atributos que el tipo cadena de caracteres (véase sección 5.2.1.4), {`value, length, endcharacter, case`}. Sus funcionalidades son las mismas que para el tipo cadena de caracteres.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoColor">
  <block name="valores" repeat="150">
    <field name="cod_hex" quotes="true" type="Alphanumeric" length="6"
                                          endcharacter="F"></field>
  </block>
</event_type>
```

Código 5.13: Uso del tipo alfanumérico.

El código 5.13 muestra un ejemplo del uso de este tipo de dato. Para el tipo de evento `EventoColor` se define un campo, `cod_hex`, cuyos valores van a ser alfanuméricos. Estas cadenas alfanuméricas tienen una longitud de 6 caracteres/dígitos y el valor máximo al que llegará será la F. De esta forma se consigue definir el color del evento con un valor hexadecimal.

5.2.1.6. Tipo Booleano

Al igual que el tipo alfanumérico podría construirse como una conexión de tipos enteros y cadena, el tipo booleano se puede solventar con un tipo entero, indicando que su valor máximo es 1 y el mínimo 0, pero dado que el tipo booleano se asocia tanto a los valores

{true, false}, como a {0, 1}, se ha considerado conveniente contemplar este tipo como uno básico. El tipo booleano tiene por defecto que sus valores sean de la forma {true, false}, si se quiere que el valor sea binario {0, 1}, se tiene que indicar asignándole al atributo `isnumeric` el valor `true`.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true" type="Integer"
                                              min="100" max="999"></field>
    <field name="activo" quotes="false" type="Boolean"
                                                 isnumeric="true"></field>
  </block>
</event_type>
```

Código 5.14: Uso del tipo booleano.

El campo `activo` que aparece en código 5.14, tendrá como valores {0, 1}, ya que el atributo `isnumeric` tiene asignado el valor `true`. Según el ejemplo, este campo nos indicaría si la terminal correspondiente está activa o no.

5.2.1.7. Tipo Fecha

El tipo fecha también podría formarse con un tipo complejo combinando el tipo entero y el tipo cadena. Pero dado que tiene varias restricciones y este tipo de dato aparece, no solo en la mayoría de los lenguajes de programación, sino también en la mayoría de los tipos de eventos, se incluye este tipo de dato como uno básico.

El atributo que incluye este tipo de dato es `format` de carácter obligatorio, que permite especificar el modo en el que se quiere mostrar la fecha. Este atributo acepta como valores una cadena que determina el formato y los valores que se quieren obtener. Los valores se definen siguiendo los patrones definidos en Java para la clase `SimpleDateFormat`, véase tabla 5.1[1]. Estos valores también se emplean para el tipo de dato Tiempo (véase sección 5.2.1.8).

En el código 5.15 vemos el uso de este tipo de dato, donde el campo `día` de tipo fecha, tiene asignado en el atributo `format` la cadena `yy-MM-DD`, donde se indica que la fecha estará separada por guiones y se mostrará primero el año (dos dígitos), luego el mes y finalmente el día.

[1]Extracto de tabla obtenida de `https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html`

Letra	Componente de tipo Fecha o Tiempo
G	Designación de la era
y	Año
Y	Semana del año (semanas completas)
M	Mes del año
w	Semana del año
D	Día del año
d	Día del mes
F	Día de la semana en el mes
E	Nombre del día en la semana
u	Número del día de la semana
a	Marcador AM/PM
H	Hora del día (0-23)
k	Hora del día (1-24)
K	Hora en AM/PM (0-11)
h	Hora en AM/PM (1-12)
m	Minutos
s	Segundos
S	Milisegundos
z	Zona de tiempo (General)
Z	Zona de tiempo (RFC 822)
X	Zona de tiempo (ISO 8601)

Tabla 5.1: Patrones para los tipos Fecha y Tiempo.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="temp" quotes="true" type="Float"
                            min="5.0" max="30.9" precision="2"></field>
    <field name="humd" quotes="true" type="Float" precision="3"></field>
    <field name="dia" quotes="true" type="Date" format="yy-MM-DD"></field>
  </block>
</event_type>
```

Código 5.15: Uso del tipo fecha.

5.2.1.8. Tipo Tiempo

La definición del tipo Fecha propuesta en la sección 5.2.1.7 es más general puesto que permite especificar tanto datos de fecha como datos de tiempo. Pero en muchos tipos de eventos se ha observado que el tiempo se puede presentar de forma separada, por lo

que para facilitar la labor del programador, se ha optado por contemplar este dato por mayor claridad.

El atributo que adquiere este tipo de dato es `format`, un atributo obligatorio (igual que para el tipo Fecha). Tiene la misma funcionalidad, es decir, especificar el modo en el que se quiere mostrar el tiempo. Los valores se definen con los patrones de la clase de Java *SimpleDateFormat* (véase tabla 5.1).

```
 <?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true" type="CodeTerminal">
    ...    </field>
    <field name="destino" quotes="false" type="CodeCountry">
    ...    </field>
    <field name="salida" quotes"false" type="Time" format="HH:mm">
    </field>
  </block>
</event_type>
```

Código 5.16: Uso del tipo tiempo.

El campo `salida` en el código 5.16, es de tipo tiempo y tiene asignado el atributo `format`, que indica que este campo mostrará las horas y los minutos separados por el carácter ":".

5.2.2. Tipos Complejos

Los tipos de datos complejos se forman combinando tipos de datos simples, estos pueden ser de cualquiera de los tipos simples que se han descrito en la sección 5.2.1.

Una definición de tipo de dato complejo, se muestra en el código 5.17. En el atributo `type` en vez de aparecer asignado un tipo de dato simple, aparece la palabra clave `ComplexType`. A continuación se definen cada uno de los tipos simples que formen al tipo complejo mediante la etiqueta `<attribute>`, que adquirirá los atributos correspondientes según el tipo de dato que se quiera utilizar. Las etiquetas `<attribute>` no tienen los atributos `name` ni `quotes`, dado que forman parte de un tipo complejo que ya los posee.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true" type="Integer" min="0"
                                                          max="99"></field>
    <field name="creado" quotes="true" type="ComplexType">
      <attribute type="Date" format="yyyy-MM-dd"></attribute>
      <attribute type="String" value="T"></attribute>
      <attribute type="Time" format="hh:mm:ss"></attribute>
      <attribute type="String" value="Z"></attribute>
    </field>
  </block>
</event_type>
```

CÓDIGO 5.17: Definición de un tipo complejo.

En el código 5.17, el campo `creado` es de tipo complejo (`ComplexType`), está formado por un tipo de dato fecha, dos de tipo cadena y otro de tipo tiempo. Los tipos cadena tienen un valor fijo, y los tipos fecha y tiempo son variables. El campo `creado` que es de tipo de tipo complejo, podrá tener valores como "`2016-07-12T12:30:45Z`" (entrecomillado porque `quotes` del campo `creado` está a `true`).

5.2.2.1. Tipos Anidados

En un tipo complejo se pueden definir elementos de tipo `<attribute>` dentro de una etiqueta `<field>`. Pero también es posible que en lugar de dar una definición de tipo simple se pueda definir otro campo. Para ello el tipo complejo en lugar de tener elementos de tipo `<attribute>`, empleará elementos de tipo `<field>`.

Un ejemplo de definición de tipo de dato complejo anidado se observa en el código 5.18 con la definición del campo `lugar`. La definición de `lugar` se realiza igual que la definición de un tipo complejo compuesto por tipos simples (véase código 5.17). La diferencia es que este tipo complejo se compone de etiquetas `<field>` en vez de `<attribute>`. Como cualquier etiqueta `<field>`, estas contienen los mismos atributos obligatorios {`name`, `quotes` y `type`}, más los atributos correspondientes al tipo de dato simple del elemento.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoLugar">
  <block name="valores" repeat="150">
    <field name="lugar" quotes="false" type="ComplexType">
      <field name="nombre" quotes="true" type="String" length="4"></field>
      <field name="latitud" quotes="false" type="Float" precision="2">
      </field>
      <field name="longitud" quotes="false" type="Float" precision="2">
      </field>
    </field>
  </block>
</event_type>
```

CÓDIGO 5.18: Definición de un tipo complejo anidado.

El tipo anidado `lugar` del código 5.18 se compone de los elementos {`nombre, latitud` y `longitud`}, luego cada evento de tipo `EventoLugar` viene determinado por `lugar{nombre, latitud, longitud}`. En el código 5.19 vemos un ejemplo en formato `JSON` de eventos tipo `EventoLugar` según la definición del código 5.18.

```
{"lugar":{"nombre":"RSAW","latitud":6.69,"longitud":1.11},
"lugar":{"nombre":"ADOC","latitud":2.98,"longitud":8.51},
"lugar":{"nombre":"IKJN","latitud":3.07,"longitud":4.75}}
```

CÓDIGO 5.19: Eventos de ejemplo en formato JSON para el código 5.18.

La principal diferencia que existe entre utilizar elementos `<attribute>` y elementos `<field>`, se observa a la hora de la generación de los eventos. Los tipos complejos definidos con elementos `<field>`, se generarán imprimiendo los valores de los atributos `name` (tal y como se observa en el código 5.19), al contrario que si se utilizan elementos `<attribute>` (tal y como puede verse en el código 5.21). El uso o no de un elemento u otro, dependerá de la estructura que tenga el tipo de evento. Para visualizar mejor las diferencias de usar elementos `<attribute>` y elementos `<field>` en un tipo anidado, se muestra en el código 5.20 la definición del tipo `EventoLugar` empleando elementos `<attribute>`, y el el código 5.21 ejemplos en formato `JSON` de eventos tipo `EventoLugar` según la definición usando elementos `<attribute>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoLugar">
  <block name="valores" repeat="150">
    <field name="lugar" quotes="false" type="ComplexType">
      <attribute type="String" length="4"></attribute>
      <attribute type="Float" precision="2"></attribute>
      <attribute type="Float" precision="2"></attribute>
    </field>
  </block>
</event_type>
```

CÓDIGO 5.20: Definición del tipo complejo `EventoLugar` usando elementos `<attribute>`.

```
{"lugar":RSAW6.691.11},
"lugar":ADOC2.988.51},
"lugar":IKJN3.074.75}}
```

CÓDIGO 5.21: Eventos de ejemplo en formato JSON para el código 5.20.

Tal y como se ha mencionado antes, el uso de una definición u otra para un tipo complejo, dependerá de cómo sea la estructura del tipo de evento. Si el tipo de evento `EventoLugar` sigue la estructura utilizada en el código 5.20, pero cada elemento está delimitado con un carácter o símbolo, este deberá incluirse en la definición. Se muestra un ejemplo de definición usando caracteres delimitadores en el código 5.22, y en el código 5.23 ejemplos de eventos en formato `JSON` de dicha definición.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoLugar">
  <block name="valores" repeat="150">
    <field name="lugar" quotes="false" type="ComplexType">
      <attribute type="String" length="4"></attribute>
      <attribute type="String" value="_"></attribute>
      <attribute type="Float" precision="2"></attribute>
      <attribute type="String" value="#"></attribute>
      <attribute type="Float" precision="2"></attribute>
    </field>
  </block></event_type>
```

CÓDIGO 5.22: Definición de un tipo complejo usando caracteres delimitadores.

```
{"lugar":RSAW_6.69#1.11},
"lugar":ADOC_2.98#8.51},
"lugar":IKJN_3.07#4.75}}
```

Código 5.23: Eventos de ejemplo en formato JSON para el código 5.20.

5.2.2.2. Atributos opcionales para tipos complejos

Los atributos que se han comentado en la sección 5.1.2 para los campos son obligatorios para la definición del elemento que compone el tipo de evento a definir. Los que se ven en esta sección son opcionales y solo pueden utilizarse en definiciones de elementos de tipo complejo. Este tipo de atributo, según aparezcan o no en la definición del evento, condicionará el valor del elemento, pero no su tipo y/o forma.

Atributo chooseone

Este atributo por defecto tiene el valor `false`, es decir, solo cuando se le asigna el valor `true` tiene efecto. Su comportamiento es parecido a los `<optionalfields>`, pero con las siguientes diferencias: se aplica a campos de tipo complejo y no se puede usar el atributo `mandatory`.

Dado que en un tipo complejo (véase sección 5.2.2) se definen los elementos {`<field>`, `<attribute>`} que componen el tipo complejo, el uso del atributo `chooseone` hace que los elementos que lo componen {`<field>`, `<attribute>`} sean excluyentes, es decir, solo uno de ellos determinará el valor del elemento al que se aplica el atributo `chooseone`.

Siguiendo con el ejemplo del tipo de evento `EventoTerminal`, en el código 5.24, se aplica el atributo `chooseone`. En la definición del tipo de evento `EventoTerminal`, al utilizar el atributo `chooseone`, se establece que del tipo complejo solo uno de los elementos simples que lo componen {`FueraDeServicio, Checkin, Cancelado, Completo`} puede ser el valor de `estado`. El elemento `estado` siempre tendrá un valor, asignado de forma aleatoria, de los especificados en el tipo complejo. Se asignan de forma aleatoria para simular la aleatoriedad existente en el mundo real a la hora de ocurrir los eventos (al igual que hacen los generadores de eventos estudiados).

Si se quieren hacer pruebas con un valor específico, siguiendo el ejemplo del tipo de evento `EventoTerminal`, se tendría que definir este únicamente con la definición del elemento `<attribute>` que se desee conservar. No haría falta el atributo `chooseone` ya que siempre se escogerá el elemento `<attribute>` definido.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="estado" quotes="true" type="ComplexType"
                                                        chooseone="true">
      <attribute type="String" value="FueraDeServicio"></attribute>
      <attribute type="String" value="Checkin"></attribute>
      <attribute type="String" value="Cancelado"></attribute>
      <attribute type="String" value="Completo"></attribute>
    </field>
    <field name="terminal_id" quotes="false" type="Integer" min="0"
                                                  max="99"></field>
  </block>
</event_type>
```

Código 5.24: Uso del atributo chooseone.

Si se observa el ejemplo del código 5.25 donde se aplica el atributo `chooseone` pero con un tipo complejo anidado, los valores que obtiene del tipo complejo `lugar`, no contempla los tres elementos como en el código 5.18. Ahora el tipo complejo `lugar` solo vendrá determinado por uno de ellos. Cada vez que se genere un evento de este tipo de evento, se escogerá de forma aleatoria uno de los tres elementos; unas veces será `nombre`, otras `latitud` y otras `longitud`.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoLugar">
  <block name="valores" repeat="150">
    <field name="lugar" quotes="false" type="ComplexType"
                                                        chooseone="true">
      <field name="nombre" quotes="true" type="String" length="4">
      </field>
      <field name="latitud" quotes="false" type="Float" precision="2">
      </field>
      <field name="longitud" quotes="false" type="Float" precision="2">
      </field>
    </field>
  </block></event_type>
```

Código 5.25: Uso del atributo chooseone con un tipo anidado.

Atributo dependence

El valor de este atributo por defecto es `false`, al igual que el atributo `chooseone`, solo cuando se le asigna el valor `true` tiene efecto. Se asigna a campos de tipo complejo, incluyendo los tipos complejos anidados, y su aplicación puede afectar a más de dos campos de tipo complejo.

Tal y como se ha comentado, en un tipo complejo se definen los elementos {`<field>`, `<attribute>`} que componen el tipo complejo. Si se asigna a un tipo complejo A el atributo `dependence` con el valor a `true`, quiere decir que otro tipo complejo B depende de este. Lo cual implica que el elemento simple {`<field>`, `<attribute>`} que se escoja para determinar el valor del tipo complejo A, determinará el valor del tipo complejo B. Esto se consigue por la posición; la posición del elemento simple que se escoja para el tipo complejo A será la posición del elemento simple que se escoja para el tipo complejo B, que dependa del A. Si del tipo complejo A dependiesen más de un tipo complejo {B, C, D...}, todos los elementos simples escogidos de esos tipos complejos tendrán la misma posición que el elemento simple escogido para el tipo complejo A.

Para indicar la relación de dependencia entre tipos complejos necesitamos indicar el tipo complejo del que depende. Esto se realiza asignándole al atributo `dependence` del tipo complejo dependiente, el nombre del tipo complejo del que depende. Por lo que cuando un tipo complejo tiene un atributo `dependence` con el valor `true` asignado, quiere decir que de él dependen otros tipos complejos, y cuando el atributo `dependence` tiene asignado el nombre de un tipo complejo, quiere decir que depende de dicho tipo complejo.

Siguiendo con el ejemplo de `EventoTerminal`, en el código 5.26, se aplica el atributo `dependence`.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="EventoTerminal">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="false" type="Integer"
                                          min="0" max="99"></field>
    <field name="destino" quotes="true" type="ComplexType"
                                                  dependence="true">
      <attribute type="String" value="Madrid"></attribute>
      <attribute type="String" value="Barcelona"></attribute>
      <attribute type="String" value="Sevilla"></attribute>
      <attribute type="String" value="Jerez de la Frontera"></attribute>
    </field>
    <field name="destSiglas" quotes="true" type="ComplexType"
```

```
                                                   dependence="destino">
      <attribute type="String" value="MDR"></attribute>
      <attribute type="String" value="BCN"></attribute>
      <attribute type="String" value="SVL"></attribute>
      <attribute type="String" value="XRZ"></attribute>
    </field>
    <field name="destId" quotes="true" type="ComplexType"
                                                   dependence="destino">
      <attribute type="Integer" value="012"></attribute>
      <attribute type="Integer" value="054"></attribute>
      <attribute type="Integer" value="099"></attribute>
      <attribute type="Integer" value="113"></attribute>
    </field>
  </block>
</event_type>
```

Código 5.26: Uso del atributo dependence.

En el ejemplo del código 5.26, del tipo complejo `destino` dependen los tipos complejos `destSiglas` y `destId`. Esto implica que cuando se escoja el elemento de tipo simple con posición 1 del tipo complejo `destino`: `Madrid`, se escogerán los elementos de tipo simple que se encuentren en la posición 1 de los elementos complejos que dependen de él; es decir, del tipo complejo `destSiglas`: `MDR` y del tipo complejo `destId`: `012`.

Para tener una visión completa de la propuesta de especificación presentada en este capítulo, en el apéndice A se recoge el XML Schema que sigue la misma. En este se recogen todos los componentes explicados con anterioridad y los atributos que pueden tener.

5.3. Conclusiones

Este capítulo se ha detallado una de las tres fases de las que se compone el método propuesto en la presente tesis (véase capítulo 4): la *definición del tipo de evento*. Para ello se ha presentado una especificación para la definición de eventos con un lenguaje de marcado XML. Se han introducido cada uno de los elementos de la especificación que ayudan a definir el tipo de evento. Estos elementos se identifican con las etiquetas propias de este lenguaje de programación y, cada uno de ellos (según definición), contienen una serie de atributos específicos y obligatorios. Concretamente se han definido los elementos

`<event_type>`, `<block>`, `<field>`, `<optionalfields>` y `<attribute>`, aunque este último no se define como un elemento propio de la estructura para definir un evento, sino como parte de un tipo de dato complejo.

Los tipos de datos introducidos (tipo simple o tipo complejo) pueden utilizarse para determinar los valores que adquirirán los elementos. Gracias a la cobertura de los tipos de datos contemplados, se logra que la especificación pueda definir una gran variedad de tipos de eventos. Entre los tipos simples se encuentran: enteros, enteros grandes, de punto flotante, cadenas de caracteres, alfanuméricos, booleanos, fecha y tiempo. Y los tipos complejos pueden definirse a través de las combinaciones de tipos simples, o bien formando tipos anidados cubriendo otra gran variedad de tipos de evento.

Esta propuesta de especificación ha seguido las pautas de [93], donde se le da una gran importancia al tipo del evento, la cual viene representada como una colección de atributos del evento. En el capítulo 9 se utilizará esta propuesta de especificación para la definición de tipo de evento, con tipos de eventos reales.

Capítulo 6

Generador de eventos IoT-TEG

En este capítulo se describen las últimas etapas del método propuesto para la generación automática de eventos: la *validación* de la definición y el *generador de eventos* según la definición. En concreto se especificará el funcionamiento y arquitectura del generador de eventos IoT-TEG (*Internet of Things - Test Event Generator*), un generador que acepta y valida las definiciones de eventos que siguen la especificación propuesta en el capítulo 5. Los eventos generados por IoT-TEG se obtienen en los formatos más comunes usados en la mayoría de las aplicaciones reales, permitiendo que estos sean casos de prueba para cualquier aplicación que esté basada en una arquitectura dirigida por eventos, o aplicaciones de procesamiento de eventos o que trabajen con eventos.

6.1. Introducción

Las aplicaciones basadas en una arquitectura dirigida por eventos se caracterizan porque necesitan una fuente de obtención de eventos. Dependiendo de la implementación del programa, estos obtendrán los eventos de una forma u otra. Así mismo, el formato de los eventos variará según el generador de eventos que se esté utilizando, tal y como se explicó en la sección 2.2. Luckham expone que hay tres principales fuentes de generadores de eventos [4]: capas IT (Information Tecnology), instrumentos (hardware, monitores software, sistemas de latidos...) y CEP (eventos complejos generados a raíz de eventos simples). Hoy en día, muchos programadores utilizan las plataformas IoT (normalmente web) como fuente de obtención de eventos, ya que son más accesibles y fáciles de manejar, además de ofrecer eventos de procesos reales de diversa índole.

Sin embargo, este hecho no evita el tener que depender de un generador de eventos apropiado para la realización de pruebas. Ya que, a pesar de que los eventos de las plataformas

IoT se obtienen de situaciones de la vida real (algo que podría ser un inconveniente ya que hay que esperar a que se capturen los eventos), sus valores deberán de ser modificados manualmente según la prueba que se quiera realizar. Además, algunas de estas plataformas IoT no ofrecen una cantidad de eventos suficiente para hacer algunos tipos de prueba. Y, adicionalmente, se corre el riesgo de que la plataforma de la que se dependa cambie su política y ya no sea pública. En este capítulo se presenta un generador de eventos, IoT-TEG, una herramienta de software libre que permite generar eventos de prueba de cualquier tipo de evento definido. Se basa en la especificación propuesta en el capítulo 5, la cual, como se ha podido comprobar, permite definir una amplia variedad de tipos de eventos. Este generador permitirá al usuario obtener eventos para hacer pruebas en su aplicación y comprobar funcionalidades específicas dotando a los eventos de valores concretos, consiguiendo así un programa robusto y sin fallos que tome las decisiones acertadas. Utilizando IoT-TEG, el programador que desee analizar su aplicación, no dependerá de plataformas o servidores públicos, se ahorrará implementar un generador de eventos para su aplicación y conseguirá los eventos generados en cualquiera de los formatos más comunes.

Para el desarrollo de IoT-TEG, se siguen las nociones que presentan Saboor y Rengasamy en el trabajo [92]. Donde se proponen unos puntos muy útiles para la realización de pruebas, desde diferentes perspectivas, para aplicaciones CEP. Entre otros:

1. El generador de eventos debe tener la habilidad de adaptarse (igual que la aplicación se adapta a la nueva fuente de eventos).
2. Los eventos de entrada deben ser generados bajo una aleatoriedad controlada (entre un rango con valores máximos y mínimos).
3. Los ficheros con los eventos de entrada tienen que tener el formato `CSV` o `XML`.
4. El generador de eventos debe ser genérico, es decir, que genere datos según un tipo de evento dado.

Adicionalmente se realiza un análisis de las plataformas IoT más populares, en la sección 3.5 se explica cada una de las plataformas IoT con detalle. La principal ventaja del uso de estas plataformas IoT, en comparación con la herramienta IoT-TEG, es que en estas plataformas los eventos provienen de situaciones y dispositivos reales. Pero las claras desventajas, a la hora de realizar pruebas, son:

1. El número de eventos a obtener en las plataformas es limitado. En concreto, algunas plataformas solo permiten obtener hasta un número determinado de eventos

en una petición, por lo que si se quiere una mayor cantidad, hay que volver a solicitar eventos hasta que se obtenga la cantidad deseada. En otras se obtiene un único evento por cada petición, por lo que hay que hay que realizar una solicitud por evento hasta conseguir la cantidad deseada. Por último, en otras plataformas, cuando se solicita un número elevado de eventos, se incluyen eventos anteriores en el grupo de "nuevos" eventos. Por ejemplo, si se solicitan 1000 eventos y en la siguiente se piden 2000 eventos, los primeros 100 eventos están incluidos en los últimos 2000.

2. No se puede personalizar el tipo de evento según las necesidades de las pruebas. Esto solamente se podría hacer de manera manual modificando los valores de los eventos una vez obtenidos, una tarea muy tediosa y propensa errores, y en el caso de generar una gran cantidad de eventos una tarea costosa en tiempo.

3. No se pueden generar automáticamente eventos con valores específicos. Al igual que en el caso anterior, se podría hacer de manera manual modificando los valores de los eventos. De nuevo una tarea que en función del tipo y del número de eventos, puede ser una tarea tediosa, propensa a errores y costosa en tiempo.

4. Hay que esperar a que los dispositivos recojan la información y actualicen los eventos en la plataforma. Una vez definido el tipo de evento para las pruebas con IoT-TEG, se obtienen los eventos de manera inmediata y, aunque no sean reales, tal y como se mostrará en la presente tesis, son útiles para hacer pruebas.

Tras el estudio de cada una de las plataformas (véase sección 3.5), no solo se han podido resaltar las ventajas y desventajas de las mismas con respecto la herramienta IoT-TEG a la hora de hacer pruebas, sino también ha permitido extraer los formatos de eventos más comunes. A continuación se listan cada una de las plataformas con los formatos aceptados:

- Buglabs [112] - formato `JSON`
- Carriots [118] - formatos `XML` y `JSON`
- Grovestreams [114] - formato `JSON`
- Lelylan [110] - formato `JSON`
- Particle [111] - formato `JSON`
- Plotly [115] - formatos `JSON` y `CSV`
- Sensorcloud [116] - formato `CSV`

- ThingSpeak [109] - formatos `JSON`, `CSV` y `XML`
- Xively [117] - formato `JSON`
- Zettajs [113] - formato `JSON`

Teniendo en cuenta los formatos existentes en las plataformas IoT, nuestra herramienta generará los eventos en los tres formatos de salida usados por dichas plataformas, es decir, `JSON, CSV, XML`.

6.2. Arquitectura de IoT-TEG

IoT-TEG está implementado en Java y acepta como entrada (a través de la línea de órdenes) la definición del tipo de evento (el fichero XML), que sigue la especificación propuesta, y el formato en el que se requieren los eventos. El generador de eventos consta de dos componentes principales:

1. Componente de validación
2. Componente de generación

La figura 6.1 muestra la relación entre estos dos componentes. El *componente de validación* obtiene la definición del tipo de evento (que vendrá en formato XML) y comprueba si esta sigue la especificación propuesta para la definición de tipos de eventos (véase capítulo 5). Si la definición es correcta, el *componente de generación* obtendrá la definición del tipo de evento y leerá el formato en el que se requieren. Tras leer la estructura del tipo de evento y saber cuántos y cómo tienen que ser generados, se obtendrá un fichero con el formato correspondiente que contiene los eventos generados.

Como ya se ha mencionado previamente, el formato de los ficheros de salida que contendrán los eventos vendrá en uno de los tres más comunes que se han utilizado en las plataformas IoT analizadas, `JSON, CSV, XML`.

A continuación se describen de forma detallada cada uno de los componentes del generador de eventos IoT-TEG.

6.2.1. Componente de validación

Este componente se encarga exclusivamente de comprobar que el fichero `XML` que contiene la definición del tipo de evento cumple la especificación propuesta en el capítulo 5 para la definición de tipos de eventos.

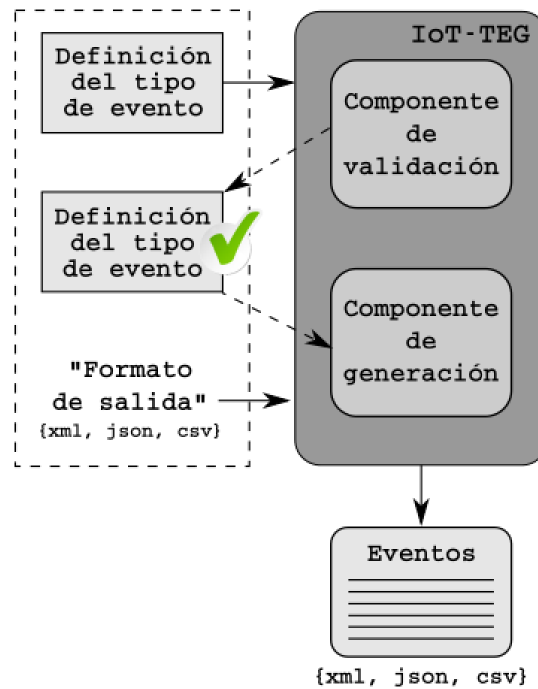

FIGURA 6.1: Arquitectura del generador de eventos IoT-TEG.

Para validar que la definición del tipo de evento introducido sigue la estructura de la especificación, se parte del elemento raíz del mismo. Este elemento solo debe contener como hijos elementos de tipo `<block>`, algo que se comprueba. Asimismo se analiza que estos elementos contengan los atributos obligatorios correspondientes, en este caso concreto el atributo que se busca es `name`. Además, dado que solo puede existir un elemento `<block>` con el atributo `repeat`, se comprueba que solo uno lo tenga.

Una vez que se comprueba que la estructura de los bloques es correcta, se analiza cada uno de los hijos de los elementos `<block>`. Según la especificación, los posibles hijos son `<field>` y `<optionalfields>`, algo que también se comprueba. Dado que cada uno de estos elementos difiere en cuanto atributos se refiere, se hacen comprobaciones separadas para ambos tipos. En el caso del elemento `<optionalfields>`, que no tiene atributos obligatorios, lo que se comprueba es que todos sus hijos sean de tipo `<field>`. Si lo que se está comprobando son los elementos de tipo `<field>` tienen que revisar los siguientes atributos: `name`, `quotes` y `type`. El hecho de no incluir alguno de ellos implicaría que la definición no sigue la especificación ya que todos son obligatorios.

El siguiente paso es comprobar los tipos de cada uno de los campos. Tal y como se ha podido observar estos pueden ser de tipo complejo o de tipo simple, luego el componente de *validación* ha de analizar cómo se han definido cada uno de ellos y los atributos que los componen. Para los tipos complejos se comprueba que los elementos que componen al tipo complejo sean `<field>` o `<attribute>`; si son de tipo `<field>` se volverán a realizar las comprobaciones oportunas para un elemento de este tipo. En el caso de ser

un elemento de tipo `<attribute>` se comprueba que este tenga el atributo `type` que determinará el tipo de dato simple que define al elemento. A continuación se comprueba si los atributos `dependence` y `chooseone` tienen los valores correctos. En el caso de `chooseone` se valida si el valor asignado es `true`, para `dependence` si el valor asignado no es `true` se comprueba si este corresponde al valor del atributo `name` de un tipo complejo que tenga `dependence` igual a `true`.

Por último, se validan los tipos de datos simples, para todos ellos se comprueba que si no está el atributo `value`, tienen que estar definidos los atributos obligatorios específicos de cada tipo de dato simple. En el caso de los tipos **Fecha** y **Tiempo** tiene que estar definido el atributo `format`, el resto de tipos de datos no tienen atributos obligatorios, pero se verifica que si están definidos sea de forma correcta. En el caso de los tipos **Entero**, **Punto Flotante** y **Entero grande** si se definen los dos atributos `max` y `min`, deben de aparecer ambos. En los tipos de datos **Alfanumérico** y **Cadena** se comprueba que si está definido el atributo `case` y su valor no coincide con el que puede aceptar el mismo, `low`, se advierte del error. En el tipo de dato **Booleano** si se define el atributo `isnumeric` se comprueba si el valor asignado es `true`.

Ejemplo: Hijo del elemento raíz no válido Si el fichero `XML` cumple la especificación, este es tomado por el *componente de generación* de eventos junto a la cadena que determina el formato de salida. Si por el contrario no se cumple alguno de los puntos anteriores, se lanza un mensaje al usuario indicando el error cometido.

Una vez descrito el proceso general de funcionamiento del *componente de validación*, a continuación se presentan ejemplos de posibles errores en la definición de un tipo de evento y su correspondiente mensaje lanzado por el *componente de validación*.

Cuando uno de los hijos de un elemento no es el correspondiente se lanza un mensaje. Por ejemplo, si uno de los hijos del elemento raíz `<event_type>` no es un elemento `<block>`, sino que es `<field>`, véase código 6.1, se lanzará el siguiente error:

``“No valid definition: The children of the root element must be block”``

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <field name="cabecera" type="date" value="25/10/2016"></field>
  <block name="valores" repeat="150">
  </block>
...
</event_type>
```

Código 6.1: Hijo del elemento raíz no es válido.

Ejemplo: Ausencia de atributo obligatorio en la definición de un elemento Si falta definir un atributo obligatorio en la definición de un elemento, se lanza un error. En el caso de no incluirse el atributo `type` en el elemento `<field>`, véase código 6.2, aparecerá el siguiente mensaje:

`”No valid definition: A “field” element needs a “type” attribute”`

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="id" quotes="true" min="0" max="20"></field>
  </block>
</event_type>
```

Código 6.2: No se incluye un atributo obligatorio en la definición de un elemento.

Ejemplo: Atributo repeat mal empleado Existen casos especiales que hay que tener en cuenta; si en la definición del tipo de evento, ninguno de los elementos `<block>` tiene el atributo `repeat` (véase código 6.3), o son varios los elementos `<block>` que tienen el atributo `repeat`, el *componente de validación* lanzará el siguiente mensaje:

`”No valid definition: One “repeat” attribute per event type definition”`

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="cabecera"
                      value="Creado:25/10/2016">
  </block>
  <block name="valores">
    <field name="id" quotes="true" type="Integer" min="0"
                                                  max="20"></field>
  </block>
</event_type>
```

Código 6.3: Definición de los elementos.

Ejemplo: Ausencia de atributo obligatorio en un dato de tipo simple Para cada tipo de dato simple se comprueban los atributos obligatorios, si no aparecen se lanza un error. En el caso de no incluir el atributo `format` en un tipo de dato Tiempo, véase código 6.4, se mostrará el siguiente mensaje:

`”No valid definition: A “Time” data type needs a “value” attribute or a “format” attribute”`

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="hora" quotes="true" type="Time"></field>
  </block>
</event_type>
```

Código 6.4: No se incluye atributo obligatorio del tipo de dato Tiempo.

Se contempla la posibilidad de que el usuario quiera definir el elemento usando el atributo `value`.

Ejemplo: Mal uso de un atributo obligatorio en un dato de tipo simple Siguiendo con los tipos de datos simples, para aquellos en los que se pueden utilizar los atributos `max` y `min`: Entero, Punto Flotante y Entero grande, si no se emplean ambos en la definición, aparece un error. En el código 6.5, no se incluye el atributo `min` en la definición de un tipo de dato Entero grande.

`”No valid definition: It is needed a “min” attribute for the “Long” data type”`

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="num" quotes="true" type="Long" max="20020406995">
    </field>
  </block>
</event_type>
```

CÓDIGO 6.5: No se incluye el atributo min en la definición de un tipo Entero grande.

Ejemplo: Valor erróneo asignado a un atributo de un dato de tipo simple Si un valor asignado al atributo `case` no es el correcto para cualquiera de los tipos de dato simple que lo utilizan: Booleano, Cadena y Alfanumérico, se lanzará un mensaje:

`”No valid definition: The default value of the case attribute (“String“ data type) works with capital letters, if you want lowercase assign “low“ to case”`

En el código 6.6 el valor `”low“` no está asignado al atributo `case` en la definición de un tipo de dato Cadena.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="character" quotes="true" type="String" case="l">
    </field>
  </block>
</event_type>
```

CÓDIGO 6.6: El valor low no está asignado al atributo case.

Ejemplo: Valor erróneo asignado a un atributo de tipo de dato complejo Y, finalmente, al igual que se comprueban los atributos de los tipos simples, se validan los atributos opcionales de los tipos complejos (`chooseone` y `dependence`). Cuando aparece el atributo `chooseone` se comprueba que el valor que tiene asignado

es `true`. En el caso del atributo `dependence`, se comprueba que tiene asignado el valor `true` y, en caso de no ser así, si el valor asignado corresponde al valor del atributo `name` de un tipo complejo que contenga el atributo `dependence` igual a `true`. Además, se comprueba que para todos los tipos complejos que tienen en su definición el atributo `dependence` igual a `true`, existe al menos un tipo complejo que dependa de él.

El código 6.7 muestra un ejemplo en el que el valor de `dependence` no es igual a `true` ni tampoco al valor del atributo `name` de un tipo complejo con `dependence` igual a `true`. El tipo complejo "siglas" tiene asignado en su atributo `dependence` el valor `sitio`, que no corresponde al valor que tiene el atributo `name` del único tipo complejo cuyo atributo `dependence` es igual a `true`. El mensaje lanzado sería:

```
''No valid definition: The value of the dependence attribute has to
be true or the value of the name attribute of a ComplexType with
true asigned to its dependence attribute.
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="lugar" quotes="true" type="ComplexType"
                                            dependence="true">
      <attribute type="String" value="Madrid"></attribute>
      <attribute type="String" value="Sevilla"></attribute>
      <attribute type="String" value="Valencia"></attribute>
    </field>
    <field name="siglas" quotes="true" type="ComplexType"
                                            dependence="sitio">
      <attribute type="String" value="MAD"></attribute>
      <attribute type="String" value="SEV"></attribute>
      <attribute type="String" value="VAL"></attribute>
    </field></block>
</event_type>
```

Código 6.7: Valor del atributo dependence inválido.

En el Apéndice A se presenta el *XML Schema* que sigue la propuesta de especificación para la definición de tipos de eventos, la cual ha de cumplir cualquier definición de tipo de evento para ser válida.

6.2.2. Componente de generación

Una vez validada la definición del tipo de evento, este componente sigue la misma ruta que el *componente de validación*; comenzando por el elemento raíz, va leyendo la estructura de los elementos `<block>`, profundizando en los elementos `<field>`, `<optionalfields>` y `<attribute>`, y leyendo los tipos de datos que los definen. Por otro lado, según el valor de la cadena que indica el formato de salida `XML, JSON, CSV` la estructura del fichero de salida variará.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
 <block name="cabecera"
                     value="Creado:25/10/2016">
 </block>
 <block name="valores" repeat="150">
   <field name="zona" quotes="true" type="String" endcharacter="F"
                                              length="1"></field>
   <field name="coordenadas" quotes="true" type="ComplexType">
     <field name="lat" quotes="false" type="Float" min="-90.0"
                                              max="90.0"></field>
     <field name="lon" quotes="false" type="Float" min="-180.0"
                                             max="180.0"></field>
 </field>
 </block>
</event_type>
```

Código 6.8: Definición de los elementos.

En el ejemplo del código 6.8 se definen los elementos del tipo de evento `TipoEvento`, tanto de tipo complejo, como de tipo simple. El tipo complejo `coordenadas`, es un tipo anidado cuyos dos elementos `{lat, lon}` son de tipo de dato `Float` con un un rango de valores cada uno. El tipo de dato simple `zona`, es de tipo de dato `String` con una longitud de un carácter, cuyo máximo valor a obtener es la letra `F`.

El *componente de generación* se mueve por los elementos `<block>`, escribe el contenido del atributo `name` y, si existe, el contenido del atributo `value` en el fichero de salida (según el formato especificado por el usuario). Si el elemento `<block>` contiene elementos hijos `<field>` que añaden más información, se escribirán según estén definidos y el formato de salida escogido. Del elemento `<block>` que contiene el atributo `repeat`, solo se escribirá el valor del atributo `name`. En el fichero de salida, y según el formato indicado por el

usuario, se escribirán los valores tanto de la estructura del tipo de evento como los valores generados según el tipo de evento.

```
{"cabecera":{Creado:25/10/2016},
"valores":[{"zona":"E","coordenadas":{"lat":-62.38017,"lon":-33.509583}},
{"zona":"C","coordenadas":{"lat":-32.789776,"lon":18.74559}},
{"zona":"C","coordenadas":{"lat":-48.45069,"lon":166.79547}},
{"zona":"B","coordenadas":{"lat":-41.949924,"lon":168.73575}},
{"zona":"F","coordenadas":{"lat":37.806435,"lon":-111.697975}},...]}
```

Código 6.9: Eventos generados en formato JSON para el código 6.8.

En el código 6.9 aparece el resultado de la generación según el formato `JSON` para el ejemplo del código 6.8. El primer valor corresponde al atributo `name` del primer elemento `<block>`, *cabecera*, y como valor el contenido del atributo `value`. A continuación el elemento `<block>` con el atributo `name` igual a *valores*; este tiene el atributo `repeat`, luego se escribirán tantos valores según el número especificado en este atributo (según el ejemplo es 150). Dentro de este elemento `<block>`, están definidos dos elementos `<field>`, uno de tipo simple y otro de tipo complejo, sus valores están determinados por los atributos utilizados para definir el evento (`min` y `max` para *lat* y *lon*, y `endcharacter` y `length` para *zona*). El valor de *cabecera* es fijo, a no ser que el usuario cambie en la definición del tipo de evento su valor.

```
<xml>
<cabecera>Creado:25/10/2016</cabecera>
<valores>
  <feed>
    <zona type="String">"F"</zona>
    <coordenadas>
      <lat type="Float">48.30255</lat>
      <lon type="Float">-59.579117</lon>
    </coordenadas>
  </feed>
  <feed>
    <zona type="String">"B"</zona>
    <coordenadas>
      <lat type="Float">-4.8984604</lat>
      <lon type="Float">95.81061</lon>
    </coordenadas>
```

```
  </feed>
  <feed>
    <zona type="String">"F"</zona>
    <coordenadas>
      <lat type="Float">74.691635</lat>
      <lon type="Float">83.09613</lon>
    </coordenadas>
  </feed>
  <feed>
    <zona type="String">"C"</zona>
    <coordenadas>
      <lat type="Float">-9.390198</lat>
      <lon type="Float">39.855988</lon>
    </coordenadas>
  </feed>
  <feed>
    <zona type="String">"D"</zona>
    <coordenadas>
      <lat type="Float">48.91452</lat>
      <lon type="Float">65.083176</lon>
    </coordenadas>
  </feed>
  ...
</valores>
</xml>
```

Código 6.10: Eventos generados en formato XML para el código 6.8.

El código 6.10 es el resultado de la generación según el formato `XML`. Para las salidas de en este formato, se han seguido las salidas observadas en las plataformas IoT; cada evento se delimita con dos etiquetas que indican el comienzo y el fin de los valores del evento. En el generador IoT-TEG, las etiquetas delimitadoras son `<feed>`, para indicar el inicio, y `</feed>` para indicar el fin. El nombre de esta etiqueta no influye, su funcionalidad es meramente estructural. El resto de etiquetas corresponde a cada uno de los elementos `<block>`, `<field>` y `<optionalfields>` definidos en el tipo de evento. Al igual que en el formato `JSON`, los valores de los elementos `<block>` que no tienen el atributo `repeat`, son valores fijos en la salida; el usuario puede cambiarlos siempre que lo desee en la definición del tipo de evento.

Si el formato de salida fuera CSV, véase código 6.11, al igual que las plataformas IoT, el generador IoT-TEG, se centra en los valores definidos dentro del elemento `<block>` que contiene el atributo `repeat`, luego no se tiene en cuenta la información adicional.

```
zona,coordenadas.lat,coordenadas.lon
"F",48.733047,-108.640205
"C",27.350014,4.1559143
"E",72.83569,23.52864
"C",72.64813,-61.179634
"B",58.31308,-14.486252
...
```

Código 6.11: Eventos generados en formato CSV para el código 6.12.

Como la filosofía que sigue el generador de eventos IoT-TEG es que los eventos se definen según la estructura que tienen, representada como una colección de atributos del evento, son los tipos de datos (simples y complejos) los que tienen un papel de peso en la generación de los mismos. Los atributos están representados en la propuesta de especificación de la presente tesis a través de los elementos `<field>` y `<optionalfields>` definidos dentro del `<block>` que contiene el atributo `repeat`. Cuando se está generando un tipo de dato, sea de tipo simple o complejo, el generador de eventos generará el valor del evento según los atributos que estén definidos en el mismo. Con estos atributos se indica tanto *la forma* del tipo de dato, como sus *posibles valores*. La finalidad de cada uno determina a qué grupo pertenece.

6.2.2.1. Tipos de datos simples

En esta sección se describe cómo actúa el *componente de generación* según los atributos definidos en los tipos de datos simples. Cabe recordar que como el generador de eventos IoT-TEG se basa en la especificación propuesta en el capítulo 5, los tipos de datos simples que se contemplan son los siguientes:

1. **Entero**, identificados por `Integer`.
2. **De punto flotante**, identificados por `Float`.
3. **Entero grande**, identificados por `Long`.
4. **Cadena de caracteres**, identificados por `String`.
5. **Alfanumérico**, identificados por `Alphanumeric`.

6. **Booleano**, identificados por `Boolean`.

7. **Fecha**, identificados por `Date`.

8. **Tiempo**, identificados por `Time`.

Respecto a los atributos que determinan *los posibles valores* de los campos, cabe aclarar que todos los valores de los tipos de datos se generan de forma aleatoria, a excepción de aquellos a los que el usuario les especifique un valor determinado a través de `value`. La aleatoriedad de los valores vendrá determinada por los atributos que tengan asignados los elementos.

Para los tipos de datos enteros, enteros grandes o puntos flotantes, los valores de los atributos `min` y `max` delimitarán el rango de los valores aleatorios que pueda obtener dicho elemento.

Con el código 6.12 obtendrían 150 repeticiones del evento tipo "TipoEvento" con unos valores en sus atributos `{terminal_id, temp, timestamp}` determinados por los atributos del tipo de evento que se define `{max, min}`. Además el tipo de dato punto flotante tiene el atributo `precision` que permite que se redondee el número generado según el valor de este atributo. Según dicho ejemplo el valor del atributo `temp` tiene que redondearse con dos decimales.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="terminal_id" quotes="true"
                               type="Integer" min="100" max="999"></field>
    <field name="temp" quotes="true" type="Float"
                               min="5.0" max="30.9" precision="2"></field>
    <field name="timestamp" quotes="false" type="Long"
                          min="1000000000000" max="9999999999999"></field>
  </block>
</event_type>
```

Código 6.12: Atributos min y max.

Para el código 6.12 los eventos que se generarán son como los del código 6.13, código 6.14 y código 6.15 según se especifique formato de salida `JSON`, `XML` o `CSV`, respectivamente.

```
{"valores":[
{"terminal_id":"123","temp":"23.44","timestamp":1939493209642},
{"terminal_id":"564","temp":"11.22","timestamp":4885623189736},
{"terminal_id":"778","temp":"6.68","timestamp":1897363488562},...]}
```

Código 6.13: Eventos generados en formato JSON para el código 6.12.

```
<xml>
<valores>
  <feed>
    <terminal_id type="Integer">"123"</terminal_id>
    <temp type="Float">"23.44"</temp>
    <timestamp type="Long">1939493209642<timestamp>
  </feed>
  <feed>
    <terminal_id type="Integer">"564"</terminal_id>
    <temp type="Float">"11.22"</temp>
    <timestamp type="Long">4885623189736<timestamp>
  </feed>
  <feed>
    <terminal_id type="Integer">"778"</terminal_id>
    <temp type="Float">"6.68"</temp>
    <timestamp type="Long">1897363488562<timestamp>
  </feed>  ...
</valores>
</xml>
```

Código 6.14: Eventos generados en formato XML para el código 6.12.

```
terminal_id,temp,timestamp
"123","23.44",1939493209642
"564","11.22",4885623189736
"778","6.68",1897363488562...
```

Código 6.15: Eventos generados en formato CSV para el código 6.12.

Para los tipos de datos alfanuméricos y cadena de caracteres, serán los atributos `length` y `endcharacter` los que indican la longitud que puede tener la cadena y hasta qué letra puede generarse la misma.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="letr_matr" quotes="true" type="String" length="3"
    </field>
    <field name="calidad" quotes="false" type="String" endcharacter="D"
                                         length="1" case="low"></field>
    <field name="cod_hex" quotes="true" type="Alphanumeric" length="6"
                                              endcharacter="F"></field>
  </block>
</event_type>
```

Código 6.16: Atributos length y endcharacter.

En el ejemplo del código 6.16 se obtendrían 150 repeticiones del evento tipo “TipoEvento” con unos valores en sus atributos `{letr_matr, calidad, cod_hex}` determinados por los atributos del tipo de evento que se definen `{length, endcharacter}`.

Siguiendo el código 6.16, las salidas obtenidas por formato de salida `JSON`, `XML` y `CSV` pueden verse en los códigos 6.17, 6.18 y 6.19.

```
{"valores":[
{"letr_matr":"AAC","calidad":d,"cod_hex":"00FFFF"},
{"letr_matr":"JXI","calidad":a,"cod_hex":"FFA500"},
{"letr_matr":"QUO","calidad":c,"cod_hex":"FFC0CB"},...]}
```

Código 6.17: Eventos generados en formato JSON para el código 6.16.

```
<xml>
<valores>
  <feed>
    <letr_matr type="String">"AAC"</letr_matr>
    <calidad type="String">d</calidad>
    <cod_hex type="Alphanumeric">"00FFFF"<cod_hex>
  </feed>
  <feed>
    <letr_matr type="String">"JXI"</letr_matr>
    <calidad type="String">a</calidad>
    <cod_hex type="Alphanumeric">"FFA500"<cod_hex>
```

```
  </feed>
  <feed>
    <letr_matr type="String">"QUO"</letr_matr>
    <calidad type="String">c</calidad>
    <cod_hex type="Alphanumeric">"FFC0CB"<cod_hex>
  </feed>  ...
</valores>
</xml>
```

Código 6.18: Eventos generados en formato XML para el código 6.16.

```
letr_matr,calidad,cod_hex
"AAC",d,"00FFFF"
"JXI",a,"FFA500"
"QUO",c,"FFC0CB"...
```

Código 6.19: Eventos generados en formato CSV para el código 6.16.

Para el tipo booleano es el atributo `isnumeric` (valor `n`) el que determina que los valores booleanos se representen a través de los dígitos {1,0}.

En el código 6.20 se obtendrían 150 repeticiones del evento tipo “TipoEvento” con unos valores en sus atributos `{seleccionado, activo}` determinados el atributo del tipo de evento que se define `{isnumeric}`.

```
 <?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="seleccionado" quotes="true" type="Boolean"></field>
    <field name="activo" quotes="false" type="Boolean" isnumeric="n">
    </field>
  </block>
</event_type>
```

Código 6.20: Atributo isnumeric.

Siguiendo el código 6.20 y suponiendo que el formato de salida fuera `JSON`, `XML` y `CSV`, los resultados se muestran en código 6.21, 6.22 y 6.23.

```
{"valores":[
{"seleccionado":"true","activo":1},
{"seleccionado":"false","activo":1},
{"seleccionado":"true","activo":0},...]}
```

Código 6.21: Eventos generados en formato JSON para el código 6.20.

```
<xml>
<valores>
  <feed>
    <seleccionado type="Boolean">"true"</seleccionado>
    <activo type="Boolean">1</activo>
  </feed>
  <feed>
    <seleccionado type="Boolean">"false"</seleccionado>
    <activo type="Boolean">1</activo>
  </feed>
  <feed>
    <seleccionado type="Boolean">"true"</seleccionado>
    <activo type="Boolean">0</activo>
  </feed>  ...
</valores>
</xml>
```

Código 6.22: Eventos generados en formato XML para el código 6.20.

```
seleccionado,activo
"true",1
"false",0
"true",1
```

Código 6.23: Eventos generados en formato CSV para el código 6.20.

Centrándose en los atributos que indican *la forma* del tipo de dato, estos indican el formato de salida que tienen que adquirir los valores. En el caso de los tipos de dato fecha y tiempo, será el atributo `format` el que permitirá la presentación de los valores de un modo u otro.

El código 6.24 muestra un ejemplo donde se obtendrían 150 repeticiones del evento tipo "TipoEvento" con unos valores en sus atributos `{hora, dia}` determinados por el atributo del tipo de evento que se define `{format}`.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="hora" quotes"false" type="Time" format="HH:mm"></field>
    <field name="dia" quotes="false" type="Date" format="yyyy-MM-DD">
    </field>
  </block>
</event_type>
```

Código 6.24: Atributo format.

Siguiendo el código 6.24, y suponiendo que el formato de salida fuera `JSON`, `XML` y `CSV`, los resultados se muestran en código 6.25, 6.26 y 6.27, respectivamente.

```
{"valores":[
{"hora":02:16,"dia":2015-06-08},
{"hora":07:11,"dia":2016-04-19},
{"hora":06:57,"dia":2016-05-14},...]}
```

Código 6.25: Eventos generados en formato JSON para el código 6.24.

```
<xml><valores><feed>
    <hora type="Time">02:16</hora>
    <dia type="Date">2015-06-08</dia>
  </feed>
  <feed>
    <hora type="Time">07:11</hora>
    <dia type="Date">2016-04-19</dia>
  </feed>
  <feed>
    <hora type="Time">06:57</hora>
    <dia type="Date">2016-05-14</dia>
  </feed>  ...
</valores></xml>
```

Código 6.26: Eventos generados en formato XML para el código 6.24.

```
hora,dia
02:16,2015-06-08
07:11,2016-04-19
06:57,2016-05-14
```

Código 6.27: Eventos generados en formato CSV para el código 6.24.

Para los tipos de datos alfanuméricos y cadena de caracteres, el atributo `case` (valor por defecto `low`) indica que la cadena ha de generarse con letras minúsculas. En el código 6.16 se muestra un ejemplo de su definición y en los códigos 6.17, 6.18, 6.19 un resultado de su generación.

6.2.2.2. Tipos de datos complejos

La forma de actuar del *componente de generación* frente a tipos de datos complejos también depende de los posibles atributos definidos en los mismos. Dado que un tipo de dato complejo se forma combinando tipos de datos simples, se analizan los atributos específicos de este tipo de dato que determinan *los posibles valores* de los campos: `chooseone` y `dependence`. Los tipos de datos complejos no poseen atributos que modifiquen el formato de salida del tipo de dato tal y como se vio en la sección 5.2.2.

Atributo chooseone Cuando el *componente de generación* encuentra que el atributo `chooseone` tiene valor `true`, selecciona de forma aleatoria cualquiera de los elementos posibles que componen al tipo complejo. De esta forma el valor del elemento de este tipo complejo va adquiriendo un valor aleatorio para cada evento generado. Como se vio en la sección 5.2.2, un tipo complejo puede estar formado por tipos simples, o bien puede ser un tipo anidado. En el código 6.28 se definen dos tipos complejos `{color, coordenadas}`. El elemento `color` está formado por tipos simples, y `coordenadas` es de tipo anidado. En este ejemplo se obtendrían 150 repeticiones del evento tipo "TipoEvento" con unos valores en sus elementos `{color, coordenadas}` determinados por el atributo `{chooseone}`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="color" quotes="true" type="ComplexType"
                                                  chooseone="true">
      <attribute type="String" value="red"></attribute>
      <attribute type="String" value="blue"></attribute>
      <attribute type="String" value="yellow"></attribute>
      <attribute type="String" value="white"></attribute>
      <attribute type="String" value="orange"></attribute>
      <attribute type="String" value="warmwhite"></attribute>
      <attribute type="String" value="cyan"></attribute>
      <attribute type="String" value="green"></attribute>
      <attribute type="String" value="orange"></attribute>
      <attribute type="String" value="pink"></attribute>
    </field>
    <field name="coordenadas" quotes="false" type="ComplexType"
    chooseone="true">
      <field name="x" quotes="false" type="Float" min="-30.0"
                                                 max="30.0"></field>
      <field name="y" quotes="false" type="Float" min="-30.0"
                                                 max="30.0"></field>
      <field name="z" quotes="false" type="Float" min="-30.0"
                                                 max="30.0"></field>
    </field>
  </block>
</event_type>
```

Código 6.28: Uso del atributo chooseone.

Siguiendo el código 6.28 y suponiendo como formato de salida `JSON`, `XML` y `CSV`, los resultados se muestran en código 6.29, 6.30 y 6.31, respectivamente.

```json
{"valores":[{"color":"yellow","coordenadas":{"z":6.7538757}},
{"color":"red","coordenadas":{"z":11.695194}},
{"color":"red","coordenadas":{"z":2.202999}},
{"color":"orange","coordenadas":{"z":-16.143444}},
{"color":"white","coordenadas":{"y":-14.639218}},...]}
```

Código 6.29: Eventos generados en formato JSON para el código 6.28.

```
<xml>
<valores>
  <feed>
    <color type="String">"cyan"</color>
    <coordenadas type="Float">
      <x type="Float">-2.5526333</x>
    </coordenadas>
  </feed>
  <feed>
    <color type="String">"green"</color>
    <coordenadas type="Float">
      <y type="Float">10.037601</y>
    </coordenadas>
  </feed>
  <feed>
    <color type="String">"orange"</color>
    <coordenadas type="Float">
      <x type="Float">11.198956</x>
    </coordenadas>
  </feed>
  <feed>
    <color type="String">"warmwhite"</color>
    <coordenadas type="Float">
      <x type="Float">-15.706362</x>
    </coordenadas>
  </feed>
  <feed>
    <color type="String">"warmwhite"</color>
    <coordenadas type="Float">
      <y type="Float">-28.462416</y>
    </coordenadas>
  </feed>...
</valores>
</xml>
```

Código 6.30: Eventos generados en formato XML para el código 6.28.

```
color,coordenadas.x,coordenadas.y,coordenadas.z
"orange",,23.00483,
"red",20.049824,,
"red",,,13.207249
"orange",-10.01852,,
"red",,28.19408,
```

Código 6.31: Eventos generados en formato CSV para el código 6.28.

Atributo dependence Para el atributo `dependence`, IoT-TEG actúa de forma similar, selecciona de forma aleatoria uno de los valores que compone el elemento de tipo complejo. Tal y como se explicó en la sección 5.2.2.2 cuando se emplea el atributo `dependence`, según el elemento escogido del tipo complejo, se escogen los elementos correspondientes de los tipos complejos que dependen de este. Este primer elemento se escoge de forma aleatoria y se almacena su posición, se comprueba la dependencia (viendo el valor del atributo `dependence`) y se utiliza la misma posición que ocupaba el elemento previamente escogido con los elementos complejos dependientes.

Se explica el comportamiento de `dependence`, tomando como ejemplo el código 6.32, donde se pretende obtener 150 repeticiones del evento tipo "TipoEvento" con unos valores en sus elementos `{letra, num, combina}` determinados por el atributo `{dependence}`. El *componente generador* escoge aleatoriamente un elemento de `letra`, por ejemplo B. Tras seleccionar este valor, el resto de valores de los `<field>` que dependen de `letra`, serán los que ocupen la misma posición que el elemento escogido en `letra`, B; `num` tendrá el valor 2 y `combina` tendrá el valor B2.

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TipoEvento">
  <block name="valores" repeat="150">
    <field name="letra" quotes="true" type="ComplexType"
                                                    dependence="true">
      <attribute type="String" value="A"></attribute>
      <attribute type="String" value="B"></attribute>
      <attribute type="String" value="C"></attribute>
    </field>
    <field name="num" quotes="false" type="ComplexType"
                                                    dependence="letra">
      <attribute type="Integer" value="1"></attribute>
```

```
      <attribute type="Integer" value="2"></attribute>
      <attribute type="Integer" value="3"></attribute>
    </field>
    <field name="combina" quotes="false" type="ComplexType"
                                             dependence="letra">
      <field name="uno" quotes="true" type="String" value="A1">
      </field>
      <field name="dos" quotes="true" type="String" value="B2">
      </field>
      <field name="tres" quotes="true" type="String" value="C3">
      </field>
    </field>
  </block>
</event_type>
```

CÓDIGO 6.32: Uso del atributo dependence.

Teniendo en cuenta la definición del tipo de evento que aparece en el código 6.32, y suponiendo que el formato de salida solicitado fuera JSON, XML y CSV, los resultados se muestran en código 6.33, 6.34 y 6.35, respectivamente.

```
{"valores":[
{"letra":"A","num":1,"combina":{"uno":"A1"}},
{"letra":"C","num":3,"combina":{"tres":"C3"}},
{"letra":"B","num":2,"combina":{"dos":"B2"}},,...]}
```

CÓDIGO 6.33: Eventos generados en formato JSON para el código 6.32.

```
<xml>
<valores>
  <feed>
    <letra type="String">"A"</letra>
    <num type="Integer">1</num>
     <combina type=ComplexType>
      <uno type="String">"A1"</uno>
     </combina>
  </feed>
  <feed>
    <letra type="String">"C"</letra>
    <num type="Integer">3</num>
```

```
        <combina type=ComplexType>
         <tres type="String">"C3"</tres>
        </combina>
     </feed>
     <feed>
       <letra type="String">"B"</letra>
       <num type="Integer">2</num>
        <combina type=ComplexType>
         <dos type="String">"B2"</dos>
        </combina>
     </feed>  ...
   </valores>
   </xml>
```

Código 6.34: Eventos generados en formato XML para el código 6.32.

```
letra,num,combina.uno, combina.dos, combina.tres
"A",1,"A1"
"C",3,,"B2"
"B",2,,,"C3"
```

Código 6.35: Eventos generados en formato CSV para el código 6.32.

6.3. Funcionalidad avanzada de lectura de consultas

El generador IoT-TEG aquí presentado permite obtener eventos aleatorios a partir de una especificación dada. Sin embargo, para obtener eventos específicos que permitan probar el sistema ante determinadas circunstancias, se necesitará no un generador de eventos aleatorios, sino un generador que permita crear eventos de prueba de una mayor calidad. En este sentido, se ha dotado al *componente de generación* de una funcionalidad adicional que permite generar eventos que cumplan determinadas restricciones específicas del programa para el que se generan. Esto permite obtener unos eventos de más calidad para las posibles pruebas a realizar.

Tal y como se ha estudiado en la sección 3.4, existen varios tipos de lenguajes EPL: orientados a flujos, orientados a reglas e imperativos. Para que esta funcionalidad de IoT-TEG pueda emplearse en la mayoría de ellos, se ha observado que gran parte de los EPL son lenguajes de consulta, por lo que esta funcionalidad se orienta a este tipo

de lenguajes. En concreto se ha escogido como referencia el lenguaje EPL de EsperTech (dado que va a ser utilizado en la presente tesis para hacer pruebas). Por ello esta funcionalidad por defecto está desactivada. En el caso de usar un EPL de consulta y querer activar esta funcionalidad, véase sección 6.3.1 para indicaciones de activación

Para esta funcionalidad se han considerado las operaciones relacionales y lógicas que aparezcan en las cláusulas `where` con tipos de datos simple de las consultas involucradas en el programa. En el código 6.36 se ve un ejemplo que destaca las partes de la consulta que se tienen en cuenta.

```
INSERT INTO cep.dns(timestamp, source, destination)
    SELECT ts, ipSrc, ipDst
    FROM sniffer.header.parsed.flat
    WHERE udpSrc = 53 OR udpDst = 53
```

Código 6.36: Partes de las consultas para la nueva funcionalidad de IoT-TEG.

Dado que son los elementos que intervienen en las operaciones relacionales y lógicas los que se tienen en consideración para generar los valores de los eventos, se estudia la estructura de estos. Primero se analizarán los elementos de las operaciones relacionales:

”campo opRel valor”

Donde *campo* es el nombre de alguno de los campos (`<field>`) definidos en la definición del evento, *opRel* es cualquiera de los operadores relacionales `{<,>,=,!=,<=,>=}` y *valor* es un valor de cualquiera de los **tipos simples** contemplados en el estándar propuesto (véase sección 5.2.1). Téngase en cuenta que en un tipo de dato complejo anidado (véase sección 5.2.2) se definen campos (`<field>`) de tipo simple, por lo que también estarían contemplados en esta funcionalidad.

Para enlazar varias operaciones relacionales se utilizan los operadores lógicos `{AND, OR}`, y los elementos involucrados siguen la siguiente estructura:

”{campo opRel valor} opLog {campo opRel valor}”

Donde *{campo opRel valor}* son las operaciones relacionales que se enlazan con el *opLog*, operador lógico, `{AND, OR}`. Con los operadores lógicos se ”especifica” qué campos (`<field>`) definidos en la definición del evento están relacionados y cómo. Pueden enlazarse tantas operaciones relacionales como se quiera a través de los operadores lógicos:

"{operaciónRel} opLog {operaciónRel} opLog {operaciónRel} opLog {operaciónRel} opLog {operaciónRel}..."

Donde se considera *{operaciónRel}* una operación relacional y *opLog* cualquiera de los operadores lógicos `{AND, OR}`.

6.3.1. Secuencia del proceso

Para que se ejecute la funcionalidad que se expone en esta sección, se escribirá, tras el formato en el que se quieren los eventos, la ruta del fichero que contiene (en texto plano) las consultas involucradas en el programa para el que se generarán los eventos.

El primer paso es localizar dentro de las consultas del programa aquellas que contienen cláusulas `where`. En el caso de no existir ninguna cláusula `where` el *componente de generación* se comportará como si no se hubiese activado esta funcionalidad, generando todos los eventos de forma aleatoria según la definición especificada. Una vez localizada una consulta con cláusula `where`, se comprueba si esta posee operadores lógicos. Si contiene se almacenan según el orden de aparición, ya que esto va a determinar la forma de almacenamiento de las operaciones relacionales que estén involucradas.

Una vez almacenados los posibles operadores lógicos se buscan en la consulta las operaciones relacionales, almacenando en una tripleta los elementos que forman la operación relacional: *{campo, opRel, valor}*. Si se han encontrado operadores lógicos en la consulta que se está analizando, quiere decir que la operación relacional almacenada está enlazada con otra mediante el operador lógico registrado. Si por el contrario no hay operadores lógicos, no se encontrarán más operaciones relacionales en la consulta.

Lo que se ha utilizado en el *componente de generación* de IoT-TEG para almacenar los valores y operaciones involucradas es un registro de tripletas cuya estructura es una "lista de listas de tripletas". Véase su declaración en la sección 6.4.

El mecanismo del *componente de generación* para almacenar los valores es el siguiente:

1. Cada operación relacional se guarda en una tripleta.
2. Todas las operaciones relacionales que aparezcan en las consultas involucradas, se van almacenando en una lista que contiene todas las tripletas.
3. Si en una consulta encontramos dos o más operaciones relacionales unidas por el operador lógico `AND`, las tripletas se guardan en la lista de tripletas anterior (mismo mecanismo).

4. El proceso de almacenamiento cambia si se encuentra un operador lógico `OR`; la lista de tripletas se duplica con la excepción del último elemento introducido (este último elemento es el primer `operaciónRel` de ''`{operaciónRel} opLog {operaciónRel}`'' donde `opLog` es el operador lógico `OR`). En esta lista de tripletas duplicada se inserta el último componente (tripleta) de la operación lógica `OR`.

5. A partir de aquí cada nueva operación relacional (tripleta), se almacenará en cada una de las listas de tripletas existentes.

6. Si existiera otro operador lógico `OR` en las consultas, se duplicarían todas las listas de tripletas existentes (menos el último elemento) y en cada una de las listas duplicadas, se introduciría la tripleta correspondiente al último componente (tripleta) de la operación lógica `OR`.

Se considera el siguiente conjunto de consultas para explicar cómo se registran los datos:

```
... where field1 < 3 AND field2 >= 50
... where field3 != 'A'
... where field4 = 'B' OR field4 = 'D'
... where field5 > 10
... where field6 < 50 OR field6 > 20
```

Código 6.37: Consultas EPL de EsperTech de ejemplo para IoT-TEG avanzado.

1. `... where field1 <3 AND field2 >= 50`

   En la primera consulta, tras la cláusula `where`, se encuentran dos operaciones relacionales unidas por el operador lógico `AND`. Las operaciones relacionales involucradas se almacenarán en una lista de tripletas que tendrá dos elementos (tripletas), una por cada operación relacional. La estructura de lista de lista de tripletas sería:

   ```
   {{[field1, <, 3], [field2, >=, 50]}}
   ```

2. `... where field3 != 'A'`

   Siguiendo con la segunda consulta, se encuentra una única operación relacional, para la cual se crea una tripleta que se añade en la lista de tripletas que contiene las dos tripletas anteriores. La estructura saliente es:

   ```
   {{[field1, <, 3], [field2, >=, 50], [field3, !=, 'A']}}
   ```

3. `... where field4 = 'B' OR field4 = 'D'`

   En la siguiente consulta se encuentran dos operaciones relacionales unidas por el operador lógico `OR`. La primera operación relacional `field4 = 'B'` se almacena en una tripleta que se añade a la lista de tripletas. Luego se hace un duplicado de la lista de tripletas, exceptuando el último elemento introducido (`field4 = 'B'`). A continuación se añade la siguiente operación relacional involucrada en la operación lógica con la que se está trabajando `field4 = 'D'`. Esta segunda lista de tripletas, se convierte en la segunda lista de la estructura "lista de listas de tripletas". Nuestra estructura estará formada ahora por una lista con dos listas de tripletas que serán:

```
{{[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                                              [field4, =, 'B']},
 {[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                                              [field4, =, 'D']}}
```

4. `... where field5 > 10`

   En la siguiente consulta se encuentra una operación relacional, tras almacenar los datos en una tripleta, se añade en ambas listas de tripletas de la estructura "lista de listas de tripletas". Nuestras dos listas de tripletas serán:

```
{{[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                         [field4, =, 'B'], [field5, >, 10]},
 {[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                         [field4, =, 'D'], [field5, >, 10]}}
```

5. `... where field6 < 50 OR field6 > 20`

   En la última consulta vuelven a estar involucradas dos operaciones relacionales unidas por el operador lógico `OR`. La primera operación relacional `field6 < 50` se almacena en cada una de las listas de tripletas de la estructura "lista de listas de tripletas" (actualmente con dos listas de tripletas). Luego se hace un duplicado de las listas de tripletas, exceptuando el último elemento introducido. En las dos nuevas listas de tripletas se añade la siguiente operación relacional involucrada en la operación lógica `field6 > 20`. Creándose cuatro listas de tripletas en la estructura "lista de listas de tripletas", como la que se muestra a continuación.

```
{{[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                [field4, =, 'B'], [field5, >, 10], [field6, <, 50]},
{[[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                [field4, =, 'D'], [field5, >, 10], [field6, <, 50]},
{[[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                [field4, =, 'B'], [field5, >, 10], [field6, >, 20]},
{[[field1, <, 3], [field2, >=, 50], [field3, !=, 'A'],
                [field4, =, 'D'], [field5, >, 10], [field6, >, 20]}}
```

Una vez conseguidos todos estos registros, dado que en esta funcionalidad solo se tienen en cuenta los tipos de dato simple, hay que recorrer la estructura "lista de listas de tripletas" para comprobar que los campos que se han registrado son de tipo simple. En caso contrario estos se eliminarán del registro (solo la tripleta que lo contenga). Una vez realizado el proceso de filtrado de tipos de datos simple, se contabiliza el número de eventos que hay que generar. Según el número de listas de tripletas almacenadas, se generará por cada una un grupo de eventos cumpliendo cada una de las restricciones. Adicionalmente, se creará un grupo de eventos con valores aleatorios. De este modo los eventos a generar se repartirán equitativamente entre los lotes creados y la generada aleatoriamente. Así si N es el número de eventos a generar, L el número de listas de tripletas creadas, se generarán

$$\left\lfloor \frac{N}{L+1} \right\rfloor \tag{6.1}$$

eventos de cada lista de tripletas, redondeando al entero inferior. En el caso de obtener un resultado decimal, dado que se redondea al entero inferior, se generarán tantos eventos aleatorios hasta completar el número de eventos requeridos por el usuario. Si por ejemplo el número de eventos a generar (especificado en la definición del tipo de evento) es 200, según el ejemplo que se muestra en el código 6.37, se generarán:

```
200 / 5 = 40
```

eventos de cada lista de tripletas, dejando entonces 40 eventos para que sean totalmente aleatorios (aunque siguiendo la definición del tipo de evento). Siguiendo con el ejemplo del código 6.37, se generarán por tanto:

- 40 eventos cuyo campo `field1` tenga valores mayores que 3, el campo `field2` con un valor mayor o igual a 50, el campo `field3` tenga un valor distinto a 'A', el

campo `field4` tenga un valor igual a 'B', el campo `field5` con valores mayores que 10 y el campo `field6` con valores menores que 50.

- 40 eventos cuyo campo `field1` tenga valores mayores que 3, el campo `field2` con un valor mayor o igual a 50, el campo `field3` tenga un valor distinto a 'A', el campo `field4` tenga un valor igual a 'D', el campo `field5` con valores mayores que 10 y el campo `field6` con valores menores que 50.
- 40 eventos cuyo campo `field1` tenga valores mayores que 3, el campo `field2` con un valor mayor o igual a 50, el campo `field3` tenga un valor distinto a 'A', el campo `field4` tenga un valor igual a 'B', el campo `field5` con valores mayores que 10 y el campo `field6` con valores mayores que 20.
- 40 eventos cuyo campo `field1` tenga valores mayores que 3, el campo `field2` con un valor mayor o igual a 50, el campo `field3` tenga un valor distinto a 'A', el campo `field4` tenga un valor igual a 'D', el campo `field5` con valores mayores que 10 y el campo `field6` con valores mayores que 20.
- 40 eventos con valores aleatorios en todos sus campos, aunque siguiendo la definición del tipo de evento.

Obviamente, en los eventos que se siguen las restricciones impuestas por las consultas también se tienen en cuenta las restricciones propias de la definición del tipo de evento, esto se ve con detalle en la siguiente sección 6.3.2.

Existen otras formas de implementar la funcionalidad avanzada de IoT-TEG. En vez de generar eventos según el rango establecido por las operaciones relacionales involucradas en las consultas, se podrían generar eventos según los valores límite que aparecen en las operaciones relacionales. Es decir, en una operación relacional *campo opRel valor*, se llama valor límite al *valor* que aparece en la misma. Los valores del elemento *campo* en los eventos generados, serán *valor + 1*, *valor - 1*, *valor + 10*, *valor - 10* (siempre dentro de las restricciones propias de la definición del tipo de evento). De este modo se asegura que los valores de *campo* estén tanto por "encima" como por "debajo" del valor límite, eso sí, no se tiene en cuenta *opRel*, el operador relacional involucrado. Esta otra implementación de la funcionalidad avanzada, se integrará en IoT-TEG y se compararán los resultados obtenidos con los de la presente tesis. Es una ampliación que se tiene en cuenta como trabajo futuro.

6.3.2. Tratamiento de datos

Tal y como se ha indicado, esta funcionalidad de IoT-TEG tiene en cuenta las operaciones relacionales y lógicas en las que intervengan datos de tipo simple. Según el estándar

propuesto en el capítulo 5, encontramos los siguientes tipos de datos dentro de esta categoría:

1. Entero
2. De punto flotante
3. Entero grande
4. Cadena de caracteres
5. Alfanumérico
6. Booleano
7. Fecha
8. Tiempo

Los operadores contemplados con cada tipo de dato depende de su uso habitual con las operaciones relacionales. A continuación se especifican, para cada tipo de dato, qué operadores se han considerado utilizando en función de uso habitual con los operadores relacionales. Es importante resaltar que solo un porcentaje de los eventos cumple las restricciones que se explicarán en detalle a continuación, ya que siempre existirá un grupo de eventos con valores completamente aleatorios independientemente de las restricciones impuestas por las consultas (pero sí siguiendo las de la definición del tipo de evento).

6.3.2.1. Entero, Punto flotante y Entero grande

Los tipos de datos enteros, grandes o datos de punto flotante suelen aparecer en las operaciones relacionales con cualquiera de los operadores `{<,>,=,!=,<=,>=}`, es por esto por lo que a la hora de sus implementaciones en la funcionalidad de IoT-TEG se han tenido en cuenta todos ellos.

Rescatando parte de la primera consulta del ejemplo del código 6.37

```
... where field1 < 3
```

y centrándose en el campo `field1`, actualmente con el operador `<` se generarán eventos cuyo valor en `field1` sea menor que 3 (más el resto de condiciones de las consultas). Pero no solo se tiene que tener en cuenta la restricción de la consulta, sino que también hay que evaluar la definición del tipo de evento. Si se considera que según la definición

del tipo de evento el campo `field1` puede tener valores de 1 a 9, entonces los valores a generar estarán en el rango [1,3] (estos se generarán de forma aleatoria según el rango anterior). El rango se establece no solo por la definición del tipo de evento, sino también por las restricciones de la consulta.

Con los operadores relacionales `{<,>,<=,>=}` se tiene el mismo comportamiento. Para el operador `!=` el valor generado cumple primero las restricciones según la definición del tipo de evento y luego se compara si el valor generado es igual al que se indica en la consulta que no lo sea. Si esto es así se vuelve a generar otro valor que cumpla ambas restricciones.

Este comportamiento se ha tenido en cuenta en la implementación de esta funcionalidad para los tres tipos de datos `{enteros, punto flotante y enteros grandes}` que suelen aparecer con cualquiera de los operadores relacionales. Adicionalmente, para los tipos de datos de punto flotante se ha tenido en cuenta, a la hora de generar los eventos, si está definido el atributo `precision`, ya que este determina la posición de redondeo del número (véase sección 5.2.1.2).

6.3.2.2. Cadenas de caracteres y Alfanuméricos

Para estos dos tipos de datos simples solo se han tenido en cuenta los operadores relacionales `{=,!=}`. El comportamiento del *componente de generación* cuando está involucrado el operador `=` es igual que si en la definición del tipo de evento estuviese el atributo `value` (véase sección 6.2.2), es decir se asigna el valor requerido.

Rescatando la tercera consulta del ejemplo del código 6.37

```
... where field4 = 'B' OR field4 = 'D'
```

se generará la mitad del número de eventos totales a generar con el campo `field4` igual a `'B'` y la otra mitad con el campo `field4` igual a `'D'` (junto con el resto de las condiciones de las consultas).

En el caso del operador `!=`, el valor generado tiene que cumplir las restricciones de la definición del tipo de evento y además tiene que ser diferente al valor que se indica en la consulta. Adicionalmente, se tiene que tener en cuenta si están definidos los atributos `endcharacter` y `length` en la definición del tipo de evento (atributos que determinan los posibles valores de los campos, véanse secciones 5.2.1.4, 5.2.1.5). Además, para estos tipos de datos, tiene que tenerse en cuenta el atributo (en la definición del tipo de evento)

que modifica la forma: `case`, ya que si este está especificado los valores han de generarse en minúscula.

Considerando la siguiente consulta del ejemplo del código 6.37

```
... where field3 != 'A'
```

se generará un porcentaje de eventos cuyo campo `field3` sea distinto de `'A'` (junto con las condiciones del resto de consultas). Si se considera que según la definición del tipo de evento el campo `field3` puede tener valores de 'A' a 'L', entonces los valores a generar estarán en el rango [B,L] (estos se generarán de forma aleatoria según el rango anterior). El rango se establece no solo por la definición del tipo de evento, sino también por las restricciones de la consulta.

6.3.2.3. Booleanos

Los operadores relacionales que se han tenido en cuenta para este tipo de datos son `{=,!=}`. El comportamiento del *Componente de generación* de eventos es igual que en los casos anteriores cuando está involucrado el operador `=`. Sin embargo, cuando aparece el operador `!=`, no hay valor aleatorio, simplemente se le asigna al evento el otro posible valor, ya que no tiene sentido la aleatoriedad con dos valores.

Adicionalmente, hay que tener en cuenta el atributo que modifica la forma: `isnumeric` (sección 5.2.1.6), ya que si está especificado los valores serán numéricos.

6.3.2.4. Fecha y Tiempo

Solo se han tenido en cuenta los operadores relacionales `{=,!=}` para estos tipos de datos simples. El comportamiento del *Componente de generación* de eventos cuando está involucrado el operador `=` es igual que en los casos anteriores, es decir se asigna el valor requerido. En el caso del operador `!=`, el valor generado tiene que cumplir las restricciones de la definición del tipo de evento y además tiene que ser diferente al valor que se indica en la consulta.

Además hay que tener en cuenta el atributo que modifica la forma, `format` (véanse secciones 5.2.1.7, 5.2.1.8), aunque en este caso es un atributo obligatorio para estos tipos de datos.

6.4. Aspectos técnicos de la implementación del generador IoT-TEG

Tal y como se ha comentado en el presente capítulo, el generador IoT-TEG está implementado en Java. La entrada de la herramienta es un fichero escrito en lenguaje `XML` que contiene la descripción del tipo de evento del que se quieren generar eventos de prueba. Para trabajar con este tipo de ficheros, se requiere de la API de Java *JDOM*[1], la cual permite manejar y recorrer los ficheros en este formato. La API de Java *JDOM* está desarrollada específicamente para dar soporte al tratamiento de `XML`: parseo, búsquedas, modificaciones, etc. Dado que está creado y optimizado para Java, hace que sea más eficiente y natural de usar para un desarrollador Java.

Otro aspecto importante dentro del generador IoT-TEG, es la generación de los valores que tendrán los diferentes atributos del evento. Para conseguir la aleatoriedad a la hora de generar los valores según el tipo de dato que defina al atributo del evento, tanto si el tipo de dato es simple como si es complejo, se utiliza la biblioteca de Java *Random*. Para los atributos de evento que se definan con un tipo de dato simple, en la misma definición del atributo de evento vendrá establecido el rango de valores en el que se tiene que generar el valor aleatorio (incluso cuando no lo especifica el usuario, se utiliza el rango por defecto establecido en cada tipo de dato). En el caso de un atributo de evento definido con un tipo de dato complejo, esta biblioteca servirá para cuando se utilicen cualquiera de los dos atributos opcionales `chooseone` o `dependence`. Cuando se emplea cualquiera de estos atributos en la definición de un tipo complejo, el generador tiene que escoger aleatoriamente uno de los elementos (`<field>` o `<attribute>`) que define al tipo complejo (véase sección 6.2.2.2).

Finalmente, otro aspecto técnico a considerar es en la funcionalidad avanzada con la que cuenta el generador IoT-TEG. Esta necesita almacenar los valores y operadores involucrados en las operaciones relacionales y lógicas que ha de localizar en las consultas EPL de EsperTech. Para ello se ha utilizado en el *componente de generación* de IoT-TEG un registro de tripletas cuya estructura es una ”lista de listas de tripletas”. Véase su declaración en Java en el código 6.38.

```java
List<List<Triplet<String, String, String>>>
```

Código 6.38: Definición Java de la estructura lista de listas de tripletas.

Esta estructura facilita la generación de conjuntos de eventos según las operaciones lógicas y relacionales que aparezcan en las consultas EPL de EsperTech, y del número de

[1]La versión que incluye IoT-TEG es la última versión estable 2.0.6

eventos de prueba solicitados por el usuario. En la sección 6.3.1 se muestra un ejemplo detallado de cómo se almacenan en listas de tripletas los valores y operadores según las operaciones lógicas encontradas.

6.5. Conclusiones

Se ha presentado IoT-TEG, una herramienta para la generación de eventos que engloba las últimas etapas del método para la generación automática de eventos de prueba propuesta en la presente tesis (véase sección 4.1). IoT-TEG genera eventos de cualquier tipo, siempre y cuando cumpla la especificación propuesta en el capítulo 5. IoT-TEG se ha desarrollado siguiendo los puntos que sugieren Saboor y Rengasamy en el trabajo [92] para la realización de pruebas, lo que ha permitido que sea una herramienta flexible ya que se adapta a las necesidades del usuario y de las pruebas.

La herramienta IoT-TEG tiene un comportamiento similar a los canales de información que incluyen las plataformas IoT, pero permite automatizar la generación de eventos de prueba haciendo que el usuario pueda:

- Elegir (sin límite) la cantidad de eventos a obtener.
- Definir el tipo de evento según sus necesidades (luego el tipo de evento está enfocado a cubrir las necesidades del usuario).
- Generar eventos con valores específicos.
- Obtener de forma inmediata los eventos para las pruebas.

Por todo lo anteriormente expuesto, y dado que IoT-TEG genera eventos en los tres formatos de salida más comunes `{JSON, CSV, XML}`, el programador no tendrá que implementar un generador de eventos específico para hacer pruebas en su programa. Además, gracias a que se pueden asignar valores específicos a los eventos, el usuario podrá hacer pruebas en funcionalidades específicas, consiguiendo un programa más robusto.

IoT-TEG consta de dos componentes: el *componente de validación*, un componente que comprueba que la definición del evento recibida cumple la especificación, y el *componente de generación*, que toma como entrada la especificación validada y el tipo de formato de salida.

Todas las definiciones de eventos tienen que cumplir la especificación propuesta, en caso contrario no se generarán los eventos solicitados y se mostrará un mensaje de error indicando qué parte de la especificación no cumple la definición.

Se ha detallado el recorrido de validación que realiza el *componente de validación*, que es el mismo que realiza el *componente de generación* para generar los valores. De este último se ha destacado cómo actúa frente a los dos tipos de atributos: los que afectan a la forma y los que afectan al valor de los eventos. Además se han mostrado ejemplos de eventos generados por la herramienta según los diferentes atributos que se contemplan en la especificación propuesta.

Se ha introducido una funcionalidad avanzada, que permite que los eventos generados cumplan algunas restricciones propias de las consultas involucradas en el programa para el cual se van a generar los eventos.

Las restricciones de las consultas que son consideradas para esta funcionalidad son aquellas operaciones relacionales y lógicas que aparecen en las cláusulas `where`. Los operadores lógicos `{OR, AND}` enlazan operaciones relacionales que tienen que seguir la estructura *"campo opRel valor"* además de ser campos de tipo simple (véase sección 5.2.1).

El proceso que sigue IoT-TEG para llevar a cabo esta funcionalidad:

1. Localización de operadores lógicos.
2. Almacenamiento en listas de tripletas (según los posibles operadores lógicos involucrados) de los elementos de las operaciones relacionales.
3. Filtrado de elementos de tipo de dato simple.
4. Cálculo de eventos a generar (con un grupo de eventos con valores totalmente aleatorio).
5. Generación de eventos, basándose en la definición del tipo de evento y en las restricciones de cada operación relacional involucrada.

El número de eventos de cada tipo se calcula según el número total de eventos que el usuario requiere dividiendo entre el número total de listas de tripletas + 1 (para tener un porcentaje aleatorio).

Según el tipo de dato simple se consideran una serie de operadores relacionales u otros a la hora de la generación de eventos. Independientemente de los operadores relacionales considerados, en todos se tienen que tener en cuenta cada uno de sus atributos específicos: los que determinan los posibles valores de los campos y los que determinan la forma del tipo de dato.

En el capítulo 9 se hace una comparativa de las salidas obtenidas por los casos de estudio utilizando sus "eventos originales"[2] y los generados por IoT-TEG, así mismo se comparan

[2] Se llaman "eventos originales" a aquellos que originalmente utiliza el caso de estudio; bien sea eventos generados por un generador de eventos propio, canales de eventos, ficheros con eventos, etc.

las salidas (en los casos que se cumplan las condiciones) con eventos generados por IoT-TEG utilizando la funcionalidad expuesta en este capítulo. Según los resultados se determinará no solo la utilidad de este método y su funcionalidad avanzada para realizar pruebas, sino también si el conjunto de eventos que se genere forma un caso de prueba de calidad.

Capítulo 7

Definición de operadores de mutación para EPL de EsperTech

Para aplicar la técnica de la prueba de mutaciones a programas escritos en un determinado lenguaje, se necesita disponer de un conjunto de operadores de mutación que van a definir los cambios a aplicar sobre el programa original y, por lo tanto, los mutantes que se pueden crear. Dado que hasta la elaboración de esta tesis doctoral no se habían definido operadores de mutación para el lenguaje EPL de EsperTech, el primer paso que se ha dado en este sentido ha sido la definición de los operadores de mutación. Este capítulo describe los operadores propuestos para EPL de EsperTech, los criterios que se han seguido a la hora de definirlos, su clasificación, nomenclatura y su comportamiento.

7.1. Introducción

Debido a la similitud de EPL de EsperTech con el lenguaje SQL, se utiliza como base para la definición de los operadores de mutación el estudio presentado por Tuya y col. [135], en el cual se definen los operadores de mutación del lenguaje SQL. De las 21 definiciones de operadores de mutación, algunas se han aplicado a EPL de EsperTech sin modificaciones, otras han tenido que adaptarse según la sintaxis de EPL de EsperTech y otras no se han considerado. Son ocho los operadores SQL descartados: UNI, ABS, NLS, NLI, NLO, IWR, IRT, IRP e IRH; cuatro los operadores adaptados de SQL (véase tabla 7.1) y nueve los operadores SQL que se han aplicado a EPL de EsperTech sin modificaciones (véase tabla 7.2). Los operadores de SQL se han adaptado modificando las palabras claves o símbolos propios del EPL de EsperTech, según el operador, pero la finalidad de la mutación es la misma. Algunos de los operadores de SQL que se han aplicado sin hacer modificaciones, se han dividido en varios operadores de mutación para EPL

OPERADOR SQL	OPERADOR EPL DE ESPERTECH
SEL	RSC
JOI	RJR
AGR	RAF
LCR	RLO

TABLA 7.1: Operadores de Mutación para EPL de EsperTech adaptados de SQL.

OPERADOR SQL	OPERADOR EPL DE ESPERTECH
SUB	RSR_1, RSR_2, RSR_3
GRU	RGR
ORD	ROM, ROS
ROR	RRO
UOI	RNO
AOR	RAO
BTW	RBR
LKE	RLM, RLA, RAW, RBW
NFL	RNW

TABLA 7.2: Operadores de Mutación para EPL de EsperTech sin modificar de SQL.

de EsperTech debido a la implementación seguida en la herramienta de generación de mutantes desarrollada (véase capítulo 8).

Adicionalmente se definen operadores propios del lenguaje EPL de EsperTech, teniendo como referencia la sintaxis del lenguaje publicada en [9], obteniéndose un total de 48 operadores de mutación para EPL de EsperTech. Como parte de este trabajo, en [158] se publica una primera clasificación los operadores de mutación de EPL de EsperTech, los cuales se dividen en 8 categorías, entre las categorías se encuentran propias del lenguaje EPL de EsperTech y otras similares a las del lenguaje SQL.

Dado que se pretende estudiar el comportamiento de programas que ejecuten consultas EPL de EsperTech, considerando aquellos cambios o fallos comunes propios de este lenguaje, se procede a analizar programas que ejecuten este tipo de consultas para hacer una recopilación de aquellos operadores de mutación más relevantes para el estudio. Con este análisis se visualizan no solo qué operadores, de los ya definidos, se pueden aplicar con más frecuencia, sino que también se definen nuevos operadores que no fueron considerados en [158], dando lugar a una segunda clasificación de operadores presentada en [159].

El lenguaje EPL de EsperTech, como ya se ha comentado en la sección 2.3.2, es de código abierto, y uno de los más utilizados en la actualidad en cuanto la gestión de eventos. Para realizar el análisis anterior sobre una gran cantidad de proyectos, se acude a una de las plataformas de desarrollo colaborativo más utilizadas a nivel mundial:

github[1]. Dentro de esta forja se buscan proyectos implementados en Java[2] que realicen consultas EPL de EsperTech. Se realiza una búsqueda de las líneas de código que se emplean para hacer una llamada de una consulta EPL de EsperTech (véase el código 7.1). El método `createEPL`, como ya se explicó en la sección 2.4.2, pertenece a la interfaz `EPAdministrator`, la cual devuelve instancias EPStatement (declaraciones EPL), que se lanzan y activan automáticamente una vez creadas.

```
epService.getEPAdministrator().createEPL("...");
```

Código 7.1: Llamada de Java para hacer una consulta EPL de EsperTech.

Tras realizar una búsqueda en los códigos de los proyectos alojados en github del método "`createEPL`", se encontraron más de 3700 resultados. Esto es, más de 3700 consultas EPL de EsperTech que han servido para detectar y determinar qué operadores de mutación son los más relevantes para este estudio.

Después de hacer estas observaciones y tras revisar las definiciones de los 48 operadores de mutación iniciales, se acuerda agrupar aquellos operadores de mutación que efectúan la mutación sobre la misma cláusula o parte del patrón EPL de EsperTech. Quedándose un total de 34 operadores de mutación que se clasifican en cuatro categorías. Las categorías se han determinado según el elemento de la consulta con el que está relacionado el cambio. Las categorías se identifican con una letra mayúscula y son las siguientes:

- **P** - Operadores de expresiones de patrones
- **W** - Operadores de ventanas
- **R** - Operadores de reemplazo
- **I** - Operadores de inyecciones de ataque de SQL

Dentro de cada categoría se definen varios operadores de mutación que se identifican mediante tres letras mayúsculas: la primera de ellas se corresponde con la categoría a la que pertenece el operador, mientras que las dos últimas identifican al operador dentro de la categoría.

Al definir los operadores se ha puesto especial cuidado en evitar que produzcan muchos mutantes equivalentes y también *mutantes no válidos*. Los *mutantes no válidos* son aquellos que no se pueden ejecutar porque violan alguna de las restricciones de sintaxis de EPL de EsperTech.

[1] Website: `http://www.github.com`

[2] Lenguaje de programación que se utiliza en Esper para lanzar las consultas EPL de EsperTech

Las tablas 7.3 y 7.4 muestran el nombre y una descripción breve de los operadores. Se han marcado con ☆ los operadores específicos de EPL de EsperTech que no aparecen en otros lenguajes. Los operadores de reemplazo pueden aparecer en cualquier parte de la consulta, incluso en los patrones. Es por esto por lo que su definición se extiende para cualquier parte de la consulta.

7.2. Operadores de expresiones de patrones

Un error muy común a la hora de escribir los patrones EPL, debido al desconocimiento de la utilidad de los paréntesis, es incluir o no incluir estos en el patrón cuando se quiere utilizar el operador `->` (nombrado como *followed by*). Si se considera la siguiente secuencia de eventos:

$A_1, B_1, C_1, B_2, A_2, D_1, A_3, B_3, E_1, A_4, F_1, B_4$

Si el patrón se ha definido como `every (A -> B)`, este detecta un evento A seguido de un evento B. Una vez que el evento B ocurra, el patrón comienza de nuevo buscando el siguiente evento A. Según la secuencia de eventos anterior:

- Coincide en B_1 por la combinación $\{A_1, B_1\}$
- Coincide en B_3 por la combinación $\{A_2, B_3\}$
- Coincide en B_4 por la combinación $\{A_4, B_4\}$

Si el patrón se ha definido como `every A -> B`, el patrón se dispara por cada evento A seguido de un evento B. Según la secuencia anterior:

- Coincide en B_1 por la combinación $\{A_1, B_1\}$
- Coincide en B_3 por la combinación $\{A_2, B_3\}$ y $\{A_3, B_3\}$
- Coincide en B_4 por la combinación $\{A_4, B_4\}$

Por lo tanto, la mutación que se lleva acabo con el operador **PFP** estando definido el patrón como `every A -> B`, sería obtener el patrón `every (A -> B)` y al contrario si este está definido con los paréntesis. La figura 7.1 muestra un ejemplo de su aplicación.

Operador		Descripción
		Operadores de expresiones de patrones
PFP	☆	Añade o elimina los paréntesis a los eventos especificados en el operador -> (nombrado como *followed by*)
PGR	☆	Elimina las condiciones “where” de los patrones (conocidas como expresiones *guards*)
PNR	☆	Elimina la palabra clave not de las expresiones de patrones con condicionales negados
POC	☆	Cambia el orden de los eventos especificados en el operador -> (*followed by*)
POM	☆	Modifica el valor del observador de patrón (timer:at, timer:interval) incrementándolo o decrementándolo en una unidad
PRE	☆	Elimina el patrón de la consulta, sustituyéndolo por la expresión de filtrado que aparece en el patrón
		Operadores de reemplazamiento
RAF		Reemplaza una función de agregación (max, min, avg, sum, count, median, stddev, avedev) por otra. La palabra reservada distinct puede añadirse en la llamada a la función
RAO		Reemplaza un operador aritmético (+, -, *, /, %) por otro
RBR		Reemplaza cada condición (a between x AND y), por expresiones relacionales (a > x AND a <= y, a >= x AND a < y) y su negativa
RGR		Elimina una expresión ”group by“ y se añade su correspondiente función de agregación (max, min)
RJR		Reemplaza las palabras claves de una cláusula ”join“ (inner, left outer, right outer, full outer, outer) por otra
RLM		Reemplaza, en los patrones de expresión de la palabra reservada ”like“, un carácter comodín (%, _) por otro
RLA		Añade al principio y al final, de los patrones de expresión de la palabra reservada ”like“, un carácter comodín (%, _)
RAW		Elimina del principio del patrón de expresión de la palabra reservada ”like“, un carácter comodín (%, _)
RBW		Elimina del final del patrón de expresión de la palabra reservada ”like“, un carácter comodín (%, _)
RLO		Reemplaza un operador lógico (and, or) por otro
RNO		Reemplaza un número e por e + 1 y e - 1
RNW		Reemplaza cada ocurrencia (is null, is not null) por otra
ROM		Reemplaza, o añade en caso de no existir, la palabra reservada (asc, desc) por otra en las expresiones ”order by“
ROS		Intercambia el orden de las propiedades de las expresiones ”order by“
RRO		Reemplaza un operador relacional (=, <>, <, >, <=, >=) por otro
RRR_1	☆	Reemplaza una función ”single row” {(cast, instanceof), (prevwindow, prevcount)} por otra
RRR_2	☆	Reemplaza una función ”single row” (prev, prevtail, prior) por otra
RSC		Reemplaza, o añade en caso de no existir, en la cláusula “select” una palabra reservada (rstream, irstream). La palabra reservada distinct puede añadirse o combinarse
RSR_1		Reemplaza, en las subconsultas de tipo I, la palabra reservada (all, any, some) por otra de las subconsultas de tipo I, de tipo II o de tipo III, con las modificaciones adicionales oportunas
RSR_2		Reemplaza, en las subconsultas de tipo II, la palabra reservada (in, not in) por otra de las subconsultas de tipo II, de tipo I o de tipo III, con las modificaciones adicionales oportunas
RSR_3		Reemplaza, en las subconsultas de tipo III, la palabra reservada (exist, not exist) por otra
RTU	☆	Reemplaza una unidad de tiempo (milliseconds, seconds, minutes, hours, days) por otra

Tabla 7.3: Operadores de Mutación para EPL de EsperTech de patrones y de reemplazamiento.

Operador		Descripción
		Operadores de ventanas
WLM	☆	Modifica el valor de la ventana de longitud incrementándolo o decrementándolo en una unidad
WTM	☆	Modifica el valor de la ventana de tiempo incrementándolo o decrementándolo en una unidad
WBL	☆	Cambia una ventana por lotes de longitud por una ventana de longitud ordinaria
WBT	☆	Cambia una ventana por lotes de tiempo por una ventana de tiempo ordinaria
		Operadores de inyecciones de ataque de SQL
ICN		Niega una expresión condicional de la consulta
IWR	☆	Elimina la condición “`where`” de la consulta

Tabla 7.4: Operadores de Mutación para EPL de EsperTech de ventanas y de inyecciones de ataque de SQL.

```
Código original:                      Código mutado por PFP:

every A=mypack.RfidEvent ->           every (A=mypack.RfidEvent ->
B=mypack.RfidEvent(itemId =           B=mypack.RfidEvent(itemId =
A.itemId)                             A.itemId))
```

Figura 7.1: Ejemplo de mutante generado por PFP.

Las expresiones *guards* son condiciones `where` localizadas en los patrones (véase sección 2.4.2, tabla 2.5) que filtran los eventos y causan la terminación del buscador de patrones. El operador **PGR** elimina esta condición `where` al completo, haciendo que no se produzca esa terminación en el buscador. La figura 7.2 muestra un ejemplo de su aplicación.

```
Código original:                      Código mutado por PGR:

every (StEvent(status='ERROR')        every (StEvent(status='ERROR')
-> StEvent(status='ERROR')            -> StEvent(status='ERROR')
where timer:within (10000))
```

Figura 7.2: Ejemplo de mutante generado por PGR.

El operador **PNR** elimina la negación de las posibles expresiones condicionales que se encuentren en un patrón. Esto se consigue quitando la palabra `not` que antecede a la condición, modelando el error que se cometería al olvidar la negación de la condición. La figura 7.3 muestra un ejemplo de su aplicación.

Como ya se comentó en el operador de mutación **PFP**, el operador `->` ayuda a definir la detección de los eventos. Con el operador **POC** lo que hacemos es intercambiar el

Código original:	*Código mutado por PNR:*
`every timer:interval(10 seconds) and not Status(terminal.id = 'T1')`	`every timer:interval(10 seconds) and Status(terminal.id = 'T1')`

FIGURA 7.3: Ejemplo de mutante generado por PNR.

orden de los eventos que tienen que detectar los patrones, es decir si el patrón se define `every A -> B`, el operador devolvería `every B -> A`. La figura 7.4 muestra un ejemplo de aplicación del operador **POC**.

Código original:	*Código mutado por POC:*
`every A=mypack.RfidEvent -> B=mypack.RfidEvent(itemId = A.itemId)`	`every B=mypack.RfidEvent(itemId = A.itemId) -> A=mypack.RfidEvent`

FIGURA 7.4: Ejemplo de mutante generado por POC.

Un error común que se podría cometer al escribir un patrón dentro de una consulta EPL de EsperTech es modificar el valor de la unidad temporal de los observadores de patrones. Los valores temporales afectan a las ventanas de tiempo las cuales, con una leve modificación como la que se quiere representar, pueden variar el comportamiento de la consulta en las que están involucradas. El operador **POM** modela los errores que pueden cometerse al introducir el valor de la unidad temporal (una constante numérica), modificándola de varias formas:

- Se incrementa su valor en una unidad
- Se decrementa su valor en una unidad

Para evitar la aparición de mutantes no válidos y dado que el valor modificado es una unidad temporal, se ha considerado el siguiente caso especial para el operador **POM**. Cuando el valor de la unidad temporal vale 0, el mutante no se genera ya que no existen los tiempos negativos. La figura 7.5 muestra un ejemplo de aplicación del operador **POM**.

Código original:	*Código mutado por POM:*
`every timer:interval(10 seconds)`	`every timer:interval(9 seconds)`

FIGURA 7.5: Ejemplo de mutante generado por POM.

El operador de mutación **PRE** elimina el patrón de la consulta, sustituyéndolo por una expresión de filtrado. Este operador modela un tipo de error del programador difícil de detectar, ya que este tipo de consultas son equivalentes cuando el patrón tiene únicamente el operador `every` y una única expresión de filtrado, en la figura 7.6 se muestra un ejemplo de consulta equivalente:

```
Código original:

select * from pattern[every
StockTickEvent(symbol='GE')]
```

```
Código mutado por PRE:

select * from
StockTickEvent(symbol='GE')
```

Figura 7.6: Ejemplo de mutante generado por PRE.

7.3. Operadores de reemplazo

El lenguaje EPL de EsperTech utiliza varios tipos de expresiones para las que se han definido operadores de mutación. Solo se han considerado aquellos operadores de mutación que sustituyen en las expresiones un operador por otro del mismo tipo, ya que estos son los fallos que pueden cometerse con mayor frecuencia. Para definir estos operadores se han tenido en cuenta operadores similares definidos para otros lenguajes [126, 127, 135].

A continuación se describen todos los operadores de mutación de expresiones y se dan ejemplos de algunos de ellos.

El operador de mutación **RRO** reemplaza, en una expresión relacional, un operador relacional {`=, <>, <, >, <=, >=`} por otro del mismo tipo.

El operador **RAO** reemplaza, en una expresión aritmética, un operador aritmético {`+, -, *, /, %`} por otro del mismo tipo.

Dado que los operadores de mutación **RAO** y **RRO** son muy similares, diferenciándose solo en el tipo de operador sobre el que actúan, la figura 7.7 muestra un único ejemplo para ellos.

```
Código original:

select * from ProductOrder as
ord where quantity < 100
```

```
Código mutado por RRO:

select * from ProductOrder as
ord where quantity > 100
```

Figura 7.7: Ejemplo de mutante generado por RRO.

El operador de mutación **RLO** reemplaza, en una expresión lógica, un operador lógico {`and`, `or`} por otro del mismo tipo. Este operador actúa como los operadores de mutación anteriores, pero con la siguiente diferencia: Si una expresión lógica es compuesta y todos los operadores lógicos son iguales, el árbol sintáctico que genera Esper considera esta expresión lógica como una sola, es decir, habría un único nodo para toda la expresión lógica. Con lo cual, dado que el analizador de MuEPL (véase sección 8.2.2) trabaja sobre el árbol sintáctico, se detectará una única ocurrencia donde aplicar el operador y todos los operadores lógicos involucrados cambiarían. En la figura 7.8 se muestra un ejemplo de su aplicación.

```
Código original:                    Código mutado por RLO:

insert into cep.prd.stock           insert into cep.prd.stock
(id, price, stock)                  (id, price, stock)
select a.id, a.price, a.stock       select a.id, a.price, a.stock
from pattern [every a = cep.prd     from pattern [every a = cep.prd
->(id = a.id and price = a.price    ->(id = a.id or price = a.price
and stock = a.stock)]               or stock = a.stock)]
```

FIGURA 7.8: Ejemplo de mutante generado por RLO.

El operador de mutación **RAF** reemplaza una función de agregación {`max, min, avg, sum, count, median, stddev, avedev`}, vistas en la sección 2.4.1, por otra del mismo tipo. Adicionalmente la palabra reservada `distinct` puede añadirse en la llamada a la función, haciendo que esta no utilice los valores repetidos en la operación a aplicar. Las funciones de agregación pueden aplicarse a eventos que estén en una ventana de flujos de eventos, o cualquier vista, o a uno o más grupos de eventos. La figura 7.9 muestra un ejemplo de aplicación del operador **RAF** (se muestran 3 mutaciones de las 15 posibles, hay que recordar que la palabra reservada `distinct` puede incluirse en estas funciones).

```
Código original:                    Código 1 mutado por RAF:

select min(latencyAC) from          select avg(latencyAC) from
CombinedEvent.win:time(30 min)      CombinedEvent.win:time(30 min)


Código 2 mutado por RAF:            Código 3 mutado por RAF:

select max(latencyAC) from          select min(distinct latencyAC) from
CombinedEvent.win:time(30 min)      CombinedEvent.win:time(30 min)
```

FIGURA 7.9: Ejemplo de mutante generado por RAF.

A continuación se definen los operadores de mutación que se aplican en la palabra reservada ”`like`”:

RLM Reemplaza, en los patrones de expresión de la palabra reservada ”`like`“, un carácter comodín {%, _} por otro.

RLA Añade al principio o al final, de los patrones de expresión de la palabra reservada ”`like`“, un carácter comodín {%, _}.

RAW Elimina del principio del patrón de expresión de la palabra reservada ”`like`“, un carácter comodín {%, _}.

RBW Elimina del final del patrón de expresión de la palabra reservada ”`like`“, un carácter comodín {%, _}.

Dado que actúan sobre una misma zona de la consulta, se muestra en la figura 7.10 un único ejemplo de cada uno de los operadores que se pueden aplicar en la consulta utilizada.

```
Código original:                      Código mutado por RLM:

select * from PersonLocatEvent        select * from PersonLocatEvent
where name like 'Tom%'                where name like 'Tom_'


Código mutado por RLA:                Código mutado por RBW:

select * from PersonLocatEvent        select * from PersonLocatEvent
where name like '%Tom%'               where name like 'Tom'
```

FIGURA 7.10: Ejemplo de mutante generado por RLM, RLA y RBW.

El operador de mutación **RNO** reemplaza un número `e` por `e + 1` y `e - 1`. Este número no ha de confundirse con el valor de la unidad de tiempo de los observadores de patrones que se modifican con el operador **POM** y los valores numéricos que modifican algunos de los operadores de ventanas: **WLM** y **WTM**. La figura 7.11 muestra un ejemplo de aplicación del operador **RNO**.

Los operadores de mutación **RRR**$_1$ y **RRR**$_2$ actúan sobre las denominadas funciones ”single row”, específicas de EPL de EsperTech, y devuelven un único valor por cada fila de resultados generados por la consulta. A continuación se definen los operadores de mutación y se especifican los posibles reemplazos entre funciones:

```
Código original:                      Código 1 mutado por RNO:

select * from pattern[                select * from pattern[
every a=name (cnt in [1:2])]          every a=name (cnt in [0:2])]

                    Código 2 mutado por RNO:

                    select * from pattern[
                    every a=name (cnt in [2:2])]
```

FIGURA 7.11: Ejemplo de mutante generado por RNO.

RRR$_1$ Reemplaza cada ocurrencia de la función `cast` por la función `instanceof` (y viceversa), así como el reemplazo de cada ocurrencia de la función `prevwindow` por `prevcount` (y viceversa)

RRR$_2$ Reemplaza cada ocurrencia de las funciones `prev, prevtail, prior` por cualquiera de las otras.

En la figura 7.12 se observa un ejemplo de aplicación del operador **RRR**$_2$.

```
Código original:                      Código 1 mutado por RRR2:

select prev(1, symbol) from           select prevtail(1, symbol) from
Trade.win:time_batch(1 min)           Trade.win:time_batch(1 min)

                    Código 2 mutado por RRR2:

                    select prior(1, symbol) from
                    Trade.win:time_batch(1 min)
```

FIGURA 7.12: Ejemplo de mutante generado por RRR$_2$.

Otro conjunto de operadores que trabajan sobre la misma parte de la consulta son **RSR**$_1$, **RSR**$_2$ y **RSR**$_3$. Las subconsultas tienen generalmente la forma $e\Re p(Q)$, donde e es un atributo o expresión, $\Re$ es un operador relacional, p es una palabra reservada y Q es la subconsulta. A continuación se especifican los cambios de cada uno de los tipos de subconsultas y las palabras reservadas de cada uno:

RSR$_1$ Reemplaza, en las subconsultas de tipo I (con la forma $e\Re p(Q)$), la palabra reservada {`all`, `any`, `some`} por otra de las subconsultas de tipo I, de tipo II o de tipo III, con las modificaciones adicionales oportunas.

RSR$_2$ Reemplaza, en las subconsultas de tipo II (con la forma *e p(Q)*), la palabra reservada {`in`, `not in`} por otra de las subconsultas de tipo II, de tipo I (considerando los diferentes operadores relacionales) o de tipo III, con las modificaciones adicionales oportunas.

RSR$_3$ Reemplaza, en las subconsultas de tipo III (con la forma *p(Q)*), la palabra reservada {`exist`, `not exist`} por otra.

En la figura 7.13 se muestra un ejemplo de 3 posibles mutaciones de los 6 posibles.

```
Código original:                      Código 1 mutado por RSR1:

select * from ProductOrder as         select * from ProductOrder as
ord where quantity < any              ord where quantity < all
(select minimumQuantity from          (select minimumQuantity from
MinimumQuantity.win:keepall())        MinimumQuantity.win:keepall())

Código 2 mutado por RSR1:             Código 3 mutado por RSR1:

select * from ProductOrder as         select * from ProductOrder as
ord where quantity in                 ord where exist
(select minimumQuantity from          (select minimumQuantity from
MinimumQuantity.win:keepall())        MinimumQuantity.win:keepall())
```

FIGURA 7.13: Ejemplo de mutante generado por RSR$_1$.

El operador **RGR** elimina una expresión ”`group by`“ y se añade su correspondiente función de agregación {`max`, `min`} en la cláusula `select`. En la figura 7.14 se muestra un ejemplo de aplicación del operador.

```
Código original:                      Código 1 mutado por RGR:

select symbol, avg(price)             select max(symbol), avg(price)
from StockTickEvent                   from StockTickEvent
group by symbol
```

FIGURA 7.14: Ejemplo de mutante generado por RGR.

La condición “`a between x AND y`” es equivalente a “`a >= x AND a <= y`”. Para simular un error de programación, el operador de mutación **RBR** va a reemplazar cada condición “`a between x AND y`” por las expresiones relacionales “`a > x AND a <= y`”, “`a >= x AND a < y`” y su negativa. En la figura 7.15 se muestra un ejemplo de las mutaciones.

```
Código original:

select * from StockTickEvent
where price
between 55 and 60

Código 1 mutado por RBR:

select * from StockTickEvent
where price > 55 and
price <= 60

Código 2 mutado por RBR:

select * from StockTickEvent
where price >= 55 and
price < 60

Código 3 mutado por RBR:

select * from StockTickEvent
where price not
between 55 and 60
```

FIGURA 7.15: Ejemplo de mutante generado por RBR.

El operador de mutación **RJR** reemplaza las palabras claves de una cláusula ”`join`“ {`inner`, `left outer`, `right outer`, `full outer`, `outer`} por otra. En la figura 7.16 se muestra un ejemplo de aplicación del operador.

```
Código original:

select * from
TickEvent.std:unique(symbol) as t
full outer join
NewsEvent.std:unique(symbol) as n
on t.symbol = n.symbol

Código 1 mutado por RJR:

select * from
TickEvent.std:unique(symbol) as t
left outer join
NewsEvent.std:unique(symbol) as n
on t.symbol = n.symbol

Código 2 mutado por RJR:

select * from
TickEvent.std:unique(symbol) as t
right outer join
NewsEvent.std:unique(symbol) as n
on t.symbol = n.symbol

Código 3 mutado por RJR:

select * from
TickEvent.std:unique(symbol) as t
outer join
NewsEvent.std:unique(symbol) as n
on t.symbol = n.symbol
```

FIGURA 7.16: Ejemplo de mutante generado por RJR.

El operador de mutación **RSC** reemplaza en la cláusula “`select`” la palabra reservada {`rstream`, `irstream`} por otra. En caso de no existir ninguna palabra reservada se añade cualquiera de ellas. Por defecto el motor suministra el flujo de eventos de inserción generados por la consulta, lo cual es equivalente a poner la palabra reservada `istream` en la cláusula `select`. Con la palabra reservada `rstream` el motor suministra el flujo de eventos de borrado generados por la consulta (los eventos que abandonan las ventanas a

las que se hace referencia en la consulta). Con la palabra reservada `irstream` el motor inserta ambos tipos de flujos de eventos. Por otro lado la palabra reservada `distinct` puede añadirse o combinarse. La figura 7.17 muestra un ejemplo de las mutaciones.

```
Código original:                    Código 1 mutado por RSC:

select rstream *                    select irstream *
from StockTick.win:time(30 sec)     from StockTick.win:time(30 sec)

                 Código 2 mutado por RSC:

                 select rstream distinct *
                 from StockTick.win:time(30 sec)
```

FIGURA 7.17: Ejemplo de mutante generado por RSC.

El operador **RNW** reemplaza cada ocurrencia {`is null`, `is not null`} por otra. En la figura 7.18 se muestra un ejemplo de aplicación del operador.

```
Código original:               Código 1 mutado por RNW:

select symbol                  select symbol
from StockTickEvent            from StockTickEvent
where price is null            where price is not null
```

FIGURA 7.18: Ejemplo de mutante generado por RNW.

Los dos siguientes operadores se aplican en la cláusula ''`order by`''. Por un lado el operador de mutación **ROM** reemplaza la palabra reservada {`asc`, `desc`} por otra en las expresiones ''`order by`''. En caso de no existir, la palabra reservada que se añade es `desc` ya que por defecto el orden es ascendente `asc`. Y el operador **ROS** intercambia el orden de las propiedades de las expresiones ''`order by`'' si existen en la consulta más de una propiedad involucrada. En la figura 7.19 se muestra un ejemplo del operador **ROM**.

El último operador de esta categoría es **RTU**, que tiene como dominio las unidades temporales que aparecen en la consulta. El operador reemplaza la unidad de tiempo {`milliseconds, seconds, minutes, hours, days`} por otra del mismo tipo en cualquier expresión temporal que aparezca en la consulta. Nótese que las unidades de tiempo pueden ser escrita de forma abreviada (sec, min, hr...), algo que no afecta al hacer la mutación. La figura 7.20 muestra un ejemplo de aplicación del operador (se muestran 3 mutaciones de los 4 posibles).

```
Código original:

select symbol from
StockTickEvent.win:time(60 sec)
output every 5 events
order by price desc, volume

Código 1 mutado por ROM:

select symbol from
StockTickEvent.win:time(60 sec)
output every 5 events
order by price asc, volume

Código 2 mutado por ROM:

select symbol from
StockTickEvent.win:time(60 sec)
output every 5 events
order by price desc, volume desc
```

FIGURA 7.19: Ejemplo de mutante generado por ROM.

```
Código original:

select type from
BTerminalEvent.win:time(10 min)

Código 1 mutado por RTU:

select type from
BTerminalEvent.win:time(10 sec)

Código 2 mutado por RTU:

select type from
BTerminalEvent.win:time(10 days)

Código 3 mutado por RTU:

select type from
BTerminalEvent.win:time(10 msec)
```

FIGURA 7.20: Ejemplo de mutante generado por RTU.

7.4. Operadores de ventanas

Como se vio en la sección 2.4.1 existen ventanas o vistas de longitud o de tiempo. El parámetro que reciben estas ventanas es un número que determina el tamaño de la ventana (si esta es de longitud), o determina el intervalo de tiempo de la ventana (si esta es de tiempo). A continuación se describen todos los operadores de mutación que se aplican en ventanas o vistas:

El operador de mutación **WLM** incrementa y decrementa en una unidad el valor numérico del tamaño que tiene como parámetro la ventana o vista de longitud. Para evitar la aparición de mutantes no válidos y dado que el valor modificado es el tamaño de la ventana, se ha considerado el siguiente caso especial para el operador. Cuando el valor del tamaño de la ventana vale 0, no se genera el mutante que decrementa el valor ya que no existen tamaños negativos.

El operador **WTM** incrementa y decrementa en una unidad el valor numérico de la unidad de tiempo que tiene como parámetro la ventana o vista de tiempo. Para evitar la aparición de mutantes no válidos y dado que el valor modificado es el valor de una unidad de tiempo, se ha considerado el siguiente caso especial para el operador: cuando el valor de la unidad de tiempo vale 0, no se genera el mutante que decrementa el valor ya que no existen tiempos negativos.

Dado que los operadores de mutación **WLM** y **WTM** son muy similares, diferenciándose solo en el tipo de ventana donde actúan, la figura 7.21 muestra un único ejemplo para todos ellos.

```
Código original:                         Código mutado por WLM:

select * from Withdrawal(amount          select * from Withdrawal(amount
>= 200).win:length(5)                    >= 200).win:length(6)
```

FIGURA 7.21: Ejemplo de mutante generado por WLM.

El operador de mutación **WBL** cambia una ventana por lotes de longitud por una ventana de longitud ordinaria. Esto simula el error, por parte del programador, de olvidar añadir "`_batsh`" en la declaración de la ventana de longitud, quedándose una declaración de una ventana de longitud ordinaria. No se considera el cambiar una ventana ordinaria a una ventana por lotes ya que modelaría un fallo poco frecuente en EPL de EsperTech.

El operador **WBT** cambia una ventana por lotes de tiempo por una ventana de tiempo ordinaria. Esto simula el error, por parte del programador, de olvidar añadir "`_batsh`" en la declaración de la ventana de tiempo, quedándose una declaración de una ventana de tiempo ordinaria. No se considera el cambiar una ventana ordinaria a una ventana por lotes ya que modelaría un fallo poco frecuente en EPL de EsperTech.

Los dos operadores de mutación **WBL** y **WBT** realizan el mismo cambio diferenciándose solo en el tipo de ventana donde actúan. Por esto se muestra en la figura 7.22 un único ejemplo para ambos.

```
Código original:                         Código mutado por WBT:

select * from                            select * from
Withdrawal.win:time_batch(4 sec)         Withdrawal.win:time(4 sec)
```

FIGURA 7.22: Ejemplo de mutante generado por WBT.

7.5. Operadores de inyecciones de ataque de SQL

Tal y como se vio en la sección 2.4.1 la cláusula `where` es opcional. Así, el operador **IWR** elimina esta cláusula de la consulta. Hay que tener en cuenta que este operador no elimina las expresiones *guards* explicadas en la sección 2.4.2, de estas se encarga el operador **PGR** (véase sección 7.2). En la figura 7.23 muestra un ejemplo de aplicación del operador **IWR**.

```
Código original:                  Código mutado por IWR:

select * from                     select * from
Withdrawal.win:length(5)          Withdrawal.win:length(5)
where amount >= 100
```

FIGURA 7.23: Ejemplo de mutante generado por IWR.

El otro operador de mutación que pertenece a esta categoría es **ICN**, que niega una expresión condicional de la consulta. Este operador, aunque no sea específico del lenguaje EPL de EsperTech, no está contemplado en el conjunto de operadores de SQL de Tuya et al. [135]. En la figura 7.24 se muestra un ejemplo de aplicación del operador **ICN**.

```
Código original:                  Código mutado por ICN:

select * from BTerminalEvent      select * from BTerminalEvent
where type = 'LowPaper' or        where not type = 'LowPaper' or
type = 'OutOfOrder'               type = 'OutOfOrder'
```

FIGURA 7.24: Ejemplo de mutante generado por ICN.

7.6. Conclusiones

Tras un primer estudio donde se definieron 48 operadores de mutación para EPL [158], se analizaron más de 3700 consultas procedentes de proyectos reales alojados en la forja github. Este análisis permitió identificar y detectar el conjunto de operadores para EPL de EsperTech más relevantes que modelan los fallos más comunes que pueden cometer los programadores a la hora de implementar una consulta EPL de EsperTech. Después de hacer estas observaciones y tras revisar las definiciones de los 48 operadores de mutación iniciales, se acuerda agrupar aquellos operadores de mutación que efectúan la mutación sobre la misma cláusula o parte del patrón EPL de EsperTech. Como consecuencia se ha definido y presentado en [159] un conjunto de 34 operadores de mutación para EPL de

EsperTech que han sido clasificados en 4 categorías: mutación de expresiones de patrones, mutación de ventanas, mutaciones basadas en reemplazo, mutaciones que simulan inyecciones de ataque de SQL. Entre estos operadores encontramos específicos de EPL de EsperTech y otros, dada la similitud entre lenguajes, basados en SQL. En el capítulo 9 se analizarán los resultados obtenidos con los casos de estudio empleados en la presente tesis (véase apéndice B), donde se justifica la necesidad de definición de operadores propios de EPL de EsperTech.

Finalmente, al definir los operadores se ha puesto especial cuidado en evitar que produzcan muchos mutantes equivalentes o mutantes no válidos.

Capítulo 8

Generador de mutantes MuEPL

La automatización de la aplicación de la prueba de mutaciones requiere disponer de una herramienta que genere automáticamente los mutantes, los ejecute frente a un conjunto de casos de prueba y compare el comportamiento de los mutantes y el programa original para los diferentes casos de prueba. En este capítulo se presenta la herramienta MuEPL que permite aplicar la técnica de mutación a las consultas EPL de EsperTech.

8.1. Introducción

La aplicación automática de la prueba de mutaciones a programas escritos en un determinado lenguaje hace necesario disponer de una herramienta que genere los mutantes, los ejecute frente a los casos de prueba y decida si un mutante ha muerto o no mediante la comparación de su comportamiento con el del programa original para cada caso de prueba. Como resultado de esta tesis se ha implementado la herramienta MuEPL, que permite aplicar la prueba de mutaciones a consultas EPL de EsperTech.

MuEPL presenta las siguiente funcionalidades:

- Analiza consultas EPL de EsperTech, produciendo como salida los operadores de mutación que se les pueden aplicar y el número de localizaciones dentro de las consultas donde estos son aplicables.
- Genera todos los posibles mutantes mediante la aplicación de los operadores de mutación definidos en el capítulo 7.
- Ejecuta las consultas originales y las consultas mutantes frente al conjunto de casos de prueba.

- Compara el comportamiento de las consultas mutadas con el comportamiento de las originales, generando la matriz de ejecución.

8.2. Arquitectura de MuEPL

MuEPL consta de cuatro componentes principales:

- Un capturador de consultas.
- Un analizador.
- Un generador de mutantes.
- Un sistema de ejecución.

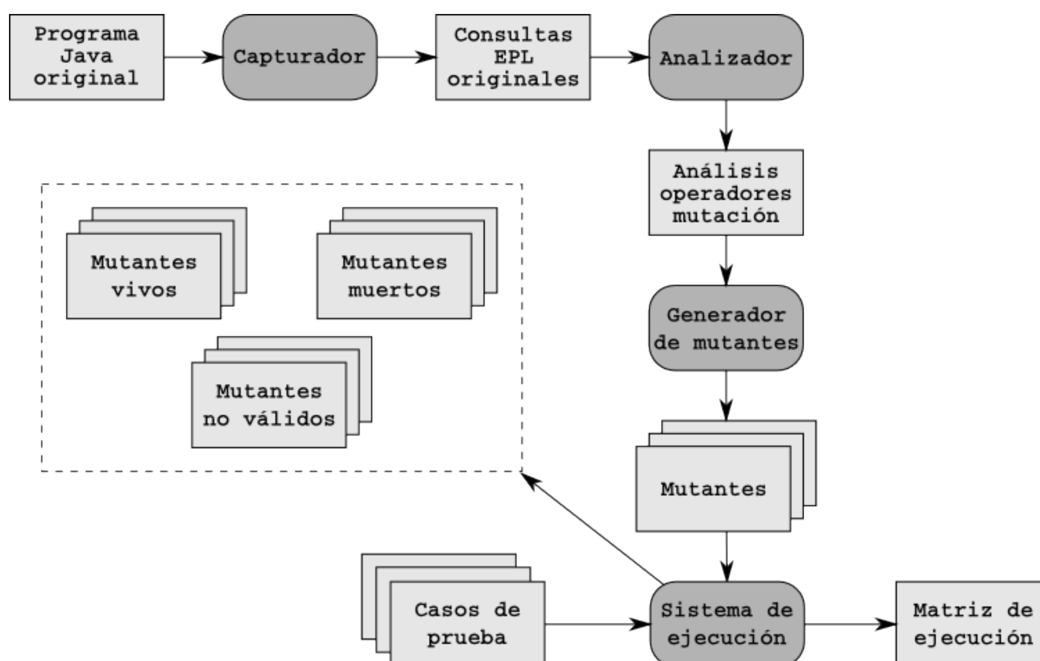

Figura 8.1: Arquitectura MuEPL.

La figura 8.1 muestra la relación entre estos componentes. El *capturador de consultas* obtiene las consultas EPL de EsperTech completas tras la ejecución del programa original Java que se esté estudiando. El programa ha de ejecutarse al menos una vez para obtener las consultas al completo, ya que de este modo se adquieren los posibles valores de las variables que solo se consiguen en tiempo de ejecución (explicado en la sección 8.1). A continuación el *analizador* recibe las consultas originales y nos indica qué operadores de mutación pueden aplicarse y su localización. Esta información la recibe el *generador de*

mutantes que genera todos los posibles mutantes. Posteriormente, el motor de ejecución ejecuta las consultas originales y los mutantes frente al conjunto de casos de prueba, comparando después sus comportamientos, lo que nos permite determinar si los mutantes han muerto o permanecen vivos. Por otro lado, pueden generarse mutantes *no válidos*, que son aquellos que no se pueden ejecutar porque violan alguna de las condiciones de las consultas EPL de EsperTech.

En las siguientes secciones se describirán cada uno de los componentes, su funcionamiento, ejemplos de su uso y resultados.

8.2.1. Capturador

Este componente es específico para todos aquellos programas cuyas consultas son creadas y lanzadas desde el propio programa Java que se esté estudiando. Para las aplicaciones cuyas consultas están almacenadas en ficheros de manera independiente del programa que las lanza, no haría falta capturar las consultas y, por tanto, pasarán directamente al siguiente componente de MuEPL, el *analizador*.

Una de las dificultades de las consultas EPL de EsperTech, a diferencia de otros lenguajes, es que las consultas EPL de EsperTech pueden estar escritas por partes en la implementación, o bien obtener valores en tiempo de ejecución. Por lo que para poder dotar a MuEPL de las funcionalidades anteriormente descritas, se ha desarrollado una funcionalidad que captura las consultas involucradas en las aplicaciones Java de forma completa. Para ello se ha modificado la clase de Esper "`EPAdministratorImpl.java`", la cual se ubica dentro de la biblioteca de Esper. Esta biblioteca modificada de Esper, ha de sustituir a la biblioteca Esper que utilice el programa Java a analizar (programa que contiene las consultas EPL de EsperTech), de este modo se podrá llevar a cabo la captura de consultas. El conjunto de consultas EPL de EsperTech capturadas, se almacenarán en un fichero `''queries.epl''` que se guardará en el lugar que indique el usuario, las consultas aparecerán según su orden de ejecución en el programa.

Dentro de la clase "`EPAdministratorImpl.java`" está la función "`createEPL`" que compone y lanza las consultas EPL de EsperTech una vez que estas reciben todos los valores involucrados en la misma (véase sección 2.4.2). Es decir, para la consulta que se muestra en código 8.1 el valor de la variable `secTimeout` (definida en el entorno del programa Java) solo es conocido en tiempo de ejecución, luego solo una vez que la componga "`createEPL`" se sabrán los valores que forman parte de la consulta. Es en este momento cuando la clase "`EPAdministratorImpl.java`", además de lanzar la consulta para su ejecución, la almacena en el fichero `''queries.epl''`.

```
String textTwo = "select Part.zone from pattern [" +
                 "every Part=CountZone(cnt in (1, 2)) ->" +
                 "(timer:interval(" + secTimeout + " sec) " +
                 "and not CountZone(zone=Part.zone, " +
                 "cnt in (0,3)))]";

EPLStatement stmtTwo =
            epService.getEPAdministrator().createEPL(textTwo);
```

Código 8.1: Ejemplo de variable que obtendrá su valor en tiempo de ejecución.

Tras ejecutar este componente, se crea un fichero que contiene un listado con las consultas que se crean y lanzan en un programa. Este fichero con las consultas lo toma por el *analizador* como entrada, de ahí que los programas con las consultas almacenadas en ficheros independientes no tengan que dar este paso.

Como ejemplo consideraremos el programa *Terminalsvc-jse* [160], que simula una terminal auto-servicio que maneja el sistema de un aeropuerto, el cual obtiene una gran cantidad de eventos de todas las terminales que están conectadas. Algunos de los eventos que se reciben son situaciones anormales como "paper low" (poca cantidad de papel), "terminal out of order" (terminal fuera de servicio)... Otros eventos contemplan actividades de los clientes: a la hora de hacer "check in", imprimir las tarjetas de embarque, etc. La figura 8.2 muestra la salida del componente *capturador*, es decir las consultas EPL de EsperTech que se componen y lanzan a través de la función "`createEPL`" desde el programa Java *Terminalsvc-jse*.

```
select a.terminal.id as terminal from pattern
        [ every a=Checkin -> ( OutOfOrder(terminal.id=a.terminal.id) and not
        (Cancelled(terminal.id=a.terminal.id) or Completed(terminal.id=a.terminal.id)))]
select * from BaseTerminalEvent where type = 'LowPaper' or type = 'OutOfOrder'
select '1' as terminal, 'terminal is offline' as text from pattern
        [ every timer:interval(60 seconds) -> (timer:interval(65 seconds) and not
        Status(terminal.id = 'T1')) ] output first every 5 minutes
insert into CountPerType select type, count(*) as countPerType from
        BaseTerminalEvent.win:time(10 min) group by type output all every 10 seconds
insert into VirtualLatency select (b.timestamp - a.timestamp) as latency from pattern
        [ every a=Checkin -> b=BaseTerminalEvent(terminal.id=a.terminal.id,
        type in ('Completed', 'Cancelled', 'OutOfOrder'))]
select * from VirtualLatency.win:length_batch(1000).stat:uni(latency)
```

Figura 8.2: Consultas capturadas del programa Terminalsvc-jse.

8.2.2. Analizador

Antes de generar los mutantes, es necesario identificar las diferentes localizaciones de las consultas EPL de EsperTech que se lanzan en el programa original que pueden ser mutadas. Este es el papel del *analizador*, que toma las consultas y lista, para cada operador, su nombre, el número de localizaciones en las que puede aplicarse y el número de mutantes diferentes que se pueden producir para cada localización. Seguimos tomando como ejemplo el programa *Terminalsvc-jse*, en la figura 8.3 se muestran las consultas del programa y en ellas se indican algunas de las localizaciones donde pueden aplicarse los operadores de mutación definidos para EPL de EsperTech.

```
                                                                    Operador PNR
select a.terminal.id as terminal from pattern
        [ every a=Checkin -> ( OutOfOrder(terminal.id=a.terminal.id) and not
        (Cancelled(terminal.id=a.terminal.id) or Completed(terminal.id=a.terminal.id)))]

                                                                    Operador RLO
select * from BaseTerminalEvent where type = 'LowPaper' or  type = 'OutOfOrder'

                                                                    Operador POM
select '1' as terminal, 'terminal is offline' as text from pattern
        [ every timer:interval(60 seconds) -> (timer:interval(65 seconds) and not
        Status(terminal.id = 'T1')) ] output first every 5 minutes       Operador RTU

                                                              Operadores WTM y WBT
insert into CountPerType select type, count(*) as countPerType from
        BaseTerminalEvent.win:time(10 min) group by type output all every 10 seconds

                                                                    Operador RAO
insert into VirtualLatency select (b.timestamp - a.timestamp) as latency from pattern
        [ every a=Checkin -> b=BaseTerminalEvent(terminal.id=a.terminal.id,
        type in ('Completed', 'Cancelled', 'OutOfOrder'))]

                                                              Operadores WLM y WBL
select * from VirtualLatency.win:length_batch(1000).stat:uni(latency)
```

FIGURA 8.3: Consultas de Terminalsvc-jse con algunos operadores localizados.

La figura 8.4 muestra la salida del analizador para estas consultas. Esta salida indica que se pueden aplicar 18 operadores de mutación de los 34 definidos. También muestra que el operador **RRO** puede aplicarse en siete localizaciones diferentes, **RSC** se puede aplicar en seis, **RTU** se puede aplicar en cinco, **RLO** se puede aplicar en cuatro, **PRE**, **POC** y **PFP** en tres, **PNR** y **POM** en dos localizaciones y, por último, **RAO**, **RNO**, **RGR**, **WLM**, **WTM**, **WBL**, **WBT**, **IWR** y **ICN** en una localidad cada uno. **RSC** y **RRO** producen cinco mutantes diferentes por cada localización, **RTU** y **RAO** producen cuatro mutantes diferentes por cada localización, **POM**, **RNO**, **WLM** y **WTM** dos mutantes por cada localización, mientras que los demás operadores solo producen un mutante por cada una de ellas.

```
PNR 2 1
POM 2 2
PRE 3 1
POC 3 1
PFP 3 1
WLM 1 2
WTM 1 2
WBL 1 1
WBT 1 1
RLO 4 1
RTU 5 4
RAO 1 4
RRO 7 5
RNO 1 2
RGR 1 2
RSC 6 5
IWR 1 1
ICN 1 1
```

FIGURA 8.4: Salida del analizador para Terminalsvc-jse.

8.2.3. Generador de mutantes

El generador de mutantes recibe como entrada el programa original Java, las consultas capturadas y la salida del analizador, produciendo como salida el conjunto de mutantes generados.

Los mutantes se codifican utilizando tres campos (figura 8.5). El campo *Operador* contiene el identificador del operador de mutación a utilizar, el campo *Localización* indica dónde debería ser aplicado y, por último, el campo *Atributo* selecciona el reemplazo exacto a llevar a cabo por el operador de mutación, en el caso de que haya varias opciones.

Operador	Localización	Atributo

FIGURA 8.5: Codificación de un mutante.

La tabla 8.1 muestra para cada operador los posibles valores del campo *Operador* (valor que coincide con el nombre del operador) y el valor máximo aceptado por el campo *Atributo*. En el caso del operador $\mathbf{RSR}_2$, operador que acepta el mayor valor de atributo, hay que tener en cuenta que la subconsulta que sufre la mutación no solo cambia su palabra reservada por la otra de su tipo (`in, not in`), sino que también, se reemplaza con las del tipo III (`exist, not exist`) y con las del tipo I (`all, any, some`). Para cada una de las palabras reservadas del tipo I, hay que añadir el operador relacional (un total de 6 posibles operadores relacionales). Luego si existen 6 posibles operadores relacionales que hay que añadir para cada palabra reservada de tipo I son 18 mutantes, más 2 cambios de las palabras reservadas de tipo III y 1 cambio de la palabra reservada del tipo II que no está en la consulta original, haciendo un total de 21 posibles cambios.

Operador & Valor		Valor max. de atributo
PFP	1	
PGR	1	
PNR	1	
POC	1	
POM	2	{+1, -1}
PRE	1	
RAF	15	[distinct] {max, min, avg, sum, count, median, stddev, avedev}
RAO	4	{+, -, /, *, %}
RBR	3	{a >x AND a <= y, a >= x AND a <y, not a between x AND y}
RGR	2	{max, min}
RJR	4	inner, left outer, right outer, full outer, outer
RLM	1	
RLA	4	%ExprLike, _ExprLike, ExprLike_, ExprLike%
RAW	1	
RBW	1	
RLO	1	
RNO	2	{+1, -1}
RNW	1	
ROM	1	
ROS	1	
RRO	5	{>, <, <>, =, >=, <=}
RRR_1	1	
RRR_2	2	prev, prevtail, prior
RSC	5	select [distinct] rstream, irstream
RSR_1	6	{e$\Re$[all, any, some](Q), e[in, not in](Q), [exist, not exist](Q)}
RSR_2	21	{e[in, not in](Q), e$\Re$[all, any, some](Q), [exist, not exist](Q)} Donde $\Re$ puede adquirir los valores [>, <, <>, =, >=, <=]
RSR_3	1	
RTU	4	{milliseconds, seconds, minutes, hours, days}
WLM	2	{+1, -1}
WTM	2	{+1, -1}
WBL	1	
WBT	1	
ICN	1	
IWR	1	

Tabla 8.1: Valores y atributos para los operadores de mutación.

Como se puede observar se ha puesto especial cuidado en la implementación del generador para que no produzca mutantes que sean idénticos al programa original. Por ejemplo, es imposible, por el diseño, reemplazar el operador relacional > por sí mismo cuando se aplica el operador **RRO**. También se ha evitado, en el operador **RAF**, que se incluya la palabra reservada `all` con la cual, según la sintaxis de EPL de EsperTech, se obtiene el mismo resultado que si no se pone, lo que implicaría un mutante equivalente.

La figura 8.6 muestra el mutante (**WBL**, 1, 1) del programa *Terminalsvc-jse*, resultado de aplicar el operador **WBL**, en la ventana de longitud por lotes de ”`VirtualLatency`“, la cual se ha transformado en una ventana de longitud ordinaria.

```
select a.terminal.id as terminal from pattern
        [ every a=Checkin -> ( OutOfOrder(terminal.id=a.terminal.id) and not
        (Cancelled(terminal.id=a.terminal.id) or Completed(terminal.id=a.terminal.id)))]


select * from BaseTerminalEvent where type = 'LowPaper' or  type = 'OutOfOrder'


select '1' as terminal, 'terminal is offline' as text from pattern
        [ every timer:interval(60 seconds) -> (timer:interval(65 seconds) and not
        Status(terminal.id = 'T1')) ] output first every 5 minutes


insert into CountPerType select type, count(*) as countPerType from
        BaseTerminalEvent.win:time(10 min) group by type output all every 10 seconds


insert into VirtualLatency select (b.timestamp - a.timestamp) as latency from pattern
        [ every a=Checkin -> b=BaseTerminalEvent(terminal.id=a.terminal.id,
        type in ('Completed', 'Cancelled', 'OutOfOrder'))]


select * from VirtualLatency.win:length(1000).stat:uni(latency)
```

Figura 8.6: Mutante (WBL, 1, 1) de Terminalsvc-jse.

El mutante que se genera es una copia exacta de todo el programa (directorios y subdirectorios que contenga), pero incluyendo un fichero adicional que contiene todas las consultas EPL de EsperTech (una de ellas tiene el correspondiente operador de mutación aplicado). Este fichero, si ha sido ejecutado el componente *capturador*, tendrá como fichero original el fichero ”queries.epl” que genera este componente. En caso contrario, será el fichero, con todas las consultas EPL de EsperTech, con el que trabaja originalmente el programa que está bajo análisis. Este fichero se ubica dentro del directorio de ejecución de cada uno de los mutantes. A la hora de la ejecución, las consultas que ejecutará el programa mutado serán las que se incluyen en este fichero. Esto se consigue gracias a otra modificación aplicada a la clase de Esper ”`EPAdministratorImpl.java`“, que busca que el fichero generado para cada mutante se encuentre en el directorio de ejecución del mismo, así como lanzar las consultas de dicho fichero en el orden que sigue el programa.

Implementación de los operadores

Para la implementación de los operadores de mutación se ha utilizado Java ya que este lenguaje de programación es el empleado por otras herramientas desarrolladas por el grupo de investigación UCASE Software Engineering Research Group, es el lenguaje de programación que se emplea para lanzar las consultas EPL de EsperTech y además permite independencia de la plataforma.

El generador de mutantes recibe los tres campos que codifican a cada mutante generado y el mutante especificado. El campo *Operador* se utiliza para elegir la clase que implementa la transformación requerida (existe una clase por cada operador definido, es decir, 34). Estas clases no necesitan más el campo *Operador* y solo reciben la *Localización* y el *Atributo*.

Las clases utilizan el campo *Localización* para obtener una referencia al nodo a transformar. Las consultas EPL de EsperTech que manejan las clases se mutan siguiendo su árbol sintáctico, así que con la *Localización* se controla que el operador a transformar sea el seleccionado. La transformación exacta que se realiza depende del campo *Atributo*.

Para comprender mejor el funcionamiento de la generación de mutantes, se propone el siguiente ejemplo en el que el operador relacional de una consulta EPL de EsperTech va a ser mutado, es decir, se va a aplicar el operador RRO definido en tabla 7.3.

```
SELECT * FROM cep.ssh.connection WHERE duration < 5
```

El generador de mutantes MuEPL toma como entrada la consulta original y los parámetros correspondientes (*Operador*, *Localización* y *Atributo*) que definen el mutante. Tras esto transforma la consulta original a su árbol sintáctico, algo que se consigue gracias a la clase `ParseHelper` desarrollada por EsperTech [9] que genera el árbol sintáctico de la consulta EPL de EsperTech. Como resultado, los elementos del árbol sintáctico que conforman esta consulta EPL de EsperTech son:

```
(EPL_EXPR (SELECTION_EXPR *)
(STREAM_EXPR (EVENT_FILTER_EXPR cep.ssh.connection))
(WHERE_EXPR (< (EVENT_PROP_EXPR (EVENT_PROP_SIMPLE duration)) 5)))
```

Aplicamos el mutante (RRO, 1, 3). El recorrido que se hace en el árbol sintáctico original para hacer la mutación pasa por los elementos: `EPL_EXPR`, su tercer hijo `WHERE_EXPR` y, finalmente, el primer hijo del elemento `WHERE_EXPR`, el operador relacional <. Una vez localizado el elemento que se quiere mutar (gracias al parámetro *Localización*, en este

ejemplo con valor 1), se lee el valor del parámetro *Atributo*. Según el operador, este parámetro puede tener diferentes valores. En el caso del operador RRO su valor puede estar comprendido entre [1,5]. Según el número introducido, el operador se cambiará por cualquiera de los operadores relacionales `{=, <>, >, <=, >=}`. En el ejemplo el valor del atributo es 3, luego se cambia por el operador relacional `>`. Tras la mutación, el árbol sintáctico de la consulta EPL de EsperTech mutada es:

```
(EPL_EXPR (SELECTION_EXPR *)
(STREAM_EXPR (EVENT_FILTER_EXPR cep.ssh.connection))
(WHERE_EXPR (> (EVENT_PROP_EXPR (EVENT_PROP_SIMPLE duration)) 5)))
```

Tras realizar la mutación en el elemento o elementos afectados, MuEPL ha de transformar el árbol sintáctico a una consulta EPL de EsperTech ”legible”, para ello se ha desarrollado una clase, `EPLTree`, que hace el proceso inverso de la clase `ParseHelper`. Se va recorriendo todos los elementos del árbol y se va construyendo la consulta. Un mecanismo no tan intuitivo si se considera que no todos los elementos tienen el mismo orden a la hora de construir la consulta EPL de EsperTech (de ahí que haya sido fundamental el estudio previo de la sintaxis de EPL de EsperTech recogido en el capítulo 2). Los elementos que conforman en este lenguaje de programación vienen recogidos en [161], los cuales se podrían agrupar en:

- Elementos de la cláusula SELECT
- Elementos de la cláusula FROM
- Elementos de la cláusula WHERE
- Elementos de la cláusula GROUP BY
- Elementos de la cláusula HAVING
- Elementos de la cláusula ORDER BY
- Elementos de las palabras reservadas del propio lenguaje
- Elementos de los operadores del lenguaje (aritméticos, lógicos, relacionales...)
- Elementos de patrones
    - Elementos de los operadores de patrones
    - Elementos de los controladores de patrones
    - Elementos de los observadores de patrones

8.2.4. Sistema de ejecución

El sistema de ejecución recibe como entrada el programa original Java que lanza las consultas EPL de EsperTech y los mutantes generados frente al conjunto de casos de prueba. Su salida son los resultados de ejecutar el programa original y los mutantes con los distintos casos de prueba (todos ejecutan los mismos casos de prueba, y en el mismo orden). La salida de cada programa, que se compara con el original para determinar si el mutante muere o no, es el conjunto de eventos procesados durante la ejecución; es decir, los casos de prueba que se ejecutan son un conjunto de eventos que recibe el programa, y la salida es el conjunto de eventos que han sido procesados por la aplicación.

Dada la naturaleza del lenguaje de programación que se está estudiando y los ”datos” que se gestionan (eventos en tiempo-real), hay que asegurar que todos los programas reciban los mismos eventos. Para aplicar la prueba de mutaciones se necesita que el programa original y los mutantes generados estén bajo las mismas condiciones para poder estudiar el comportamiento; esto implica que sean ejecutados en la misma máquina (o máquinas con las mismas características), y reciban los mismos eventos. ¿Cómo puede controlarse que el programa original y los mutantes estén bajo las mismas condiciones de ejecución?

Las consultas EPL de EsperTech de los programas solo se ejecutan cuando ocurren los eventos que las ”activan”. Para probar el funcionamiento correcto de estos programas, algunos incluyen generadores de eventos o hacen pruebas con eventos almacenados en ficheros, servidores, etc. para forzar la ejecución de las consultas. Los eventos que se obtienen a través de los propios generadores de eventos del programa se reciben de forma aleatoria, dando lugar a programas pseudo-deterministas. Para evitar la posible aletoriedad existente se plantea que en el *sistema de ejecución* los programas a ejecutar se sincronicen y se ejecuten de forma paralela, de tal forma que todos reciban a la vez los mismos eventos. Esta forma de ejecución también ayudará a paliar uno de los principales problemas de la prueba de mutaciones: el gran coste computacional que se produce al ejecutar todos los mutantes generados frente al conjunto de casos de prueba. Para conseguirlo se utiliza la clase de Java `CyclicBarrier`, la cual sincroniza los hilos de ejecución (programa original y mutantes). Los hilos llaman al método *await()* de la clase, creándose una barrera donde todos los hilos se detienen hasta que haya tantos hilos a la espera como se le haya indicado. Una vez alcanzada la “barrera” por todos entonces continúa la ejecución de todos los hilos. La figura 8.7 ilustra el proceso. Los hilos se esperan mutuamente, una vez que los N hilos están esperando en la barrera, se liberan y todos pueden continuar su ejecución.

Tras varias ejecuciones y pruebas, se determina que este mecanismo no resuelve el problema pseudo-determinista de los programas estudiados que incluyen generadores de

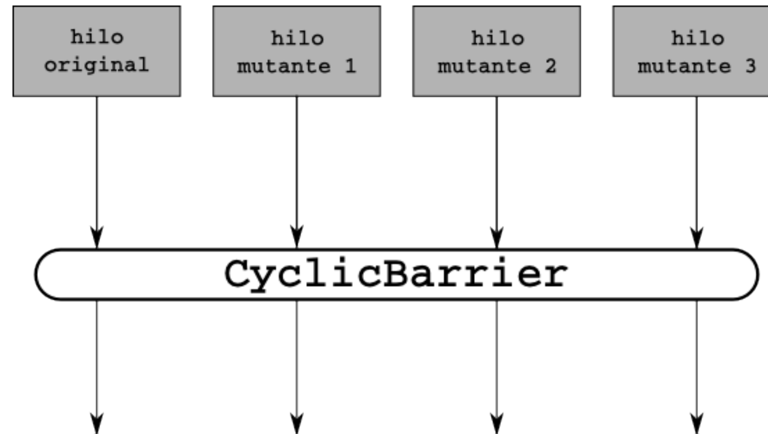

FIGURA 8.7: Mecanismo de ejecución en paralelo.

eventos. El programa original y los mutantes no reciben los mismos eventos, existen más parámetros que los determinan: tiempo del sistema, tipo de máquina... Esta no es la solución, así que se opta por convertir estos programas que incluyen generadores de eventos, en programas deterministas. Pero, ¿cómo?

La solución que se adopta es la de implementar, en el propio programa a estudiar, una funcionalidad que vuelque los eventos generados por el programa a un fichero. El formato de este fichero será uno de los más empleados según se estudió en la sección 6.1 (en los programas que incluyen un generador de eventos y que han sido utilizados en las pruebas de la presente tesis, el formato elegido ha sido el `csv`). Por otro lado, todos aquellos programas que quieran utilizar la herramienta MuEPL y que obtengan los eventos desde su propio generador, o desde un servidor, tendrán que cambiar en su implementación la forma de obtener los eventos. Se tendrá que implementar una funcionalidad para leer los eventos según el formato del fichero donde se almacenen. De este modo, los eventos que procesen los programas serán los mismos y todos estarán bajo las mismas condiciones. La funcionalidad para leer los eventos desde el fichero, servirá para aplicar la prueba de mutaciones en cada programa utilizando no solo los eventos propios de la aplicación, sino también los eventos generados por el método propuesto en la presente tesis (véase capítulo 5).

A pesar de que la ejecución en paralelo no solventa el problema pseudo-determinista, esta se mantiene ya que sí ayuda con el coste computacional de la prueba de mutaciones, véase la sección 9.5.

8.3. Criterios para matar mutantes

Una vez definidos los operadores de mutación para EPL de EsperTech y la arquitectura de MuEPL, se procede a establecer los criterios que determinan cuándo un mutante está muerto: *killing criteria*. Esto es definir los criterios para considerar la salida del mutante diferente a la del original. En [158] se recoge una primera propuesta de *killing criteria* en la que, según el operador de mutación, se establece un criterio u otro. En el *killing criteria* se determina que las salidas son diferentes cuando:

1. El número de eventos obtenidos entre el original y el mutante difiere: se refiere a que la cantidad de eventos que se obtienen en la salida por parte del programa mutado va a ser diferente a la cantidad de eventos del programa original.

2. La latencia de tiempo entre el original y el mutante: hace referencia a que, a pesar de obtener el mismo número de eventos, el tiempo que tarda el programa mutante es diferente (pudiendo ser más o menos tiempo) que el programa original.

3. El contenido de la salida difiere entre el original y el mutante: se establece que a pesar de obtener el mismo número de mutantes en el mismo periodo de tiempo, el contenido de estos eventos obtenidos como salida del programa mutado, es diferente al obtenido por el programa original.

Tras aplicar MuEPL con algunos de los programas estudiados en esta tesis (véase apéndice B), se observa que el *killing criteria* tiene que definirse el mismo para todos los operadores de mutación, ya que lo que varía es la cantidad de eventos procesados, sea cual sea la mutación o cambio que se aplique a la consulta. Los programas con consultas EPL de EsperTech reciben un conjunto de eventos de los cuales algunos son los que "activan" las consultas, es decir, son los eventos que son procesados por las consultas. Esto lleva a definir como salida el conjunto de eventos procesados por las consultas. Teniendo esto en cuenta se implementa en MuEPL una funcionalidad que permite realizar comparativas de las salidas del programa original y de los mutantes con los siguientes criterios (llegándose a implementar la mezcla de algunos):

1. Número de eventos: Esto se consigue contabilizando y comparando el número de eventos de las salidas del original y los mutantes. En cuanto el número difiera en una unidad, el mutante se considera muerto.

2. Tiempo de ejecución: Dado que las modificaciones que aplican los operadores de mutación van a afectar en el procesado de más o menos eventos, lo cual afecta a la latencia de tiempo, se controla el tiempo de ejecución de cada uno de los programas

mutantes y el programa original. Si el tiempo de ejecución difiere, el mutante se considera muerto.

3. Contenido de los eventos: La información que almacenan los eventos (del mismo tipo) sigue una misma estructura. Si en la salida del programa se obtiene siempre el mismo número de eventos y en el mismo orden, se tiene que analizar el contenido de los eventos para encontrar las posibles diferencias. Para esto se indica una palabra delimitadora (común en todos los eventos) y a partir de esta se observa el contenido. Si en cualquiera de los eventos hay un valor diferente, el mutante se considera muerto.

8.4. Conclusiones

Se ha presentado MuEPL, una herramienta de generación y ejecución de mutantes para consultas EPL de EsperTech. MuEPL aplica la prueba de mutaciones (véase capítulo 2 e incorpora los 34 operadores de mutación definidos para EPL de EsperTech.

La herramienta MuEPL permite automatizar la aplicación de la prueba de mutaciones a programas que ejecuten consultas EPL de EsperTech ya que genera todos los mutantes de dicho programa mediante la aplicación de los operadores de mutación definidos. Asimismo ejecuta estos mutantes frente al conjunto de casos de prueba y compara el comportamiento de los mutantes con el del programa original.

MuEPL consta de cuatro componentes: el *capturador*, un componente que obtiene las consultas completas tras la ejecución del programa original Java; el *analizador*, que determina qué operadores de mutación son aplicables a las consultas EPL de EsperTech; el *generador de mutantes*, que toma como entrada el programa original y la salida del analizador, lo que permite generar todos los mutantes, y el *sistema de ejecución*, que ejecuta el programa original y los mutantes frente al conjunto de casos de prueba, comparando sus comportamientos y produciendo como resultado la matriz de ejecución que permite clasificar los mutantes en vivos, muertos y no válidos.

En el sistema de ejecución se ha desarrollado un método para la sincronización de los programas, que ayuda a reducir el coste computacional. En el capítulo 9 se mostrará la efectividad del método aplicado utilizando la clase de Java `CyclicBarrier`.

Se ha modificado la biblioteca Esper para que los programas Java que usan EPL de EsperTech puedan ser analizados dinámicamente con MuEPL, permitiendo capturar las consultas que se ejecutan y de este modo generar los mutantes.

Los criterios para considerar muertos los mutantes, *killing criteria*, se han definido igual para todos los operadores de mutación. Se contemplan los siguientes criterios: número de eventos, número del tipo de eventos y contenido de los eventos. En el capítulo 9 se presenta un estudio que determinará qué criterio o mezcla de criterios es más adecuado a aplicar en las aplicaciones. Las ejecuciones han de realizarse con eventos con valores y estructuras determinadas, lo que lleva a desarrollar un generador de eventos (véase capítulo 6) que acepte las especificaciones determinadas por el usuario (véase capítulo 5) para cualquier programa que quiera ser analizado y los procese.

Capítulo 9

Resultados

En este capítulo se recogen los resultados de las diferentes pruebas donde se emplea la propuesta realizada y las herramientas desarrolladas: IoT-TEG y MuEPL.

Para comprobar la rapidez de la propuesta de especificación definimos, utilizando el método propuesto, los eventos generados por IoT-TEG a partir de los eventos reales de la plataforma IoT "ThingSpeak" [109]. Por tanto, los eventos generados a través del método se comparan con los originales de la plataforma.

Otra prueba que se realiza para la validación del método, es la aplicación de la prueba de mutaciones en distintos casos de estudio (véase apéndice B). Para validar los eventos de prueba generados, utilizamos la prueba de mutaciones con los eventos originales de cada uno de los casos de estudio y con los generados por la herramienta IoT-TEG. Los resultados obtenidos tras aplicar la prueba de mutaciones con ambos conjuntos de casos de prueba se comparan y analizan.

Para aplicar la prueba de mutaciones en EPL de EsperTech con MuEPL, se necesita establecer el criterio para considerar muerto un mutante. Una vez definido, se emplea MuEPL para realizar comparativas de ejecución de los casos de estudio usando sus eventos originales y los generados por IoT-TEG (cuando se cumplan las condiciones, también se usarán eventos generados usando la funcionalidad avanzada de la herramienta). Adicionalmente, se justifica la definición de los operadores de mutación propios del lenguaje EPL de EsperTech analizando los resultados.

Finalmente, se hace un estudio sobre la efectividad del `CyclicBarrier` empleado en MuEPL (véase sección 8.2.4).

Todos los experimentos que aparecen en este capítulo se han ejecutado en una máquina Intel(R) Xeon(R) de 64 bits con 260GHz y 16GB RAM, con la distribución Linux Ubuntu 14.04.03 LTS.

9.1. Aplicación de la especificación para definir tipos de eventos con tipos de eventos reales

Para abarcar el mayor número de tipos de eventos posible con la especificación que se propone en la presente tesis, se acudió a ”ThingSpeak” [109], una plataforma de datos abierta y una API para el ”Internet de las Cosas”, que permite a cualquier usuario recoger, almacenar, analizar, visualizar y actuar sobre datos que provienen de sensores o transmisores como Arduino, Raspberry Pi, BeagleBone Black y otro tipo de hardware. Esta plataforma contiene una serie de canales en los cuales existe la posibilidad de compartir dicha información haciéndolos públicos.

Cada día esta plataforma crece, ya que son muchos los usuarios alrededor de todo el mundo que quieren compartir sus datos a través de estos canales. En cada uno de esos canales se define un tipo de evento (la estructura), que determina el tipo de estudio que se está realizando. A día de hoy dispone de más de 5000 canales, algunos de ellos *activos* y otros no. Se dice que un canal *no es activo* cuando no se obtienen eventos del mismo. Los tipos de eventos que están definidos en estos canales son para estudios meteorológicos, térmicos, referentes a la luz, agua, humedad, ruido, fermentación, conexiones de Internet, etc. Para este trabajo se han analizado los tipos de eventos de más de 500 canales, aproximadamente un 10 % de los canales disponibles en el momento del estudio (como se ha comentado el incremento de canales de la plataforma es considerable). De los 508 canales analizados, 450 eran válidos y el resto era no activos. El hecho de no ofrecer eventos imposibilita observar los datos que componen el tipo de evento, motivo por el que fueron descartados. El resto de tipos de eventos, 450, pueden ser definidos por la especificación que en esta Tesis se propone. El listado de los canales analizados de la plataforma ”ThingSpeak” se encuentra localizado en el repositorio de UCASE Software Engineering Research Group[1].

En [156] se hace una comparativa de los diferentes eventos ofrecidos por 10 canales de la plataforma “ThingSpeak”, y los obtenidos utilizando el método propuesto en la presente tesis. En dicho trabajo no solo se muestran los eventos obtenidos utilizando este método, sino que también se presentan las definiciones de los tipos de eventos utilizando la especificación propuesta en el capítulo 5.

Además de analizar los tipos de evento de la plataforma ”ThingSpeak”, se definieron los tipos de evento de los casos de estudio utilizados en esta tesis (véase apéndice B) y en el apéndice C aparecen las definiciones de todos ellos.

[1] `https://neptuno.uca.es/redmine/projects/sources-fm/repository/show/trunk/src/IoT-TEG/test`

9.2. Selección del criterio para matar mutantes

En el capítulo 8 se presentó el generador de mutantes MuEPL, para el cual se tiene que establecer un criterio para determinar cuándo un mutante está muerto (*killing criteria*). Para establecerlo se van a comparar, las salidas de los programas mutantes y el programa original, en concreto, el número de eventos, el contenido de los eventos, el orden de los eventos y los tiempos de ejecución. Estas salidas permitirán establecer el *killing criteria* para los operadores de mutación de EPL de EsperTech.

Los criterios que se han implementado en MuEPL son los siguientes:

1. Tiempo de ejecución: Dado que las modificaciones que aplican los operadores de mutación van a afectar en el procesado de más o menos eventos, lo cual afecta a la latencia de tiempo, se controla el tiempo de ejecución de cada uno de los programas mutantes y el programa original. Si el tiempo de ejecución difiere, el mutante se considera muerto.

2. Contenido y orden de los eventos: La información que almacenan los eventos (del mismo tipo) sigue una misma estructura. Si en la salida del programa se obtiene siempre el mismo número de eventos y en el mismo orden, se tiene que analizar el contenido de los eventos para encontrar las posibles diferencias. Para esto se indica una palabra delimitadora (común en todos los eventos) y a partir de esta se observa el contenido. Si en cualquiera de los eventos hay un valor diferente, el mutante se considera muerto.

3. Contenido y número de eventos: Esto se consigue contabilizando y comparando el número de eventos de las salidas del original y los mutantes. En cuanto el número difiera en una unidad, el mutante se considera muerto. Si el número de eventos es el mismo, se comprueba si existe algún evento en la salida mutante que no aparezca en la salida del original, en caso de existir un evento con contenido diferente, el mutante se considera muerto.

Para establecer el *killing criteria* se analizarán las salidas según el criterio considerado: tiempo de ejecución, contenido y orden de los eventos y contenido y número de eventos. El caso de estudio utilizado para seleccionar el criterio para matar mutantes es ”Ecological Island” (véase sección B.3).

MUTANTE	EJEC. 1	EJEC. 2
PRE_1_1	132798×10^6	150234×10^6
PRE_2_1	145561×10^6	164108×10^6
PRE_3_1	148289×10^6	173542×10^6
PRE_4_1	149061×10^6	175753×10^6
RLO_1_1	261457×10^6	297612×10^6
RLO_2_1	274098×10^6	306308×10^6
RLO_3_1	264084×10^6	305661×10^6
RLO_4_1	271133×10^6	309855×10^6
...	...	...
RRO_9_3	272380×10^6	288722×10^6
RRO_9_4	272532×10^6	301018×10^6
RRO_9_5	262821×10^6	301022×10^6
RSC_1_1	261089×10^6	299392×10^6
...	...	...
WTM_1_1	358800×10^6	294585×10^6
WTM_1_2	233408×10^6	150963×10^6
WTM_2_1	359644×10^6	305415×10^6
WTM_2_2	247237×10^6	168906×10^6
WTM_3_1	358780×10^6	298505×10^6
...	...	...

TABLA 9.1: Comparativas de los tiempos de ejecución.

9.2.1. Tiempo de ejecución

El tiempo de ejecución de un programa puede variar por muchos factores: tipo de máquina, sistema operativo, ejecución de otros programas, datos de entrada, etc. Hoy en día el tiempo de ejecución de un programa puede medirse con unidades de tiempo que ofrecen una alta precisión, esto hace que la más mínima variación modifique el valor de este parámetro.

El primer paso es establecer la unidad de tiempo: nanosegundos. El segundo paso es comprobar que con una aplicación ejecutada varias veces en una misma máquina con unos datos de entrada (siempre los mismos eventos) se obtengan los mismos tiempos de ejecución. Si no fuera así, se debe descartar el criterio.

Para esta parte del análisis se escogieron, para el caso de estudio "Ecological Island", los eventos lanzados por el canal 16302 de la plataforma "ThinkSpeak" [109]. Se almacenaron 1000 eventos en un fichero con formato `JSON` que sirvieron como caso de prueba.

Se realizaron dos ejecuciones del caso de estudio bajo las mismas condiciones y ejecutando siempre los mismos 1000 eventos almacenados. En la tabla 9.1 se muestran los resultados obtenidos; en la primera columna, algunos de los mutantes generados por MuEPL y, en las sucesivas columnas, cada uno de los tiempos de ejecución (en nanosegundos) que han obtenido cada uno de ellos en las dos ejecuciones. Estos resultados muestran valores de tiempo dispares para ejecuciones realizadas en una misma máquina y con los mismos eventos de entrada.

Llama la atención la gran diferencia de tiempo entre las ejecuciones de algunos de los mutantes; en alguno de ellos se llega a una diferencia de más de 70 segundos, más de 1 minuto de diferencia a la hora de ejecutar el mismo programa bajo las mismas condiciones y con los mismos datos de entrada. Si se hubieran obtenido unos tiempos relativamente parecidos, se podría haber establecido un rango de tiempo para determinar qué mutantes se consideran muertos y cuales no. Así que se descarta definitivamente el criterio del tiempo de ejecución para considerar muerto o no un mutante, ya que con un mismo programa se obtienen diferentes tiempos de ejecución.

9.2.2. Contenido y orden de los eventos

Para esta segunda parte del experimento se observan las salidas propias del programa. En este caso las salidas del programa que se analizan son los eventos capturados y procesados. El proceso a seguir es comprobar que la misma aplicación ejecutada al menos dos veces en una misma máquina y con los mismos datos de entrada (siempre los mismos eventos), se obtengan los mismos eventos en el mismo orden.

Se sigue con el caso de estudio ”Ecological Island”, con los mismos 1000 eventos almacenados del canal 16302 de la plataforma ”ThinkSpeak”, y se ejecuta dos veces bajo las mismas condiciones.

El tipo de evento que se procesa en este caso de estudio tiene los siguientes atributos:

- `timestamp` indica la fecha y hora que se produce el evento.
- `islandId` el número que identifica la isla de la que se están recopilando datos.
- `filledPercentage` indica el porcentaje de llenado de la isla. Es opcional, es decir, puede aparecer o no en el evento.
- `blocked` atributo booleano que nos indica si la isla está bloqueada (valor numérico 1) o no (valor numérico 0).
- `alertLevel` indica el nivel de alerta en el que se encuentra la isla.
- `islandName` el nombre de la isla.
- `temperature` temperatura en grados Fahrenheit que hay en la isla.

Un ejemplo de eventos del canal 16302 se encuentra en el código 9.1, donde se muestran los valores de cada uno de sus atributos que determinarán si los eventos son capturados y procesados por las consultas de la aplicación.

```
{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}
...
```

Código 9.1: Eventos del canal 16302 de la plataforma ThinkSpeak.

Tal y como se comentó en el la sección 8.3 los eventos del mismo tipo siguen una misma estructura así que, para facilitar la comparativa de los resultados obtenidos tras las dos ejecuciones, se recorta del código la parte común de los eventos: `timestamp` e `islandId`. Los resultados obtenidos de la primera y segunda ejecución aparecen en los códigos 9.2 y 9.3 respectivamente.

```
... 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}
...
```

Código 9.2: Killing criteria - Contenido y orden de los eventos: Ejecución 1.

```
... 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}
...
```

Código 9.3: Killing criteria - Contenido y orden de los eventos: Ejecución 2.

Se puede observar que con las diez primeras líneas obtenidas en la salida (los 10 primeros eventos procesados) el orden varía: el primer evento de la primera ejecución aparece segundo en la segunda ejecución, el segundo de la primera ejecución aparece tercero en la segunda ejecución, el tercer evento de la primera ejecución aparece en primera posición en la segunda. Así que, aunque se procesen los mismos eventos, el orden del procesado de los eventos cambia. Luego el criterio para considerar muerto un mutante según el contenido y el orden de los eventos tampoco puede considerarse ya que, para un mismo programa ejecutado bajo las mismas condiciones y con los mismos datos de entrada, el orden de los eventos procesados (la salida) puede ser diferente.

9.2.3. Contenido y número de eventos

El último criterio considera el contenido y el número de eventos procesados. Por ello, tras las ejecuciones anteriores, se observa que el número de eventos que captura y procesa un mismo programa bajo las mismas condiciones y con los mismos eventos de entrada son los mismos (aunque en diferente orden). Lo que lleva a escoger este criterio como el *killing criteria* para este tipo de programas de procesado de eventos.

Además de las dos ejecuciones anteriores del caso de estudio (”Ecological Island”), se realizan ocho ejecuciones más bajo las mismas condiciones y con los mismos datos de entrada, un total de diez ejecuciones. Utilizando como *killing criteria* el contenido y el número de eventos procesados, se obtiene que para los 181 mutantes generados en este caso de estudio.

- 96 mutantes están muertos - el número de eventos procesados es diferente, al menos en una unidad, al número de eventos procesados por el original, o existe al menos un evento cuyo contenido no está en el original.
- 77 mutantes están vivos - el número de eventos procesados es igual que los procesados por el original, así como el contenido de estos.
- 8 mutantes son erróneos - la mutación realizada no sigue los estándares de EPL de EsperTech.

Estos datos se obtienen para las diez ejecuciones. Luego, como se obtienen el mismo número de mutantes muertos, vivos y erróneos en todas, se selecciona como *killing criteria* el contenido y el número de eventos procesados.

9.3. Comparativas de ejecución

Una vez establecido como *killing criteria* el contenido y el número de eventos procesados, se pasa a demostrar la utilidad del generador de eventos IoT-TEG presentado en el capítulo 6. En esta sección se harán comparativas de las salidas (de acuerdo al *killing criteria*) de los diferentes casos de estudio explicados en detalle en el apéndice B:

1. Se realizarán ejecuciones utilizando los eventos originales de las aplicaciones. Se llaman eventos originales aquellos con los que la aplicación originalmente se ejecuta, bien sean eventos de un servidor o plataforma pública, eventos generados por un generador de eventos propio de la aplicación, eventos almacenados en ficheros para hacer pruebas...
2. Se realizarán ejecuciones utilizando los eventos generados por IoT-TEG para cada uno de los casos de estudio.
3. Se realizarán ejecuciones utilizando los eventos generados por IoT-TEG para cada uno de los casos de estudio, pero utilizando (en aquellos casos de estudio que tengan cláusulas `where` con operaciones relacionales y lógicas involucradas) la funcionalidad avanzada de IoT-TEG.

9.3.1. Caso de estudio - Self-Service Terminal

Dentro de las distribuciones de la compañía EsperTech [9], se ofrecen diversos casos de estudio para comprobar la potencia y versatibilidad de EPL. Uno de los programas de ejemplos al que se le ha aplicado MuEPL es el denominado "Self-Service Terminal" [160]. El ejemplo está basado en J2EE y simula un sistema de auto servicio de un aeropuerto, el cual obtiene una gran cantidad de eventos de las terminales que están conectadas al mismo. Algunos eventos indican situaciones anormales como "paper low" (poco papel), "terminal out of order" (terminal fuera de servicio), etc. Otros eventos contemplan actividades de los clientes: a la hora de hacer "check in", imprimir las tarjetas de embarque, etc. En este caso de estudio hay involucradas 6 consultas EPL de EsperTech. Para obtener más información sobre este caso de estudio, véase el apéndice B.1.

Para hacer las pruebas con la metodología propuesta en la presente tesis, se define el tipo de evento "TerminalEvent". Esta definición (véase la sección C.1) se emplea para generar eventos a través de IoT-TEG. Adicionalmente se modifica la aplicación para que en vez de obtener los eventos del generador propio de la aplicación, estos sean obtenidos de un fichero en formato `CSV` generado por IoT-TEG.

La tabla 9.2 muestra los operadores de mutación que pueden aplicarse tras utilizar la herramienta MuEPL en el caso de estudio.

Operador	Ocurrencia	Atributo
PNR	2	1
POM	2	1
PRE	3	1
POC	3	1
PFP	3	1
WLM	1	2
WTM	1	2
WBL	1	1
WBT	1	1
RLO	4	1
RTU	5	4
RAO	1	4
RRO	7	5
RNO	1	2
RGR	1	2
RSC	6	5
IWR	1	1
ICN	1	1

Tabla 9.2: Resultados tras aplicar el *analizador* con el caso de estudio Terminal.

Los 18 operadores de mutación que se pueden aplicar son de las cuatro categorías: expresiones de patrones, reemplazamiento, ventanas e inyecciones de ataque de SQL, siendo un total de 120 mutantes los que genera el *generador de mutantes* MuEPL.

Para el *sistema de ejecución* se necesitan los casos de prueba: el conjunto de eventos originales del caso de estudio y los generados por IoT-TEG.

El caso de estudio "Self-Service Terminal" posee un generador de eventos, al cual se le han de introducir los siguientes parámetros: número de iteraciones y segundos de espera. El número de iteraciones indica al generador de eventos cuántos conjuntos de eventos o lotes de eventos se han de generar y, los segundos de espera hacen referencia al tiempo que tiene que esperar entre la creación de lotes de eventos. La cantidad de eventos que pertenece a cada lote es aleatoria, no es un parámetro que se pueda controlar.

Para obtener los eventos originales, se ha volcado la salida del generador de eventos de este caso de estudio a ficheros con formato `CSV`; cada uno de esos ficheros contiene el lote de eventos que forman parte de un caso de prueba. Los casos de prueba se han obtenido introduciendo los siguientes parámetros en la aplicación: 1 iteración y 1000 segundos de espera (parámetros usados 3 veces, se obtienen 3 casos de prueba) y 1 iteración y 10000 segundos de espera (parámetros usados 3 veces, se obtienen 3 casos de prueba), un total de 6 casos de prueba. Como se ha comentado en el párrafo anterior, el número de eventos que pertenece a cada lote es aleatorio. La cantidad de eventos obtenidos, junto con los resultados obtenidos de las 6 ejecuciones, se plasman en la tabla 9.3.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	180	176	170	20693	7184	12328
Iteraciones	1	1	1	1	1	1
Seg. espera	1000	1000	1000	10000	10000	10000
Muertos	19	19	20	113	111	113
Vivos	81	81	82	0	0	0
Erróneos	20	20	18	7	9	7

Tabla 9.3: Resultados del caso de estudio Self-Service Terminal usando sus eventos originales.

Los resultados de la tabla 9.3 muestran que, en este caso de estudio, los segundos de espera influyen; ya que en el número de eventos generados aumenta e incrementa el número de mutantes muertos.

Con el generador de eventos IoT-TEG se han generado 3 ficheros (con formato `CSV`), uno con 200 eventos (número cercano al número de eventos de los tres primeros casos de prueba originales), otro con 500 eventos y otro con 1000 eventos, cuyos valores son totalmente aleatorios. Se quiere observar el comportamiento de la aplicación utilizando esos eventos generados pero con un tiempo de espera diferente. Para ello cada uno de estos ficheros se ha ejecutado con 1000 segundos de espera y con 10000 segundos de espera, un total de 6 casos de prueba. Los resultados de las seis ejecuciones se presentan en la tabla 9.4.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	200	200	500	500	1000	1000
Iteraciones	1	1	1	1	1	1
Seg. espera	1000	10000	1000	10000	1000	10000
Muertos	35	35	112	113	113	113
Vivos	80	80	1	0	0	0
Erróneos	5	5	7	7	7	7

Tabla 9.4: Resultados del caso de estudio Self-Service Terminal usando eventos generados por IoT-TEG.

Los eventos que se han empleado en el primer caso de prueba son los mismos que se emplean en el segundo caso de prueba, los del tercero son los mismos que en el cuarto y los del quinto son los mismos que los del sexto. El parámetro de entrada que cambia en cada caso de prueba es el número de segundos que hay que esperar. Tras observar los resultados de la tabla 9.4, se puede decir que con el generador de eventos IoT-TEG, se ha conseguido matar a todos los mutantes con un menor número de eventos.

9.3.2. Caso de estudio - Transaction

Otro de los casos de estudios que ofrece la compañía EsperTech, y al que también se le ha aplicado MuEPL, es "Transaction 3-Event Challenge" [162]. Este ejemplo consiste

en hacer el seguimiento de los tres componentes de una transacción. En este caso de estudio se le da mucha importancia al uso de al menos tres componentes: identificador del cliente, identificador del proveedor e identificador de la terminal de transacción.

Al igual que el caso de estudio “Self-Service Terminal” (véase sección 9.3.1), esta aplicación también cuenta con un generador de eventos. Que tiene dos parámetros: el tamaño del bloque de eventos y el número de transacciones a realizar. En este caso de estudio hay involucradas 5 consultas EPL de EsperTech. Para obtener más información sobre este caso de estudio, véase el apéndice B.2.

Para hacer las pruebas con la metodología propuesta en la presente tesis, se define el tipo de evento “TransactionEvent”. Esta definición (véase la sección C.2) se emplea para generar eventos a través de IoT-TEG. Adicionalmente se modifica la aplicación para que en vez de obtener los eventos del generador propio de la aplicación, estos sean obtenidos de un fichero en formato `CSV` generado por IoT-TEG.

La tabla 9.5 muestra los operadores de mutación que el analizador de MuEPL establece como aplicables en este caso de estudio.

Operador	Ocurrencia	Atributo
WTM	9	2
WBT	9	1
RLO	1	1
RTU	9	4
RAF	15	15
RAO	3	4
RRO	2	5
RJR	2	4
RGR	2	2
RNW	1	1
RSC	5	5
IWR	2	1
ICN	2	1

Tabla 9.5: Resultados tras aplicar el *analizador* con el caso de estudio Transaction.

Los 13 operadores de mutación que se pueden aplicar son de las categorías: reemplazamiento, ventanas e inyecciones de ataque de SQL, siendo un total de 353 mutantes los que genera el *generador de mutantes* MuEPL.

Al igual que en el caso de estudio anterior, para el *sistema de ejecución* se necesitan los casos de prueba: el conjunto de eventos originales del caso de estudio y los generados por IoT-TEG.

El caso de estudio “Transaction” posee un generador de eventos, al cual se le han de introducir los siguientes parámetros: el tamaño del bloque de eventos y el número de transacciones que se quieren realizar. El tamaño del bloque de eventos viene especificado

con las palabras reservadas {`tiniest` (20 eventos), `tiny` (499 eventos), `small` (4999 eventos), `medium` (14983 eventos), `large` (49999 eventos), `larger` (1999993 eventos), `largerger` (9999991 eventos)}. El número de transacciones, como su nombre indica, especifica el número de transacciones que se quieren efectuar.

Para los eventos originales, se han volcado los eventos (en formato `CSV`) que ha generado el generador de eventos del programa de la siguiente forma: un caso de prueba por cada una de las palabras reservadas que indican el tamaño del bloque de eventos, con 1 como el número de transacciones a realizar y un caso de prueba por cada una de las palabras reservadas que indican el tamaño del bloque de eventos, con 10 como el número de transacciones a realizar; un total de 14 casos de prueba con los eventos originales. Los resultados obtenidos de las 14 ejecuciones, se plasman en las tablas: 9.6 y 9.7.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6	TC.7
Bloque	tiniest	tiny	small	medium	large	larger	largerer
Transacciones	1	1	1	1	1	1	1
Muertos	30	30	30	30	30	30	30
Vivos	176	176	176	176	176	176	176
Erróneos	147	147	147	147	147	147	147

TABLA 9.6: Resultados del caso de estudio Transaction usando sus eventos originales, una transacción.

	TC.8	TC.9	TC.10	TC.11	TC.12	TC.13	TC.14
Bloque	tiniest	tiny	small	medium	large	larger	largerer
Transacciones	10	10	10	10	10	10	10
Muertos	46	46	46	46	46	46	46
Vivos	160	160	160	160	160	160	160
Erróneos	147	147	147	147	147	147	147

TABLA 9.7: Resultados del caso de estudio Transaction usando sus eventos originales, diez transacciones.

Tras observar los resultados de las tablas: 9.6 y 9.7, se puede ver que el número de transacciones influye en el número de mutantes muertos, cuanto mayor sea el número de transacciones, mayor será el número de mutantes muertos.

Llama la atención los resultados obtenidos con los casos de prueba originales. A pesar de ser diferentes los valores de los eventos de cada uno de los casos de prueba, el número de mutantes muertos en todas las ejecuciones donde se usa el mismo número de transacciones, ha sido el mismo. Para comprobar hasta qué punto influye el número de transacciones, y no el parámetro que determina el tamaño de los bloques de eventos (en este caso de estudio), se procede a realizar el experimento con los eventos generados por IoT-TEG con las siguientes características. Se han generado 2 casos de prueba con el generador de eventos IoT-TEG cuyos contenidos recrean el número de eventos que genera la aplicación para 1 y 10 transacciones respectivamente. Para este caso de prueba, utilizaremos en la aplicación el número de transacciones recibidas con cada uno de los

tamaños de bloque de eventos asignado a cada palabra reservada. En total se obtienen 14 casos de prueba. Los resultados obtenidos de las 14 ejecuciones, se plasman en las tablas: 9.8 y 9.9.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6	TC.7
Bloque	tiniest	tiny	small	medium	large	larger	largerer
Transacciones	1	1	1	1	1	1	1
Muertos	33	33	33	33	33	33	33
Vivos	173	173	173	173	173	173	173
Erróneos	147	147	147	147	147	147	147

TABLA 9.8: Resultados del caso de estudio Transaction usando eventos generados por IoT-TEG, una transacción.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6	TC.7
Bloque	tiniest	tiny	small	medium	large	larger	largerer
Transacciones	10	10	10	10	10	10	10
Muertos	49	49	49	49	49	49	49
Vivos	157	157	157	157	157	157	157
Erróneos	147	147	147	147	147	147	147

TABLA 9.9: Resultados del caso de estudio Transaction usando eventos generados por IoT-TEG, diez transacciones.

Tras observar los resultados anteriores, se generan dos casos de estudio con IoT-TEG, en donde se recrea el número de eventos que genera la aplicación para 50 y 100 transacciones. Tras ver el comportamiento de la aplicación, se ejecuta una vez cada uno, obteniéndose un número de mutantes muertos igual que en las pruebas realizadas con diez transacciones. Es decir, que con ambas (50 y 100 transacciones) se obtienen un total de 49 mutantes muertos y 157 vivos. El siguiente paso sería analizar las 157 mutaciones y comprobar si son equivalentes o mutantes difíciles de matar. Independientemente, se observa que el número de mutantes muertos que se obtienen con los casos de prueba generados con la herramienta IoT-TEG es mayor que el número de mutantes muertos que se obtiene con los eventos que originalmente usa la aplicación.

9.3.3. Caso de estudio - Ecological Island

Este caso de estudio [163] ha sido desarrollado con el objetivo de contribuir en el desarrollo de las *Smart Cities*, dotando a las islas ecológicas de cierta “inteligencia” para reducir costes en la empresa recolectora, reducir la contaminación medioambiental y aumentar la eficiencia energética así como velar para el cuidado del patrimonio de la ciudad siendo alertados los servicios de emergencia en su caso y, en general, hacerle al ciudadano la vida más placentera.

Las actividades que se notifican en esta aplicación son: detección de fuego, la entrada del contenedor está bloqueada, el contenedor está medio lleno y el contenedor está prácticamente lleno.

La aplicación está desarrollada para leer en formato `JSON`, ya que como fuente de eventos lee los eventos (en este formato) de la plataforma IoT "ThinkSpeak", en concreto de los canales: 16302, 16306, 16307 y 16308. En este caso de estudio hay involucradas 4 consultas EPL de EsperTech. Para obtener más información sobre este caso de estudio, véase el apéndice B.3.

Para para hacer las pruebas con la metodología propuesta en la presente tesis, se define el tipo de evento "IslandEvent". Esta definición (véase la sección C.3) se emplea para generar eventos, en formato `JSON`, a través de IoT-TEG pudiéndose usar la misma entrada de parámetros del caso de estudio.

La tabla 9.10 muestra qué operadores de mutación pueden aplicarse a las 4 consultas implicadas en este caso de estudio.

Operador	Ocurrencia	Atributo
PRE	4	1
WTM	4	2
WBT	4	1
RLO	8	1
RTU	4	4
RRO	15	5
RNO	23	12
RSC	4	5

Tabla 9.10: Resultados tras aplicar el *analizador* con el caso de estudio Ecological Island.

Los 8 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento y ventanas, siendo un total de 181 mutantes los que genera el *generador de mutantes* MuEPL.

Para el *sistema de ejecución* se necesitan los casos de prueba: el conjunto de eventos originales del caso de estudio y los generados por IoT-TEG.

Para obtener los eventos originales se han almacenado en formato `JSON` los eventos de los canales: 16302, 16303, 16306 y 16307 de la plataforma "ThinkSpeak". De todos los canales se han capturado 1000 y 2000 eventos, teniendo un total de 8 casos de prueba: 1000 y 2000 eventos del canal 16302; 1000 y 2000 eventos del canal 16303, etc. Hay que comentar que tras analizar los valores de estos eventos, se observó que en todos los canales, los 1000 eventos que se habían capturado estaban incluidos entre los 2000. Es decir, que los 1000 eventos del canal 16302 estaban almacenados en los 2000 eventos del

canal 16302, los 1000 eventos del canal 16303 estaban añadidos entre los 2000 eventos del canal 16303, etc. Los resultados de las ocho ejecuciones se presentan en la tabla 9.11.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6	TC.7	TC.8
Canal	16302	16302	16303	16303	16306	16306	16307	16307
Eventos	1000	2000	1000	2000	1000	2000	1000	2000
Muertos	112	110	109	112	110	112	110	112
Vivos	61	63	64	61	63	61	63	61
Erróneos	8	8	8	8	8	8	8	8

Tabla 9.11: Resultados del caso de estudio Ecological Island usando sus eventos originales.

Se observa que de los 181 mutantes generados, el número de mutantes muertos con los diferentes casos de prueba no varía mucho a pesar de usar en alguno de ellos el doble de eventos como parámetros de entrada.

Con el generador de eventos IoT-TEG se han generado 4 casos de prueba: 500, 1000, 2000 y 5000 eventos. Los eventos generados tienen valores totalmente aleatorios, los resultados de las cuatro ejecuciones se presentan en la tabla 9.12

	TC.1	TC.2	TC.3	TC.4
Eventos	500	1000	2000	5000
Muertos	109	112	111	112
Vivos	64	61	62	61
Erróneos	8	8	8	8

Tabla 9.12: Resultados del caso de estudio Ecological Island usando eventos generados por IoT-TEG.

Como se puede observar, el incremento del número de eventos no es significativo a la hora de obtener un mayor número de mutantes muertos. El número de mutantes muertos que se obtienen con estos casos de prueba generados son parecidos a los que se obtienen con los eventos que originalmente usa la aplicación.

Tras analizar los mutantes que quedan vivos en todos casos de prueba que han logrado matar a 112 mutantes (tanto con eventos originales, como con eventos generados con IoT-TEG), se comprueba que son los mismos. El siguiente paso es analizar cada una de las 61 mutaciones que no han sido posibles de detectar. Tras su estudio se comprueba que todas ellas son mutantes equivalentes ya que con las mutaciones aplicadas se va a obtener el mismo resultado que con el caso de prueba original. Por ejemplo, la razón que más se repite en la equivalencia de los mutantes se debe a los valores que pueden obtener los sensores involucrados en el caso de estudio, los cuales son booleanos (en formato numérico $\{0, 1\}$). En la mutación realizada se muta el operador relacional “sensor = 1” por “sensor >= 1”, luego siempre vamos a obtener los mismos resultados, ya que los sensores no van a obtener valores superiores a 1.

9.3.4. Caso de estudio - Domótica

Este caso de estudio es una versión del que está propuesto por Boubeta et al. en [164]. Este ejemplo simula los sensores de una casa inteligente que, dependiendo de la información que reciben en tiempo real, actúan en consecuencia. Los patrones implementados que incluye la aplicación, activa las siguientes acciones automáticas: consumo irresponsable de energía, detección de fuego, fallo de potencia y uso irresponsable de la televisión.

Como fuente de eventos se utiliza la plataforma IoT "Xively", una plataforma que ya no es abierta. El acceso a los eventos se ha realizado gracias a la cuenta que se había creado previamente el Dr. Boubeta. Esta plataforma ofrece los eventos de uno en uno, sin posibilidad de adquirir un lote de eventos, además hay que tener en consideración que no todos los canales de eventos se actualizan a la vez. Los canales que se emplean en esta aplicación como fuentes de eventos son: 62988, 71257 y 891225. Por otro lado, se tuvo que tener en cuenta que, de cada uno de los canales, se obtenía un tipo de evento diferente y que los valores de los atributos que se utilizaban de cada tipo podían tener distintas unidades (la temperatura podría venir en grados Celsius o Fahrenheit), así que se normalizaron los datos.

La aplicación está desarrollada para leer en formato `JSON` los eventos que se capturan de los diferentes canales, para ello esta incluye una clase que diferencia el tipo de evento según el canal del que le llegue la información. Para evitar hacer distinciones entre los tipos de eventos y para hacer las pruebas con la metodología propuesta en la presente tesis, se define el tipo de evento "DomoticEvent", que engloba los atributos que se utilizan de cada tipo de evento. Esta definición (véase la sección C.4) se emplea para generar eventos a través de IoT-TEG. Adicionalmente se modifica la aplicación para que en vez de leer los eventos de los diferentes canales, estos sean obtenidos de un fichero en formato `JSON` generado por IoT-TEG siguiendo la estructura del tipo de evento "DomoticEvent". En este caso de estudio hay involucradas 4 consultas EPL de EsperTech. Para obtener más información sobre este caso de estudio, véase el apéndice B.4.

La tabla 9.13 muestra qué operadores de mutación pueden aplicarse en las 4 consultas implicadas en el caso de estudio.

Los 13 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento y ventanas, siendo un total de 112 mutantes los que genera el *generador de mutantes* MuEPL.

Para el *sistema de ejecución* se necesitan los casos de prueba: el conjunto de eventos originales del caso de estudio y los generados por IoT-TEG.

Operador	Ocurrencia	Atributo
PNR	1	1
POM	2	2
PRE	4	1
POC	3	1
PFP	3	1
WTM	1	2
WBT	1	1
RLO	5	1
RTU	3	4
RAO	1	4
RRO	9	5
RNO	4	2
RSC	4	5

Tabla 9.13: Resultados tras aplicar el *analizador* con el caso de estudio Domótica.

Para los eventos originales se han almacenado en formato `JSON` los eventos de los canales: 62988, 71257 y 89125 de la plataforma ”Xively”. De todos los canales se han capturado un total de 21 eventos, teniendo un total de un caso de prueba: 13 eventos del canal 62988, 4 eventos del canal 71257 y 4 eventos del canal 89125. La plataforma “Xively”, además de ya no ser abierta, proporciona los eventos de uno en uno. También hay que tener en cuenta, que algunos canales están más activos que otros, luego la actualización de los datos obtenidos puede no ser tan frecuente. Esto último se confirma con los canales 71257 y 89125, que no se actualizan desde el año 2015 (los valores de los 4 eventos obtenidos de cada canal son iguales). El canal 62988 sigue activo, pero actualiza sus valores con poca frecuencia, como mucho una o dos veces al día (pudiéndose no actualizarse en un día); este es el motivo del número tan bajo de eventos originales. Los 21 eventos se almacenaron en un fichero en formato `JSON` para que se leyesen como un lote de eventos por el caso de estudio, convirtiéndose así en un único caso de prueba. Los resultados de la ejecución del caso de prueba se presentan en la tabla 9.14.

	TC.1
Canales	62988 | 71257 | 89125
Eventos	21
Muertos	30
Vivos	77
Erróneos	5

Tabla 9.14: Resultados del caso de estudio Domótica usando sus eventos originales.

Con el generador de eventos IoT-TEG se han generado 3 casos de prueba: uno de ellos con 100 eventos, otro con 150 eventos y otro con 200 eventos. Los eventos generados tienen valores totalmente aleatorios, los resultados de las tres ejecuciones se presentan en la tabla 9.15

Como se puede observar, se consiguen matar a 107 mutantes y se localizan 5 mutantes erróneos, algo que no se conseguía con los eventos originales. El uso del generador de

	TC.1	TC.2	TC.3
Eventos	100	150	200
Muertos	107	59	73
Vivos	0	48	34
Erróneos	5	5	5

TABLA 9.15: Resultados del caso de estudio Domótica usando eventos generados por IoT-TEG.

eventos IoT-TEG ha ayudado a realizar la prueba de mutaciones en este caso de estudio de forma satisfactoria. Uno de los problemas que se pueden encontrar es el cambio de política en una plataforma IoT (como es el caso de Xively). Al contar con la cuenta del Dr. Boubeta, esto fue un problema menor en nuestro estudio. Pero algo que se escapa del control del programador es la frecuencia y actualización de los canales de la plataforma IoT de los que depende la aplicación. Gracias al generador IoT-TEG, se han podido paliar estos dos inconvenientes.

9.3.5. Caso de estudio - DENMEvaP

Distributed Event-driven Network Monitoring Evaluation Prototype (DENMEvaP) [165] es un caso de estudio que consiste en un pequeño prototipo consistente en dos equipos:

1. Un nodo sensor que actúa como un productor de eventos y adquiere los datos en formato raw de la red monitorizada.
2. Un nodo de procesamiento de datos que prepara la infraestructura de comunicación, el motor CEP y el consumidor de eventos que se utiliza para medir el rendimiento.

Esta aplicación está compuesta por diversos módulos que están implementados para detectar diferentes comportamientos en una infraestructura de comunicaciones. Debido a que está implementado con el lenguaje de programación Clojure[2] y por recomendación de su autor, el Dr. Rüediger Gad, se ha ejecutado cada módulo de forma independiente, dando lugar a los siguientes programas:

- Brute Force
- Sniffer Flat
- Sniffer Flood
- Simple Sniffer

[2]Lenguaje de programación funcional (dialecto de Lisp), que entre sus características incluye una sintaxis concisa para la mejora de la concurrencia.

- Sniffer Congestion

Adicionalmente, el autor recomienda extraer de cada módulo los patrones EPL y no ejecutar su aplicación para realizar las pruebas. Así que para poder utilizar los patrones EPL se implementa una aplicación Java (para cada módulo) que captura los eventos que serán procesados por los patrones.

El formato original de los eventos que maneja este caso de estudio es `PCAP` (Packet CAPture). Para este estudio, los eventos originales que se han utilizado son los siguientes ficheros en formato `PCAP` almacenados en el repositorio de la aplicación:

1. `arp_spoofing_attack_attacker_2012-10-11` (6970 eventos)
2. `arp_spoofing_attack_victim_2012-11-13` (4712 eventos)
3. `dns-http-icmp_wlan_2013-11-30` (133 eventos)
4. `ssh_brute_force_sshpass_wlan0_2012-11-26` (6917 eventos)
5. `syn_flood_sender_non-rand_2012-11-07` (9439 eventos)
6. `syn_flood_sender_rand_2012-11-26` (10010 eventos)

No todos los eventos en estos ficheros PCAP contienen los mismos atributos, dependiendo del módulo del programa algunos serán procesados o no. Los eventos almacenados en estos ficheros se capturaron o generaron de diversas fuentes. Este conjunto de eventos será empleado en el conjunto de módulos que se derivan de la aplicación DENMEvaP. En este caso de estudio hay involucradas 37 consultas EPL de EsperTech. Para obtener más información sobre este caso de estudio y la metodología de generación o captura de los eventos en formato `PCAP`, véase el apéndice B.5.

Para la lectura de los eventos en formato `PCAP`, se utiliza una clase implementada por el autor que permite extraer los valores de los ficheros. Una vez que están almacenados se pueden lanzar los eventos al motor CEP. Para hacer las pruebas con la metodología propuesta en la presente tesis, se define un tipo de evento adecuado a cada módulo tras analizar los valores y atributos que procesa cada uno. Cada definición (véase la sección C.5) se emplea para generar eventos a través de IoT-TEG. Adicionalmente se modifica cada módulo para que en vez de leer los eventos con formato `PCAP`, estos sean obtenidos de un fichero en formato `JSON` que ha sido generado por IoT-TEG siguiendo la estructura del tipo de evento definido para cada módulo.

9.3.5.1. Módulo - Brute Force

La tabla 9.16) muestra los operadores de mutación que MuEPL determina que pueden aplicarse en las 6 consultas involucradas en este caso de estudio.

Operador	Ocurrencia	Atributo
PRE	3	1
POC	3	1
PFP	3	1
RLO	7	1
RAF	3	15
RAO	1	4
RRO	33	5
RNO	9	2
RGR	2	2
RSC	6	5
ICN	2	1
IWR	2	1

Tabla 9.16: Resultados tras aplicar el *analizador* con el caso de estudio Brute Force.

Los 12 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento y de inyecciones de ataque de SQL, siendo un total de 286 mutantes los que genera el *generador de mutantes* MuEPL.

Para pasar a la parte del *sistema de ejecución* se necesitan los casos de prueba con el conjunto de eventos a lanzar (tanto los originales del caso de estudio como los generados por IoT-TEG).

Para los eventos originales se han utilizados seis ficheros que contienen los eventos en formato `PCAP` (véase sección 9.3.5). Los resultados de las seis ejecuciones se presentan en la tabla 9.17.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	6970	4712	133	6917	10010	9439
Muertos	39	39	38	38	38	38
Vivos	247	247	248	248	248	248
Erróneos	0	0	0	0	0	0

Tabla 9.17: Resultados del caso de estudio Brute Force usando sus eventos originales.

Se observa que de los 286 mutantes generados, el número de mutantes muertos con los diferentes casos de prueba es casi siempre el mismo.

Con el generador de eventos IoT-TEG se han generado 3 casos de prueba con: 100 eventos, 200 eventos y 500 eventos siguiendo la definición del tipo de evento “BruteForce” (véase sección C.5.1). Esta definición se ha adecuado a este módulo, según el rango de los posibles valores que pueden tomar los atributos del tipo de evento. Los eventos generados tienen

valores totalmente aleatorios, los resultados de las tres ejecuciones se presentan en la tabla 9.18.

	TC.1	TC.2	TC.3
Eventos	100	200	500
Muertos	39	38	39
Vivos	247	248	247
Erróneos	0	0	0

TABLA 9.18: Resultados del caso de estudio Brute Force usando eventos generados por IoT-TEG.

El número de mutantes muertos que se obtienen con estos casos de prueba generados es muy parecido que los que se obtienen con los eventos que originalmente usa la aplicación en todos los casos de prueba con los eventos originales.

Una de las consultas involucradas en esta aplicación, sí cumple las condiciones necesarias para aplicar la funcionalidad avanzada de IoT-TEG:

```
INSERT INTO cep.ssh.connection.duration (startTimestamp, endTimestamp, duration,
source, destination, sourcePort, destinationPort)
  SELECT startTimestamp, endTimestamp, duration, source, destination, sourcePort,
  destinationPort
  FROM cep.tcp.connection.duration
  WHERE destinationPort = 22
```

El atributo `destinationPort` que aparece en la cláusula `where` de la consulta, está involucrado en una operación relacional y además está definido como tipo de dato simple. Como cumple las condiciones para aplicar la funcionalidad de lectura de consultas de IoT-TEG, se generan 3 casos de prueba con: 500 eventos, 1000 eventos y 5000 eventos. Los eventos generados tienen los siguientes valores:

1. Caso de prueba con 500 eventos: 250 eventos con `destinationPort` = 22 y 250 eventos con `destinationPort` con valores aleatorios.
2. Caso de prueba con 1000 eventos: 500 eventos con `destinationPort` = 22 y 500 eventos con `destinationPort` con valores aleatorios.
3. Caso de prueba con 5000 eventos: 2500 eventos con `destinationPort` = 22 y 2500 eventos con `destinationPort` con valores aleatorios.

Los resultados de las tres ejecuciones se presentan en la tabla 9.19.

Con 5000 eventos generados con la funcionalidad avanzada de IoT-TEG, se consiguen matar a 253 mutantes y se localizan 33 erróneos. Algo que no se conseguía con ningún

	TC.1	TC.2	TC.3
Eventos	500	1000	5000
Muertos	40	48	256
Vivos	246	238	0
Erróneos	0	0	33

Tabla 9.19: Resultados del caso de estudio Brute Force usando eventos generados por IoT-TEG y la funcionalidad de lectura de consultas.

caso de prueba usando eventos originales (véase tabla 9.17). Por tanto, el poder adaptar los valores de los atributos del tipo de evento al caso de estudio que se está analizando, gracias a la herramienta IoT-TEG, agiliza y mejora las pruebas.

9.3.5.2. Módulo - Sniffer Flat

Utilizando la herramienta MuEPL en este módulo del caso de estudio DENMEvaP, el *analizador* devuelve la salida de la tabla 9.20, tras comprobar qué operadores de mutación pueden aplicarse en las 11 consultas involucradas.

Operador	Ocurrencia	Atributo
PFP	5	1
POC	5	1
PRE	5	1
RAO	3	4
RLO	9	1
RNO	9	2
RRO	45	5
RSC	11	5
IWR	6	1
ICN	6	1

Tabla 9.20: Resultados tras aplicar el *analizador* con el caso de estudio Sniffer Flat.

Los 10 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento y de inyecciones de ataque de SQL, siendo un total de 347 mutantes los que genera el *generador de mutantes* MuEPL.

Para pasar a la parte del *sistema de ejecución* se necesitan los casos de prueba con el conjunto de eventos a lanzar (tanto los originales del caso de estudio como los generados por IoT-TEG).

Para los eventos originales se han utilizado los ficheros en formato `PCAP` explicados con anterioridad (véase sección 9.3.5). Los resultados de las seis ejecuciones se presentan en la tabla 9.21.

Se observa que de los 347 mutantes generados, el número de mutantes muertos con los diferentes casos de prueba varía mucho. Por los resultados obtenidos con estos casos de

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	6970	4712	133	6917	10010	9439
Muertos	335	335	101	335	32	127
Vivos	0	0	234	0	315	208
Erróneos	12	12	12	12	0	12

Tabla 9.21: Resultados del caso de estudio Sniffer Flat usando sus eventos originales.

prueba, se puede decir que el número de eventos no es un parámetro que afecte a la hora de matar los mutantes en este caso de estudio (véase TC.5).

Con el generador de eventos IoT-TEG se han generado 3 casos de prueba con: 100 eventos, 200 eventos y 500 eventos siguiendo la definición del tipo de evento "SnifferFlat" (véase sección C.5.2). Esta definición se ha adecuado a este módulo, según el rango de los posibles valores que pueden tomar los atributos del tipo de evento. Los eventos generados tienen valores totalmente aleatorios, los resultados de las tres ejecuciones se presentan en la tabla 9.22.

	TC.1	TC.2	TC.3
Eventos	100	200	500
Muertos	252	199	275
Vivos	84	137	72
Erróneos	11	11	0

Tabla 9.22: Resultados del caso de estudio Sniffer Flat usando eventos generados por IoT-TEG.

Los resultados de las ejecuciones con los eventos generados por IoT-TEG, muestran que con un número de eventos inferior al de los ficheros con eventos originales, se obtienen unos resultados relativamente buenos para la prueba de mutaciones en este caso de estudio. Los valores de los eventos que se han generado, según la definición del tipo de evento "SnifferFlat" han ayudado, ya que la definición del tipo de evento se ha adaptado a este caso de estudio.

Una de las consultas involucradas en esta aplicación, sí cumple las condiciones necesarias para aplicar la funcionalidad avanzada de IoT-TEG:

```
INSERT INTO cep.dns (timestamp, source, destination)
 SELECT ts, ipSrc, ipDst FROM sniffer.header.parsed.flat
 WHERE udpSrc = 53 OR udpDst = 53
```

Los atributos `udpDst` y `udpSrc` aparecen en la cláusula `where` de la consulta, están involucrados cada uno en una operación relacional y ambos en una operación lógica y, además, están definidos como de tipo de dato simple. Como cumplen las condiciones para aplicar la funcionalidad de lectura de consultas de IoT-TEG, se generan 5 casos

de prueba con: 100 eventos, 200 eventos, 500 eventos, 1000 eventos y 5000 eventos. Los eventos generados tienen los siguientes valores:

1. Caso de prueba con 100 eventos: 33 eventos con `udpSrc` = 53 y `udpDst` con valores aleatorios, otros 33 eventos con `udpDst` = 53 y `udpSrc` con valores aleatorios y 34 eventos con `udpSrc` y `udpDst` con valores aleatorios.
2. Caso de prueba con 200 eventos: 66 eventos con `udpSrc` = 53 y `udpDst` con valores aleatorios, otros 66 eventos con `udpDst` = 53 y `udpSrc` con valores aleatorios y 68 eventos con `udpSrc` y `udpDst` con valores aleatorios.
3. Caso de prueba con 500 eventos: 166 eventos con `udpSrc` = 53 y `udpDst` con valores aleatorios, otros 166 eventos con `udpDst` = 53 y `udpSrc` con valores aleatorios y 168 eventos con `udpSrc` y `udpDst` con valores aleatorios.
4. Caso de prueba con 1000 eventos: 333 eventos con `udpSrc` = 53 y `udpDst` con valores aleatorios, otros 333 eventos con `udpDst` = 53 y `udpSrc` con valores aleatorios y 334 eventos con `udpSrc` y `udpDst` con valores aleatorios.
5. Caso de prueba con 5000 eventos: 1666 eventos con `udpSrc` = 53 y `udpDst` con valores aleatorios, otros 1666 eventos con `udpDst` = 53 y `udpSrc` con valores aleatorios y 1668 eventos con `udpSrc` y `udpDst` con valores aleatorios.

Los resultados de las cinco ejecuciones se presentan en la tabla 9.23.

	TC.1	TC.2	TC.3	TC.4	TC.5
Eventos	100	200	500	1000	5000
Muertos	173	147	288	336	318
Vivos	166	190	59	0	0
Erróneos	8	10	0	11	29

TABLA 9.23: Resultados del caso de estudio Sniffer Flat usando eventos generados por IoT-TEG y la funcionalidad de lectura de consultas.

Con 1000 eventos generados con la funcionalidad avanzada de IoT-TEG, se consigue matar a 336 mutantes y localizar 11 mutantes erróneos. Esto se conseguía con varios casos de prueba usando eventos originales (véase tabla 9.21), pero en los que se empleaban un número elevado de eventos (siempre superior a 4000 eventos). Nuevamente podemos comprobar cómo el poder adaptar los valores de los atributos del tipo de evento al caso de estudio que se está analizando, gracias a la herramienta IoT-TEG, se agilizan y se mejoran las pruebas.

9.3.5.3. Módulo - Sniffer Flood

Siguiendo los pasos de la herramienta MuEPL, el *analizador* devuelve como salida la tabla 9.24 tras comprobar qué operadores de mutación pueden aplicarse en las 7 consultas que contiene la aplicación.

Operador	Ocurrencia	Atributo
PFP	2	1
PNR	1	1
POC	2	1
POM	1	2
PRE	2	1
RGR	8	2
RLO	3	1
RNO	8	2
RRO	20	5
RSC	7	5
RTU	18	4
WTM	4	2
WBT	4	1
ICN	1	1
IWR	1	1

Tabla 9.24: Resultados tras aplicar el *analizador* con el caso de estudio Sniffer Flood.

Los 14 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento, de inyecciones de ataque de SQL y ventanas, siendo un total de 265 mutantes los que genera el *generador de mutantes* MuEPL.

Para pasar a la parte del *sistema de ejecución* se necesitan los casos de prueba con el conjunto de eventos a lanzar (tanto los originales del caso de estudio como los generados por IoT-TEG).

Para los eventos originales se han utilizado los ficheros en formato `PCAP` explicados con anterioridad (véase sección 9.3.5.1). Los resultados de las seis ejecuciones se presentan en la tabla 9.25.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	6970	4712	133	6917	10010	9439
Muertos	260	246	73	256	262	256
Vivos	2	16	189	6	0	6
Erróneos	3	3	3	3	3	3

Tabla 9.25: Resultados del caso de estudio Sniffer Flood usando sus eventos originales.

Se observa que de los 265 mutantes generados, el número de mutantes muertos con los diferentes casos de prueba varía mucho. En este caso de estudio parece que cuanto mayor sea el número de eventos, mayor número de mutantes muertos se obtiene.

Con el generador de eventos IoT-TEG se han generado 6 casos de prueba con: 100 eventos, 200 eventos, 500 eventos, 1000 eventos, 10000 eventos y 20000 eventos siguiendo la definición del tipo de evento "SnifferFlood" (véase sección C.5.3). Esta definición se ha adecuado a este módulo, según el rango de los posibles valores que pueden tomar los atributos del tipo de evento. Los eventos generados tienen valores totalmente aleatorios, los resultados de las seis ejecuciones se presentan en la tabla 9.26.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	100	200	500	1000	10000	20000
Muertos	22	22	22	22	188	159
Vivos	240	240	240	240	77	106
Erróneos	3	3	3	3	0	0

Tabla 9.26: Resultados del caso de estudio Sniffer Flood usando eventos generados por IoT-TEG.

Con los eventos generados de IoT-TEG, hasta que no generamos 10000 eventos, no aumenta el número de mutantes muertos. Se comprueba si con la funcionalidad avanzada de IoT-TEG se consigue incrementar este valor. La consulta que cumple las condiciones necesarias para aplicar la funcionalidad avanzada es la siguiente:

```
INSERT INTO cep.tcp.syn.sent
 (timestamp, source, destination, sourcePort, destinationPort)
 SELECT ts, ipSrc, ipDst, tcpSrc, tcpDst
 FROM sniffer.header.parsed.flat
 WHERE tcpFlags = 2
```

El atributo `tcpFlags` aparece en la cláusula `where` de la consulta, está involucrado en una operación relacional y, además, está definido como un tipo de dato simple. Como cumple las condiciones para aplicar la funcionalidad de lectura de consultas de IoT-TEG, se generan 3 casos de prueba con: 100 eventos, 1000 eventos y 10000 eventos. Los eventos generados tienen los siguientes valores:

1. Caso de prueba con 100 eventos: 50 eventos con `tcpFlags` = 2 y 50 eventos con `tcpFlags` con valores aleatorios.

2. Caso de prueba con 1000 eventos: 500 eventos con `tcpFlags` = 2 y 500 eventos con `tcpFlags` con valores aleatorios.

3. Caso de prueba con 10000 eventos: 5000 eventos con `tcpFlags` = 2 y 5000 eventos con `tcpFlags` con valores aleatorios.

Los resultados de las tres ejecuciones se presentan en la tabla 9.27.

	TC.1	TC.2	TC.3
Eventos	100	1000	10000
Muertos	22	25	195
Vivos	240	237	76
Erróneos	3	3	0

TABLA 9.27: Resultados del caso de estudio Sniffer Flood usando eventos generados por IoT-TEG aplicando la funcionalidad avanzada.

En comparación con las salidas de la tabla 9.26 el número de mutantes muertos se incrementa, pero no se consigue igualar al número de mutantes muertos con los eventos originales. Tras analizar el resto de consultas involucradas, se observa la existencia de patrones de eventos (véase sección 2.4.2) con diversas condiciones en donde están implicados los mutantes que no logramos matar con los eventos generados con IoT-TEG. Estos son tan complejos y específicos que, si no se preparan correctamente los casos de prueba, es muy difícil detectar los mutantes. No hay que olvidar que los casos de prueba originales fueron generados y preparados por el autor de la herramienta para comprobar la correcta ejecución de todas las consultas involucradas en su programa. Esto significa que hay que seguir trabajando en analizar en profundidad las consultas EPL de EsperTech involucradas para generar los valores de los atributos de los eventos acorde a las pruebas que se quieran realizar.

9.3.5.4. Módulo - Simple Sniffer

En este módulo, el *analizador* de la herramienta MuEPL, proporciona los operadores de mutación que pueden aplicarse en las 9 consultas que contiene la aplicación, véase tabla 9.28.

OPERADOR	OCURRENCIA	ATRIBUTO
PFP	4	1
POC	4	1
PRE	4	1
RAO	2	4
RLO	8	1
RNO	7	2
RRO	37	5
RSC	9	5
ICN	5	1
IWR	5	1

TABLA 9.28: Resultados tras aplicar el *analizador* con el caso de estudio Simple Sniffer.

Los 10 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento y de inyecciones de ataque de SQL, siendo un total de 282 mutantes los que genera el *generador de mutantes* MuEPL.

Para pasar a la parte del *sistema de ejecución* se necesitan los casos de prueba con el conjunto de eventos a lanzar (tanto los originales del caso de estudio como los generados por IoT-TEG).

Para los eventos originales se han utilizado los ficheros en formato `PCAP` explicados con anterioridad. Los resultados de las seis ejecuciones se presentan en la tabla 9.29.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	6970	4712	133	6917	10010	9439
Muertos	274	269	29	234	18	264
Vivos	0	5	245	40	264	10
Erróneos	8	8	8	8	0	8

Tabla 9.29: Resultados del caso de estudio Simple Sniffer usando sus eventos originales.

Se observa que de los 282 mutantes generados, el número de mutantes muertos con los diferentes casos de prueba varía mucho. El número de eventos no es un parámetro que afecte a la hora de matar los mutantes en este caso de estudio, véanse los resultados obtenidos con el TC.5.

Con el generador de eventos IoT-TEG se han generado 3 casos de prueba con: 100 eventos, 200 eventos y 500 eventos siguiendo la definición del tipo de evento "SimpleSniffer" (véase sección C.5.5). Esta definición se ha adecuado a este módulo, según el rango de los posibles valores que pueden tomar los atributos del tipo de evento. Los eventos generados tienen valores totalmente aleatorios, los resultados de las tres ejecuciones se presentan en la tabla 9.30.

	TC.1	TC.2	TC.3
Eventos	100	200	500
Muertos	160	194	274
Vivos	122	80	0
Erróneos	0	8	8

Tabla 9.30: Resultados del caso de estudio Simple Sniffer usando eventos generados por IoT-TEG.

Con 500 eventos generados con IoT-TEG, se consigue matar a 274 mutantes y localizar los 8 mutantes erróneos. Algo que solo se conseguía con el caso de prueba TC.1 usando eventos originales (véase tabla 9.29) en el que se empleaban casi 7000 eventos. Una vez más se comprueba que adaptar los valores de los atributos del tipo de evento, con IoT-TEG, facilita y mejora la ejecución de las pruebas.

9.3.5.5. Módulo - Sniffer Congestion

La tabla 9.31 muestra los operadores que el *analizador* de MuEPL detecta que pueden aplicarse en las 4 consultas que contiene la aplicación.

Operador	Ocurrencia	Atributo
PRE	1	1
POC	1	1
PFP	1	1
RLO	1	1
RAO	1	4
RRO	5	5
RNO	1	2
RSC	4	5
ICN	3	1
IWR	3	1

Tabla 9.31: Resultados tras aplicar el *analizador* con el caso de estudio Sniffer Congestion.

Los 10 operadores de mutación que se pueden aplicar son de las categorías: expresiones de patrones, reemplazamiento y de inyecciones de ataque de SQL, siendo un total de 61 mutantes los que genera el *generador de mutantes* MuEPL.

Para pasar a la parte del *sistema de ejecución* se necesitan los casos de prueba con el conjunto de eventos a lanzar (tanto los originales del caso de estudio como los generados por IoT-TEG).

Para los eventos originales se han utilizado los ficheros en formato `PCAP` explicados con anterioridad (véase sección 9.3.5.1). Los resultados de las seis ejecuciones se presentan en la tabla 9.32.

	TC.1	TC.2	TC.3	TC.4	TC.5	TC.6
Eventos	6970	4712	133	6917	10010	9439
Muertos	38	28	25	37	4	12
Vivos	21	31	34	22	57	49
Erróneos	2	2	2	2	0	0

Tabla 9.32: Resultados del caso de estudio Sniffer Congestion usando sus eventos originales.

Se observa que de los 61 mutantes generados, el número de mutantes muertos con los diferentes casos de prueba varía mucho. El número de eventos no es un parámetro que afecte a la hora de matar los mutantes en este caso de estudio, véanse los resultados obtenidos con el TC.5.

Con el generador de eventos IoT-TEG se han generado 3 casos de prueba con: 100 eventos, 200 eventos y 500 eventos siguiendo la definición del tipo de evento “SimpleSniffer” (véase sección C.5.5). Esta definición se ha adecuado a este módulo, según el rango de los posibles

valores que pueden tomar los atributos del tipo de evento. Los eventos generados tienen valores totalmente aleatorios, los resultados de las tres ejecuciones se presentan en la tabla 9.33.

	TC.1	TC.2	TC.3
Eventos	100	200	500
Muertos	35	35	35
Vivos	24	24	24
Erróneos	2	2	2

Tabla 9.33: Resultados del caso de estudio Sniffer Congestion usando eventos generados por IoT-TEG.

Como se puede observar, con los eventos generados con IoT-TEG, el incremento del número de eventos no es significativo a la hora de obtener un mayor número de mutantes muertos. El número de mutantes muertos que se obtienen con estos casos de prueba generados (35 mutantes muertos) es menor que los que se obtienen con los eventos que originalmente usa la aplicación en dos de los seis casos de prueba con los eventos originales (38 para TC.1 y 37 para TC.4), superándolos en los otros cuatro. Al igual que en el módulo "Sniffer Flood" (véase sección 9.3.5.3) en las consultas existen una serie de patrones de eventos (véase sección 2.4.2) con condiciones en donde están implicados los mutantes que no logramos matar con los eventos generados con IoT-TEG. Esto significa que hay que seguir trabajando en analizar en profundidad las consultas EPL de EsperTech involucradas para generar los valores de los atributos de los eventos acorde a las pruebas que se quieran realizar.

9.4. Análisis de operadores de mutación definidos para EPL de EsperTech

En las tablas 7.3 y 7.4 se presentan el nombre y una descripción breve de los 34 operadores de mutación considerados para el lenguaje de programación EPL de EsperTech. Tal y como se ha comentado, algunos de los operadores de mutación son "reutilizados" de los del lenguaje de programación SQL debido a su similitud. No obstante, como se pretenden definir los operadores de mutación de los fallos comunes propios de este lenguaje, se analizan programas de usuarios que utilizan este lenguaje de programación para localizarlos.

Tras ese estudio se establece que los operadores propios de EPL de EsperTech son: todos los operadores que están dentro de la categoría "Operadores de expresiones de patrones", todos los operadores de la categoría "Operadores de ventanas", los operadores de la

categoría “Operadores de reemplazo” RRR_1, RRR_2 y RTU, y el operador de la categoría de inyecciones de ataque IWR (véanse tabla 7.3 y tabla 7.4).

Si consideramos las nueve aplicaciones en las que hemos aplicado la prueba de mutaciones (se recuerda la división realizada con el caso de estudio DENMEvaP); tras analizar las salidas del componente *Analizador* de MuEPL, se observa que en todas las aplicaciones menos en una, “Transaction” (véase sección 9.3.2), aparecen al menos tres de los seis operadores de la categoría “Operadores de expresiones de patrones”. Los operadores de la categoría “Operadores de ventanas”, aparecen en cinco de las nueve aplicaciones y, como mínimo, dos de los cuatro operadores se aplican. La misma ocurrencia (cinco de las nueve aplicaciones) tiene el operador de reemplazo RTU, algo lógico ya que las unidades de tiempo que reemplaza están localizadas en ventanas de las consultas EPL de EsperTech. Se destaca que el operador IWR aparece en siete de las nueve aplicaciones; no se aplica en “Ecological Island” (véase sección 9.3.3) ni en “Domótica” (véase sección 9.3.4). Por último, los operadores de reemplazo RRR_1 y RRR_2 no aparecen en ningún caso de estudio, las mutaciones que realizan estos operadores son en las funciones `“single row”` exclusivas de este lenguaje de programación. Esta exclusividad puede significar que; el uso de estas funciones no esté muy difundido entre los “nuevos” programadores (de ahí que no se haya encontrado ninguna ocurrencia de estos operadores en los casos de estudio), o que no sean utilizadas tan frecuentemente como para considerarlas dentro de los fallos comunes en este lenguaje de programación.

Como consecuencia se podría decir que los operadores de mutación propios de EPL de EsperTech definidos en la presente tesis (a excepción de RRR_1 y RRR_2) aparecen frecuentemente en las consultas de este lenguaje, siendo indispensable considerarlos para hacer este tipo de pruebas.

9.4.1. Análisis de mutantes potencialmente equivalentes

En la tabla 9.34 se observa el análisis de los mutantes potencialmente equivalentes generados en cada caso de estudio por cada operador de mutación.

Para el caso de estudio DENMEvaP se han agrupado los resultados de todos los módulos en los que se dividió a la hora de hacer las pruebas (solo se encuentran mutantes potencialmente equivalentes en el módulo “Sniffer Congestion”). De los tres casos de estudio donde se han encontrado, se destaca el operador RSC que aparece en los tres, aunque es el operador RAF el que más operadores potencialmente equivalentes genera seguido de RTU y RSC. El caso de estudio donde el operador RTU obtiene un número mayor de mutantes equivalentes, Transaction (véase sección 9.3.2), se observó que las mutaciones aplicadas no afectaban al resultado según el contexto de la aplicación (la mutación en

Caso de estudio	Equiv.	Operadores
Terminal	0	-
Transaction	157	ICN 1, IWR 1, RAF 75, RJR 6, RNW 1, RSC 17 RTU 36, WBT 3, WTM 17
Eco. Island	61	RTU 16, RNO 6, RRO 14, RSC 20, WBT 1, WTM 4
Domótica	0	-
DENMEvaP	3	RRO 1, RSC 2

Tabla 9.34: Análisis del tipo de mutantes generados en cada caso de estudio según el operador de mutación.

la unidad de tiempo no afectaba a la consulta). En la mayoría de ocurrencias de este operador en otros casos de estudio, sí se localizan fácilmente las mutaciones generadas. El operador RSC aparece en otros casos de estudio y los mutantes que ha generado han podido matarse (aunque sí que se ha observado que en muchas ocasiones con cierta dificultad, solo un caso de prueba lo mata).

La mayoría de los mutantes potencialmente equivalentes detectados, se ha observado que son equivalentes según el contexto de la aplicación; tal y como se explicó en el caso de estudio "Ecological Island" (véase sección 9.3.3), si un parámetro solo puede obtener valores dentro de un cierto rango $\{0, 1\}$ y la mutación es el operador relacional "parametro = 1" por "parametro >= 1", como los valores del parámetro solo van a ser $\{0, 1\}$, siempre se obtendrá el mismo resultado.

Además se observa que de los 11 operadores que generan mutantes equivalentes, solo 3 son de los que se han denominado propios del lenguaje EPL de EsperTech.

Si se analizan de forma independiente cada uno de los operadores que generan mutantes equivalentes se extrae que:

Operador ICN Puede aplicarse en 5 casos de estudio de los 9 considerados (si se consideran independientes los módulos del caso de estudio DENMEvaP), siendo un total de 17 ocurrencias en todos ellos, generándose un total de $17 \times 1 = 17$ mutantes (siendo 1 el valor máximo del atributo, véase tabla 8.1). De esos 17 únicamente un mutante generado ha resultado equivalente.

Operador IWR Puede aplicarse en 7 casos de estudio de los 9 considerados, siendo un total de 20 ocurrencias en todos ellos, generándose un total de $20 \times 1 = 20$ mutantes (siendo 1 el valor máximo del atributo, véase tabla 8.1). De esos 20 únicamente un mutante generado ha resultado equivalente.

Operador RAF Puede aplicarse en 2 casos de estudio de los 9 considerados, siendo un total de 18 ocurrencias en todos ellos, generándose un total de $18 \times 15 = 270$ mutantes (siendo 15 el valor máximo del atributo, véase tabla 8.1). De esos 270 mutantes generados, 75 han resultado equivalentes.

Operador RJR Puede aplicarse en 1 caso de estudio de los 9 considerados, siendo un total de 2 ocurrencias en todos ellos, generándose un total de $2 \times 4 = 8$ (siendo 4 el valor máximo del atributo, véase tabla 8.1). De esos 8 mutantes generados, 6 han resultado equivalentes.

Operador RNW Puede aplicarse en 1 caso de estudio de los 9 considerados, siendo un total de 1 ocurrencia en todos ellos, generándose un total de $1 \times 1 = 1$ (siendo 1 el valor máximo del atributo, véase tabla 8.1). El único mutante generado ha resultado ser equivalente.

Operador RSC Puede aplicarse en todos los casos de estudio, siendo un total de 56 ocurrencias en todos ellos, generándose un total de $56 \times 5 = 280$ (siendo 5 el valor máximo del atributo, véase tabla 8.1). De esos 280 mutantes generados, 39 han resultado equivalentes.

Operador RTU Puede aplicarse en 5 casos de estudio de los 9 considerados, siendo un total de 39 ocurrencias en todos ellos, generándose un total de $39 \times 4 = 156$ mutantes (siendo 4 el valor máximo del atributo, véase tabla 8.1). De esos 156 mutantes generados, 52 han resultado equivalentes.

Operador WBT Puede aplicarse en 5 casos de estudio de los 9 considerados, siendo un total de 19 ocurrencias en todos ellos, generándose un total de $19 \times 1 = 19$ mutantes (siendo 1 el valor máximo del atributo, véase tabla 8.1). De esos 19 mutantes generados, 4 han resultado equivalentes.

Operador WTM Puede aplicarse en 5 casos de estudio de los 9 considerados, siendo un total de 19 ocurrencias en todos ellos, generándose un total de $19 \times 2 = 19$ mutantes (siendo 2 el valor máximo del atributo, véase tabla 8.1). De esos 38 mutantes generados, 21 han resultado equivalentes.

Operador RRO Puede aplicarse en todos los casos de estudio, siendo un total de 173 ocurrencias en todos ellos, generándose un total de $173 \times 5 = 865$ (siendo 5 el valor máximo del atributo, véase tabla 8.1). De esos 865 mutantes generados, 15 han resultado equivalentes.

Operador RNO Puede aplicarse en 8 casos de estudio de los 9 considerados, siendo un total de 62 ocurrencias en todos ellos, generándose un total de $62 \times 2 = 124$

mutantes (siendo 2 el valor máximo del atributo, véase tabla 8.1). De esos 124 mutantes generados, 6 han resultado equivalentes.

Este análisis permite observar que los operadores ICN, IWR, RNO y RRO, generan aproximadamente un 5 % o menos de mutantes equivalentes. Por el contrario, los operadores RAF, RJR, RNW, RSC, RTU, WBT y WTM requieren como trabajo futuro estudiar su redefinición para reducir el número de mutantes equivalentes que producen. Esta redefinición requerirá buscar nuevos casos de estudio para analizar la aplicación de estos operadores.

9.5. Reducción del tiempo de ejecución

Tal y como se ha explicado en la sección 8.2.4, la ejecución en paralelo implementada, hace que los hilos de ejecución (programa original y mutantes) se esperen mutuamente en una “barrera” que, una vez alcanzada por todos, permite la ejecución en paralelo de todos los hilos. La máquina donde se quiera ejecutar MuEPL utilizando esta clase de Java, tiene que contar con una memoria RAM suficiente que permita la ejecución en paralelo del conjunto de programas mutantes y original. Esto dependerá del tamaño que ocupe la aplicación a probar; si no es suficiente se puede parar la máquina o no lograr el objetivo de reducción del tiempo de ejecución.

Para demostrar la reducción de tiempo de ejecución, se ha utilizado el módulo “Sniffer Congestion” del caso de estudio DENMEvaP (véase sección B.5). Se ha ejecutado bajo MuEPL un total de 20 veces utilizando los siguientes casos de prueba: con 100 eventos (ejecutado 10 veces, 5 usando la paralelización y 5 de forma secuencial), con 500 eventos (ejecutado 4 veces, 2 usando la paralelización y 2 de forma secuencial), con 1000 eventos (ejecutado 4 veces, 2 usando la paralelización y 2 de forma secuencial) y otro con 10000 eventos (ejecutado 2 veces, 1 usando la paralelización y 1 de forma secuencial) en la misma máquina (es decir, en todas ejecuciones se estaba bajo las mismas condiciones). En la tabla 9.35 se muestran los tiempos de ejecución obtenidos utilizando y sin utilizar la paralelización.

En la columna Ejecución se comparan las ejecuciones usando y sin usar la paralelización, en la columna N. eventos se recogen el número de eventos que intervienen en la ejecución y en la columna T. ejecución el tiempo en milisegundos que ha tardado la ejecución. Como se puede comprobar el tiempo de ejecución con la paralelización se reduce considerablemente respecto a una ejecución secuencial de los programas mutantes.

Ejecución	N. eventos	T. ejecución
Secuencial 1	100	2199102
Paralela 1	100	102298
Secuencial 2	100	2196618
Paralela 2	100	101081
Secuencial 3	100	2198173
Paralela 3	100	140796
Secuencial 4	100	2198925
Paralela 4	100	100632
Secuencial 5	100	2196484
Paralela 5	100	100809
Secuencial 1	500	10199118
Paralela 1	500	186894
Secuencial 2	500	10197481
Paralela 2	500	186687
Secuencial 3	1000	19205627
Paralela 3	1000	239345
Secuencial 4	1000	19234292
Paralela 4	1000	279632
Secuencial 5	10000	169486500
Paralela 5	10000	1790484

Tabla 9.35: Resultados de las ejecuciones del módulo Sniffer Congestion usando y sin usar la paralelización.

9.6. Usabilidad del generador de eventos IoT-TEG

Tal y como se ha mostrado en los resultados obtenidos en la presente tesis, la especificación para la definición de tipos de evento, junto con la herramienta IoT-TEG, permite generar una amplia variedad de eventos para diversos contextos. Esto es gracias a las propiedades que incluye la especificación y a la flexibilidad con la que esta permite definir la estructura del tipo de evento.

Esta propuesta de especificación para la definición de tipos de eventos, junto con la herramienta IoT-TEG, permiten obtener un conjunto amplio y variado de casos de prueba (eventos) para aquellas aplicaciones que los procesan. A la hora de realizar las pruebas en las aplicaciones IoT, son los desarrolladores de las herramientas los únicos que conocen el rango de valores de los atributos de los eventos que procesan su aplicación. Si no se conocen estos rangos de valores, no podemos hacer una definición de tipo de evento más precisa, que permita obtener una mayor cobertura.

El conjunto de casos de prueba que se genere a través del método propuesto, variará según la prueba que se quiera realizar al programa. La generación de eventos de prueba dependerá del tipo de prueba a realizar; el desarrollador encargado de esta labor definirá el tipo de evento y el rango de valores de sus atributos según el tipo de prueba:

- Para pruebas de rendimiento, dependiendo de qué funcionalidad o parte del programa se quiera medir el rendimiento, los eventos que se generen necesitarán o no unos valores específicos. Si, por ejemplo, se quiere medir el rendimiento de una funcionalidad que esté ligada a una ventana de tiempo, se generarán eventos cuyos valores apliquen más o menos carga a esta parte del sistema. Si lo que se quiere medir es el rendimiento general de la aplicación, se generarán eventos según los valores determinados de los atributos del tipo de evento, y la cantidad de eventos irá acorde con un funcionamiento normal de la aplicación o dispositivo que se esté probando.

- Para las pruebas de estrés, se necesita un conjunto de eventos que ayude a determinar si la aplicación aguanta esa carga continuada de información. Con esta prueba se quiere comprobar si hay alguna fuga de memoria en la aplicación. Al igual que con las pruebas de rendimiento, los valores de los eventos dependerán de qué funcionalidad, o parte de la aplicación se le quiera aplicar la prueba de estrés.

- Las pruebas negativas son un tipo de prueba que pretende determinar el funcionamiento del sistema cuando este recibe eventos con valores incorrectos en sus atributos. Este tipo de prueba se está empezando a incluir en los sistemas IoT. Los eventos generados podrán diferir tanto en la estructura del tipo de evento como en los valores de los atributos del tipo de evento. Si por ejemplo el atributo de un tipo de evento es de tipo "Entero", se modificará en la definición del tipo de evento para que este atributo sea de tipo "Cadena". De este modo se podrá observar el funcionamiento del dispositivo cuando procesa un evento cuyos atributos no tienen los valores correctos; si el dispositivo esperaba un evento con un atributo de tipo "Entero" que ahora es de tipo "Cadena" se comprobará su reacción al procesar este evento (se parará, dará un error, lo procesará sin problemas...).

- Para pruebas unitarias, los eventos que se generen necesitarán unos valores específicos en los atributos del tipo de evento que espere el dispositivo o la parte del código a probar. En el caso de querer comprobar si se ejecutan las alarmas o actividades correspondientes cuando se activa un sensor, se modificará en la definición del tipo de evento el valor del atributo que representa a el sensor (sensorA = true, por ejemplo). De este modo se podrá comprobar si estando activo este sensor se ejecutan las alarmas o actividades correspondientes.

- Para las pruebas de integración, los eventos que se generen necesitarán una estructura específica según el tipo de evento que se espere en el nuevo dispositivo a integrar, por ejemplo cuando quiere hacer una simulación antes de que el componente físico se integre en el sistema. Tampoco sería necesaria una gran cantidad de eventos. Para este tipo de pruebas se podrían utilizar el conjunto de eventos

generados para las pruebas unitarias de los distintos dispositivos que formen el sistema, o generar nuevos eventos utilizando las mismas definiciones de tipos de evento empleadas en las pruebas unitarias de cada dispositivo.

- El conjunto de casos de prueba obtenido de las pruebas negativas, unitarias y de integración, se podrá utilizar para realizar pruebas de regresión de todo el sistema IoT. Ya que cada dispositivo ha sido probado con su conjunto de casos de prueba específico y se ha comprobado su funcionamiento, ahora se podrá observar y probar el sistema completo con el conjunto de casos de prueba.

Como puede observarse, el desarrollador podrá generar el conjunto de casos de prueba según sus necesidades y las pruebas que quiera realizar. Como con cualquier software, si se realizan pruebas desde los inicios de su implementación, se podrá observar el funcionamiento del mismo y cualquier error de implementación podrá detectarse más rápidamente. El conjunto de casos de prueba obtenido a lo largo de la implementación, servirá como conjunto de pruebas final para evaluar el sistema. Por otro lado, dado que se conoce su funcionamiento con ese conjunto ya probado, se podrá generar otro conjunto de eventos de prueba para comparar y contrastar el funcionamiento del sistema.

Para cualquiera de las pruebas a realizar en una aplicación IoT, es la definición de tipo de evento que elabore el desarrollador, la que determinará la completa cobertura de los valores de los atributos del tipo de evento.

9.7. Conclusiones

La primera de las pruebas realizada era definir diferentes tipos de eventos con la propuesta de especificación presentada en la presente tesis. Los eventos que se definieron para la primera prueba provenían de la plataforma "ThingSpeak"; un total de 450 tipos de eventos que pudieron definirse utilizando la propuesta de especificación diseñada en esta tesis, que como ya se ha mencionado sigue las pautas de Luckham [93]. Seguir dichas pautas, ha permitido que todos los tipos de eventos analizados de la plataforma puedan ser definidos, así como los tipos de eventos que se emplean en los casos de estudio utilizados en esta tesis.

Se ha determinado que el *killing criteria* de este tipo de aplicaciones que procesan eventos es el contenido y el número de eventos procesados por el conjunto de las consultas EPL de EsperTech que conforman el programa que está siendo estudiado. Se ha llegado a esta conclusión tras comprobar que al ejecutar dos veces el mismo programa bajo las mismas condiciones (misma máquina y mismos datos de entrada), el tiempo de ejecución

variaba considerablemente y el contenido de las salidas (los eventos procesados) también se veía afectado en el orden. Tras comprobar que a pesar de obtener tiempos diferentes de ejecución y un orden distinto de los eventos procesados, el contenido y el número de eventos de las salidas era el mismo, se establece este como *killing criteria*.

Por otro lado se han ejecutado los 5 casos de estudio, 9 aplicaciones en total si se contabilizan de forma independiente los módulos en los que se han dividido para hacer las pruebas el caso de estudio DENMEvaP (véase sección 9.3.5), con el sistema de generación y ejecución de mutantes MuEPL (capítulo 8). Se han utilizado en las pruebas tanto sus eventos originales[3], como los generados a través del método propuesto en la presente tesis (capítulo 5).

En 7 de las 9 aplicaciones, se consigue matar al máximo número de mutantes (teniendo en cuenta los mutantes equivalentes) con los casos de prueba generados por IoT-TEG (véase capítulo 6). Cabe resaltar que esto último no se consigue en la mayoría de los casos de estudio usando los eventos originales, es cuando se usan los casos de prueba generados por IoT-TEG cuando se logra. En los casos de estudio donde sí se consigue matar a la mayoría de mutantes con casos de prueba originales, existe un caso de prueba generado por IoT-TEG que también lo consigue, pero con un número bastante más reducido de eventos. La funcionalidad de lectura de consultas tiene una gran importancia, ya que es la que ayuda a obtener un conjunto de prueba de mayor calidad con el que conseguir matar a ese máximo número de mutantes.

En las otras 2 aplicaciones (ambas incluidas en el caso de estudio DENMEvaP), en una se consigue un número aproximado de mutantes muertos y, en la otra, el número de mutantes muertos es inferior (comparando con los casos de prueba con eventos originales). Tras analizar los mutantes vivos en ambas, se observa que estos se encuentran aplicados en patrones de eventos en los que existen diversas condiciones a cumplir. Estos patrones de eventos son tan complejos y específicos que hacen muy difícil detectar los mutantes. Si bien es cierto que los casos de prueba originales consiguen matar a estos eventos, no hay que olvidar que estos se generaron y prepararon por el autor de la herramienta para comprobar la correcta ejecución de todas las consultas involucradas. Esto significa que la funcionalidad de lectura de consultas ayuda a generar conjuntos de casos de prueba de mayor calidad para la prueba de mutaciones, pero como trabajo futuro hay que seguir trabajando en la evaluación de las restricciones de las consultas EPL de EsperTech involucradas para realizar este tipo de pruebas.

[3]Se llaman eventos originales aquellos con los que la aplicación originalmente se ejecuta, bien sea eventos de un servidor o plataforma pública, eventos generados por un generador de eventos propio de la aplicación, eventos almacenados en ficheros para hacer pruebas...

Adicionalmente, se analiza la definición de los operadores de mutación específicos de EPL de EsperTech, ya que en la gran mayoría de los casos de estudio utilizados aparecen los operadores considerados como propios de este lenguaje. Los operadores de la categoría "Operadores de expresiones de patrones", pueden aplicarse (como mínimo tres de los seis) en 8 de las 9 aplicaciones, los operadores de la categoría "Operadores de ventanas", pueden aplicarse (como mínimo dos de los cuatro) en 5 de las 9 aplicaciones al igual que el operador de reemplazo RTU y, el de inyección de ataque IWR se aplica en 7 de las 9 aplicaciones. Sin embargo, RRR_1 y RRR_2 no aparecen en ningún caso de estudio, esto puede ser por la mutación que realizan en las funciones `"single row"`; el uso de estas funciones puede no estar muy difundido entre los "nuevos" programadores, o que no sean utilizadas tan frecuentemente como para considerarlas dentro de los fallos comunes en este lenguaje de programación. Además, se analizan los operadores de mutación que generan más mutantes potencialmente equivalentes, donde se observa que prácticamente en todos los casos, la equivalencia viene por el contexto donde se desarrolla la aplicación.

Cabe destacar que solo 3 de los 11 operadores que generan mutantes potencialmente equivalentes son de los que se han considerado como propios del lenguaje EPL de EsperTech. Tras analizar cada uno de ellos, se observa que las definiciones de los operadores de mutación: RAF, RJR, RNW, RSC, RTU, WBT y WTM, necesitarían mejorarse, para evitar las equivalencias. Esta mejora queda como trabajo futuro; en la que se analizará de nuevo la sintaxis de EPL de EsperTech y se buscarán nuevos casos de estudio para analizar la aplicación y mutación de estos operadores.

Finalmente se ha mostrado que el uso de la paralelización ayuda a paliar uno de los principales problemas de la prueba de mutaciones, el coste computacional que implica la ejecución de una vasta cantidad de mutantes contra los casos de prueba. Esta clase ayuda a sincronizar los hilos de ejecución (programa original y mutantes), haciendo que todos ellos se ejecuten una vez alcanzada la "barrera" en paralelo, reduciendo así el tiempo de ejecución. Cabe destacar que la máquina donde se quiera ejecutar MuEPL con paralelización, necesita una cantidad de memoria RAM suficiente para poder mantener todos los procesos en memoria a la hora de realizar la ejecución en paralelo del conjunto de programas mutantes y original. Esto dependerá del tamaño que ocupe la aplicación a probar; si no es suficiente (en caso extremo) se puede parar la máquina o no lograr el objetivo de reducción del tiempo de ejecución.

Capítulo 10

Conclusiones y Trabajo Futuro

En este último capítulo se describen las conclusiones obtenidas tras la realización de la investigación sobre la generación automática de casos de prueba (eventos) para lenguajes EPL y su validación a través de la prueba de mutaciones en consultas EPL de EsperTech. Así mismo se exponen las líneas de trabajo futuro relacionadas con el tema.

10.1. Contribuciones y Conclusiones

En el primer capítulo de esta disertación se establecieron los objetivos de esta tesis doctoral encaminados a la definición de un método de obtención de casos de prueba para aplicaciones que procesan y analizan grandes cantidades de datos (eventos) en tiempo real y su posterior validación aplicando la prueba de mutaciones en el lenguaje de programación EPL de EsperTech.

Para la consecución de estos objetivos se proponían un conjunto de subobjetivos, todos los cuales han sido alcanzados. A continuación se detallan las conclusiones obtenidas tras la consecución de cada uno de estos.

10.1.1. Estado del Arte

El primer objetivo que se planteó en esta tesis fue la recopilación del estado del arte de los estudios sobre el Internet de las Cosas (IoT), el procesamiento de eventos complejos (CEP) y las pruebas aplicadas, la expansión de los lenguajes de procesamiento de eventos, los generadores de eventos y plataformas IoT, la definición de operadores de mutación y su integración en herramientas de mutación y la aplicación de la prueba de mutaciones

en sistemas de tiempo real. Este objetivo se ha llevado a cabo mediante la realización de una extensa revisión bibliográfica (véase el capítulo 3).

En los estudios encontrados sobre IoT, se plantea cómo han de hacerse las pruebas en este tipo de aplicaciones, las diferencias con las pruebas realizadas a cualquier otra aplicación web o de escritorio y qué tipo de pruebas han de realizarse. Hay que destacar la importancia, en este tipo de aplicaciones, de las pruebas de seguridad. Se encuentran en la literatura guías y proyectos donde se hace hincapié en este tipo de pruebas.

En cuanto a los diferentes tipos de pruebas en aplicaciones CEP se encontraron diversos estudios donde se hablaba de los distintos tipos de prueba y su aplicación al sistema CEP: para comprobar si se cubrían todas las fases del sistema [83] o si se superaban ciertas pruebas de aceptación [84], para modelar y hacer pruebas de comportamiento funcionales [85], para testear de forma automática interfaces gráficas [86, 87], etc.

Otro de los puntos tratados en el estado del arte son los diferentes tipos de Lenguajes de Procesamiento de Eventos, destacando los estudios más relevantes de cada uno. Entre todos ellos se destacan las bondades de EPL de EsperTech, el cual tiene un carácter libre y es un EPL orientado a flujos, características que han sido relevantes para ser escogido como lenguaje de estudio en la presente tesis.

La mayoría de los generadores de casos de prueba (eventos) son para áreas muy específicas, no hay una gran variedad de este tipo de herramientas. Que, al igual que los generadores encontrados, pueden ser de carácter comercial o públicas y además permiten al usuario gestionar el tipo de actividad a monitorizar.

En cuanto a la aplicación de la prueba de mutaciones en lenguajes de programación, se han encontrado trabajos donde la aplican tanto en lenguajes de programación "tradicionales", como lenguajes de programación "nuevos" (estos últimos especialmente en publicaciones más recientes). La mayoría de los trabajos encontrados no solo incluyen la definición de los operadores de mutación del lenguaje analizado, sino que incluyen la integración de una herramienta, al igual que en esta tesis doctoral.

Los trabajos encontrados sobre la aplicación de la prueba de mutaciones en sistemas de tiempo real son aportaciones que estudian lenguajes tradicionales que han ido evolucionando y que, actualmente, se utilizan para implementar sistemas de tiempo real: Java, C, Ada; pero no son lenguajes diseñados para este tipo de sistemas.

Finalmente, entre los trabajos relacionados, se observa que todos están enfocados en hacer pruebas en las consultas. A pesar de la flexibilidad de configuración de algunos, los eventos han de declararse de forma manual. En el método que se propone son las

definiciones de los tipos de eventos las que son verificadas para conseguir de manera automática el conjunto de eventos para realizar las pruebas.

10.1.2. Proponer un método para generar de forma automática eventos para pruebas

En el capítulo 4 se introduce el método que automatiza la generación de eventos para realizar pruebas en sistemas que procesen eventos.

Este método permite generar eventos de cualquier tipo. Para ello, el usuario ha de seguir las etapas de las que se compone: definir la estructura del tipo de evento siguiendo la propuesta de especificación, validar la definición y una vez validada, generar los eventos a través del generador de eventos IoT-TEG (en los capítulos 5 y 6, se describe en detalle cada etapa).

10.1.3. Proponer una especificación para la definición de los tipos de eventos

La propuesta de especificación (véanse capítulo 5 y apéndice A) se ha desarrollado siguiendo las recomendaciones de Luckham [93], entre otras. Se ha utilizado un lenguaje fuertemente tipado (XML) y se ha tenido en cuenta que los tipos de datos de los atributos de los eventos pueden ser simples y complejos.

Gracias a la flexibilidad de la especificación desarrollada, esta permite definir una amplia variedad de tipos de eventos, algo que se ha verificado con más de 450 tipos de eventos reales pertenecientes a canales de la plataforma ThingSpeak y programas que gestionan eventos en tiempo real. Esto ayudó no solo a validar la especificación sino también a mejorarla y precisar sus valores y estructura.

10.1.4. Implementar una herramienta para generar casos de prueba para lenguajes procesadores de eventos

Para alcanzar este punto se ha desarrollado la herramienta IoT-TEG. Esta automatiza la generación de eventos de pruebas para las aplicaciones que procesan eventos: valida la definición del tipo de evento con el que se va a trabajar y, una vez validada, genera el número de eventos que requiere el programador para realizar pruebas (véase capítulo 6). Para que una definición de tipo de evento pueda ser validada ha de seguir la especificación propuesta en el capítulo 5 para definir tipos de eventos.

Esta herramienta se ha desarrollado siguiendo los puntos que sugieren Saboor y Rengasamy en el trabajo [92] para la realización de pruebas, lo que ha permitido que sea una herramienta flexible ya que se adapta a las necesidades del usuario y de las pruebas. IoT-TEG tiene un funcionamiento similar a los canales de información que incluyen las plataformas IoT, pero con las siguientes ventajas:

1. El usuario puede definir el número de eventos a generar sin límites.
2. El tipo de evento es definido por el usuario (luego el tipo de evento está enfocado a cubrir las necesidades del usuario).
3. La generación de eventos se puede realizar con valores específicos.
4. La obtención de los eventos es rápida (no hay que esperar a la fuente).

Adicionalmente, se ha comprobado el formato de salida de las plataformas IoT más populares, e IoT-TEG cubre los tres formatos de salida más comunes `{JSON, XML, CSV}`.

Gracias a las propiedades que se consideran en la propuesta de especificación, se pueden definir una vasta variedad de tipos de eventos, así como limitar y asignar valores específicos a los atributos del tipo de evento. En IoT-TEG se implementa una funcionalidad que permite generar valores según las consultas EPL de EsperTech que van a procesarlos. Esta funcionalidad se centra en las operaciones relacionales y lógicas que aparecen en las consultas, generando un conjunto de eventos que cumplen las restricciones que aparecen en ellas. Esta funcionalidad, junto con las propiedades de la especificación, permiten obtener un conjunto de casos de prueba de mayor calidad que si se generan los eventos de manera aleatoria y sin limitaciones.

IoT-TEG permite generar casos de prueba a aplicaciones IoT. Para ello el desarrollador encargado de esta labor deberá definir el tipo de evento así como el rango de sus atributos según el tipo de prueba a realizar. Esta definición del tipo de evento será la que determinará la cobertura de los valores de los atributos del tipo de evento, según se trate de prueba de estrés, pruebas de integración, etc.

10.1.5. Definir un conjunto de operadores de mutación adaptados a las características del lenguaje EPL de EsperTech

Con el objetivo de validar la herramienta de generación de eventos de prueba IoT-TEG, se aplica la prueba de mutaciones en aplicaciones que ejecutan consultas del lenguaje EPL de EsperTech. Para poder aplicar la prueba de mutaciones a programas escritos

en un determinado lenguaje necesitamos disponer de un conjunto de operadores de mutación especialmente adaptado a las características de dicho lenguaje [123]. Una de las aportaciones de esta tesis ha sido la definición de este conjunto de operadores de mutación, ya que anteriormente a su realización no se había aplicado la prueba de mutaciones al lenguaje EPL de EsperTech. Concretamente se han definido 34 operadores de mutación que modelan los fallos en que pueden incurrir los programadores al implementar una consulta EPL de EsperTech. Estos operadores se han clasificado en 4 categorías: operadores de expresiones de patrones, operadores de ventanas, operadores de reemplazo y operadores de inyecciones de ataque de SQL. Los trabajos [158, 159] describen los operadores definidos, al igual que se hace en el capítulo 7.

Como base se han utilizado los operadores de mutación definidos por Tuya y col. [135], en el cual se definen los operadores de mutación del lenguaje SQL. Un lenguaje muy similar a EPL de EsperTech que contempla 21 operadores de mutación, de los cuales algunos se han aplicado a EPL de EsperTech sin modificaciones, otros han tenido que adaptarse según la sintaxis de EPL de EsperTech y otros no se han considerado por no ajustarse a la sintaxis de EPL.

10.1.6. Implementación de una herramienta de generación y ejecución de mutantes EPL de EsperTech

Los operadores de mutación definidos para EPL de EsperTech han sido implementados e integrados en MuEPL, una herramienta de generación y ejecución automática de mutantes (véase el capítulo 8).

MuEPL automatiza la aplicación de la prueba de mutaciones a las aplicaciones Java que lanzan consultas EPL de EsperTech: captura, en el caso de que las consultas estén integradas en el código, las consultas EPL de EsperTech que van a ser mutadas; genera todos los mutantes de las consultas aplicando los operadores de mutación anteriormente citados; ejecuta los mutantes frente al conjunto de casos de prueba; y compara el comportamiento de los mutantes con el de la consulta original, produciendo una matriz de ejecución que permite saber si un mutante ha muerto o no con cada caso de prueba.

Esta herramienta incluye un sistema de ejecución que permite ejecutar en paralelo los diferentes mutantes generados y el programa original, consiguiendo paliar uno de los principales problemas de la prueba de mutaciones: el coste computacional.

10.1.7. Evaluar los resultados con diferentes casos de estudio

Finalmente, el séptimo objetivo, consistía en la evaluación de los resultados obtenidos tras la aplicación de la prueba de mutaciones en diferentes casos de estudio utilizando los eventos originales de cada uno de ellos y utilizando los eventos generados a través del método propuesto en la presente tesis. Por otro lado, a raíz de la implementación de MuEPL se han podido evaluar los tiempos de ejecución utilizando y sin utilizar ejecución paralela. Finalmente, tras los resultados obtenidos con los diferentes casos de estudio, se puede justificar la definición de los operadores de mutación para EPL de EsperTech (véase capítulo 7). Los resultados obtenidos tras las ejecuciones permiten establecer las siguientes conclusiones:

- El método propuesto para la generación automática de eventos permite, gracias a la especificación para la definición de tipos de eventos y a IoT-TEG, generar eventos de una amplia variedad de tipos de eventos.
- MuEPL permite aplicar la prueba de mutaciones en las consultas EPL de EsperTech de cualquier programa que procese eventos que esté implementado con el lenguaje de programación Java (se recuerda que las consultas EPL de EsperTech son lanzadas a través de su motor CEP escrito en Java, lenguaje de programación que lanza las consultas).
- Al comprobar la validez de los eventos generados a través del método propuesto usando la prueba de mutaciones, se destaca que en 7 de las 9 aplicaciones utilizadas, se matan al máximo de mutantes (teniendo en cuenta los mutantes potencialmente equivalentes). Algo que no se logra con todos los casos de estudio en los que se emplean los eventos originales del mismo. En las otras 2 aplicaciones (dentro del caso de estudio DENMEvaP), en una se obtiene un número de mutantes muertos muy similar que usando los eventos originales, y en la otra el número es inferior. Esto se debe a que las mutaciones están aplicadas dentro de patrones de eventos con condiciones muy específicas que dificultan localizarlas. Las pruebas realizadas con los eventos originales los localizan, pero recordamos que todos los casos de prueba originales del caso de estudio DENMEvaP, tienen valores específicos "forzados" por el programador (véase apéndice B.5).
- Con los eventos generados por IoT-TEG, utilizando la funcionalidad avanzada, se mata a un mayor número de mutantes. Algo lógico ya que se está consiguiendo que el conjunto de eventos que se genere por IoT-TEG sea de mayor calidad. Este conjunto de eventos cumplen las restricciones que aparecen en las consultas EPL de EsperTech que los van a procesar, luego es más probable que sean evaluados por las mismas, lo que ayudará a que se detecten las posibles mutaciones.

- Los tiempos de ejecución utilizando paralelización son considerablemente menores que los que se obtienen sin utilizarla. Lo que implica que el coste computacional de las ejecuciones se ve reducido paliando así este inconveniente de la prueba de mutaciones. Se recuerda que la máquina donde se quiera ejecutar MuEPL utilizando paralelización, tiene que contar con una cantidad de memoria RAM suficiente para poder mantener todos los procesos en memoria a la hora de realizar la ejecución en paralelo del conjunto de programas mutantes y original. Esto dependerá del tamaño que ocupe la aplicación a probar; si no es suficiente se puede parar la máquina o no lograr el objetivo de reducción del tiempo de ejecución.

- De los operadores de mutación definidos para EPL de EsperTech, aquellos considerados como los operadores propios del lenguaje, se pueden aplicar en la mayoría de las pruebas ya que aparecen frecuentemente en las consultas de este lenguaje. Considerando las 9 aplicaciones en las que se han realizado las pruebas; los operadores clasificados en la categoría "Operadores de expresiones de patrones" se aplican en 8. En estas aparecen al menos 3 de los 6 operadores, es decir, como mínimo un 50 % de los operadores de esta categoría se aplican cada vez que existe un patrón de eventos en la consulta. Los operadores clasificados en la categoría "Operadores de ventana" se aplican en 5 de las 9 aplicaciones, al igual que el operador RTU (encargado de reemplazar las unidades de tiempo de ventanas). Como mínimo un 50 % de los operadores de la categoría "Operadores de ventana" (al menos 2 de los 4 totales) se aplican cuando existe una ventana en la consulta. Finalmente, el operador IWR aparece en 7 de las 9 aplicaciones. Sin embargo, los operadores RRR_1 y RRR_2 no aparecen en ningún caso de estudio, bien porque el uso de las funciones `‘‘single row’’` donde se aplican no esté muy difundido entre los "nuevos" programadores, o que no sean utilizadas tan frecuentemente como para considerarlas dentro de los fallos comunes en este lenguaje de programación.

  Del análisis de mutantes equivalentes, se llega a la conclusión de que la mayoría de ellos son equivalentes según el contexto de la aplicación. Del conjunto de los 11 operadores que genera mutantes equivalentes, solo 3 están considerados dentro de los operadores propios del lenguaje de programación EPL de EsperTech. Tras analizar cada uno de ellos, se observa que las definiciones de los operadores de mutación: RAF, RJR, RNW, RSC, RTU, WBT y WTM, necesitarían mejorarse, para evitar las equivalencias. Esta mejora queda como trabajo futuro; en la que se analizará de nuevo la sintaxis de EPL de EsperTech y se buscarán nuevos casos de estudio para analizar la aplicación y mutación de estos operadores.

10.2. Líneas de Investigación Futuras

Una vez descritas las conclusiones obtenidas en esta tesis doctoral, se explican las líneas de investigación futuras que se abren con este trabajo.

10.2.1. Ampliar la funcionalidad de lectura de consultas

Tras analizar los resultados obtenidos, se observa que existe la necesidad de ampliar la funcionalidad de IoT-TEG, para que no solo se consideren las operaciones lógicas y relacionales que aparecen en las consultas EPL de EsperTech (en tipos de datos simples), sino que debe ampliarse a los tipos de datos complejos, así como a los patrones que han de cumplir los eventos. De esta forma el conjunto de casos de prueba que se obtenga gracias a esta funcionalidad será de mayor calidad.

Por otro lado, se quiere implementar esta funcionalidad de tal forma que los valores de los atributos de los eventos generados, vengan determinados según los valores límite que aparezcan en las operaciones relacionales. En una operación relacional *campo opRel valor* se llama valor límite al *valor* que aparece en la misma. Los valores del elemento *campo* en los eventos generados, serán *valor + 1*, *valor - 1*, *valor + 10*, *valor - 10* (siempre dentro de las restricciones propias de la definición del tipo de evento). De este modo se asegura que los valores de *campo* estén tanto por "encima" como por "debajo" del valor límite.

10.2.2. Añadir la generación secuencial en IoT-TEG

Los resultados que se obtienen con el generador IoT-TEG son aleatorios (dentro de un rango). Se quiere añadir una propiedad para los tipos simples: Integer, Float, Long, String, Alphanumeric, Date y Time con la que se puedan generar de forma secuencial dentro de un rango. En el caso del tipo Integer, se generarían valores secuenciales del 1 al 10 (incrementándose en una unidad) si el rango especificado fuera [1, 10], en el caso del tipo String se generarían valores secuenciales de la L a la R (siguiendo el orden alfabético) si el rango especificado fuera [L, R], en el caso del tipo Date se generarían valores secuenciales de 04/12/2017 al 28/04/2018 (incrementándose los días, meses y años según el calendario gregoriano) si el rango indicado fuese [04/12/2017, 28/04/2018], etc. Es posible que para ciertas pruebas se necesite que los valores sigan una secuencia y esta propiedad ayudaría a la generación de estos eventos.

10.2.3. Generación de varios tipos de eventos en IoT-TEG

Actualmente IoT-TEG genera el número de eventos que el usuario requiera de un tipo de evento. En los casos de estudio Domótica 9.3.4 y DENMEvaP 9.3.5, se procesaban eventos de distintos tipos. En el caso de estudio de Domótica, los eventos venían de diferentes canales IoT de la plataforma ThingSpeak y el programa original extraía de cada evento los valores de los atributos relevantes. En el caso de estudio DENMEvaP, los eventos de los casos de prueba fueron preparados (véase sección B.5); según el conjunto de casos de prueba estos venían de uno o más dispositivos, que finalmente se almacenaban en un único fichero. Al igual que en Domótica, del conjunto de eventos procesados se extraen los valores de los atributos relevantes. Para paliar este inconveniente, se creó un tipo de evento que unificara y adaptaba los tipos de eventos involucrados según la prueba a realizar. Pero hay que tener en cuenta que en un canal de eventos pueden circular diferentes tipos de evento. Así que si el usuario quiere generar diferentes cantidades de eventos de prueba de varios tipos de evento, de forma aleatoria, se necesita que IoT-TEG lea varias definiciones de tipos de eventos y genere un conjunto aleatorio de cada uno de ellos según las cantidades especificadas. Esto hay que incluirlo en IoT-TEG ya que podría resultar interesante para hacer pruebas en un sistema que procese distintos tipos de evento.

10.2.4. Añadir la generación condicional en IoT-TEG

Como IoT-TEG ya contiene una generación de dependencia gracias al atributo `dependence`, sería interesante añadir la opción de generación condicional. Esta generación condicional de eventos consistiría en, por un lado, no volver a usar el valor de un atributo de evento si este ya ha sido usado. Otra opción sería, en el caso de generación de varios tipos de evento que comparten atributo, si un valor de un atributo de evento ha sido generado en un tipo de evento, volverlo a usar en el siguiente tipo de evento. Esta generación condicional de valores se implementará según el elemento condicional donde se aplique (entre tipos de evento, atributos de evento, etc).

10.2.5. Generación de eventos distribuidos en el tiempo

IoT-TEG es capaz de generar eventos para diferentes tipos de prueba dada la definición de un tipo de evento. Esto correspondería a simular el lanzamiento de eventos por parte de un dispositivo en un canal IoT. Para poder simular un canal IoT de distintos dispositivos durante un largo periodo de tiempo, además de incluir la generación de varios tipos de eventos, se debe incluir la posibilidad de programar la generación de eventos distribuidos

en el tiempo. Es decir, añadir la posibilidad de que IoT-TEG, cada T milisegundos, segundos, minutos..., genere N eventos. Siendo T la unidad de tiempo estipulada y N el número de eventos que indique el usuario. Esta generación de eventos ayudaría a simular un canal real durante un periodo de tiempo en el que están conectados diversos dispositivos. Así mismo habría que ofrecer un mecanismo de conexión que permitirá saber si se han generado o no nuevos eventos por parte de IoT-TEG.

10.2.6. Desarrollar un componente de validación de la salida

IoT-TEG genera eventos de prueba para cualquier programa que procese eventos en cualquiera de los formatos considerados en la presente tesis `XML`, `JSON` y `CSV`. La salida de la aplicación en cuestión, validará el correcto funcionamiento del programa según el flujo de eventos procesado. Esta comprobación se implementará a través de un *componente de validación de la salida* en el que el usuario podrá indicar qué actividades o eventos complejos son los esperados según los eventos de prueba generados. De esta forma se podrá comprobar si los eventos de prueba generados por IoT-TEG verifican el correcto funcionamiento de las distintas partes del software a probar.

10.2.7. Conexión de MuEPL con la herramienta GAmera

GAmera [166, 167] es una herramienta de generación de mutantes en la que se ha basado MuEPL para su diseño. GAmera está basada en la técnica de mutación evolutiva [168], la cual se ha demostrado que reduce el coste computacional de la prueba de mutaciones al generar un subconjunto de los mutantes totales, seleccionado por un Algoritmo Genético. Dado que MuEPL está diseñada basándose en esta herramienta, y que el Algoritmo Genético de GAmera está implementado de tal forma que es muy sencillo utilizarlo con cualquier lenguaje de programación, se podría conectar MuEPL con GAmera con el fin de generar un subconjunto de los mutantes totales, paliando aún más el coste computacional.

10.2.8. Mejorar la definición de algunos operadores de mutación

Tras analizar los operadores de mutación que generan mutantes equivalentes, se observa que hay que mejorar la definición de los operadores: RAF, RJR, RNW, RSC, RTU, WBT y WTM para evitar las equivalencias. Para lograrlo se analizará de nuevo la sintaxis de EPL de EsperTech y se buscarán nuevos casos de estudio para analizar la aplicación y mutación de estos operadores. El resto de operadores de mutación (en los casos de estudio utilizados en la presente tesis) genera alrededor, o menos, de un 5 % de mutantes equivalentes.

10.2.9. Mejorar el componente Capturador de MuEPL

Tal y como se ha comentado en la sección 8.2.1, para el desarrollo de MuEPL se han modificado las bibliotecas del propio EPL de EsperTech. Esto supone que el programador, en vez de usar las originales, ha utilizar las ofrecidas por MuEPL. Esta mejora permitirá la compatibilidad entre MuEPL y las bibliotecas de EsperTech ahorrando el trabajo extra al programador y evitando posibles problemas de actualización.

10.3. Publicaciones

10.3.1. Publicaciones Internacionales

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, Juan José e Inmaculada Medina-Bulo. IoT-TEG: Event Generator System. En: *Journal of Systems and Software JSS. Special Issue on Software Reliability Engineering* (En primera revisión para ser publicado).
- Lorena Gutiérrez-Madroñal. Method to Test Internet of Things Systems. En: *3dr Spanish-German Symposium on Applied Computer Science SGSOACS 2016. Cádiz, Spain.*
- Lorena Gutiérrez-Madroñal. Mutation Testing in Event Programming Language. En: *7th European Symposium on Computational Intelligence and Mathematics ESCIM 2015 Cádiz, Spain.*
- Lorena Gutiérrez-Madroñal. Applying Mutation Testing to Event Processing Language. En: *11th International Summer School on Training And Research On Testing, TAROT 2015 Cádiz, Spain.*
- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, Juan José e Inmaculada Medina-Bulo. Mutation Testing: Guideline and Mutation Operator Classification. En: *ICCGI 2014, The Ninth International Multi-Conference on Computing in the Global Information Technology*, pags. 171-179.
- Lorena Gutiérrez-Madroñal. Mutation Testing and Event Processing Language (EPL). En: *Queen's Graduate Computing Society Conference (QGCSC 2012), Kingston, Ontario, Canadá.*
- Lorena Gutiérrez-Madroñal, H. Shahriar, M. Zulkernine, Domínguez-Jiménez, J.J. e Inmaculada Medina-Bulo. En: *23rd IEEE International Symposium on Software Reliability Engineering ISSRE 2012, Dallas, Texas, Estados Unidos de América.*

10.3.2. Publicaciones Nacionales

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, Juan José e Inmaculada Medina-Bulo. Generación automática de eventos de prueba para sistemas de IoT. En: *XXI Jornadas de Ingeniería del Software y Bases de Datos (JISBD - CEDI 2016), Salamanca, Spain.*

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, J.J. e Inmaculada Medina-Bulo. Prueba de mutaciones sobre consultas de procesamiento de eventos en aplicaciones en tiempo real. En: *XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD - SISTEDES 2012), Almería, Spain.*

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, J.J. e Inmaculada Medina-Bulo. Análisis del lenguaje EPL para la prueba de mutaciones. En: *III Jornadas Predoctorales de la Escuela Superior de Ingeniería, Cádiz, Andalucía, España*, 2011.

- Antonia Estero-Botaro, Domínguez-Jiménez, J.J., Lorena Gutiérrez-Madroñal e Inmaculada Medina-Bulo. Evaluación de la calidad de los mutantes en la prueba de mutaciones. En: *Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos, A Coruña, Spain*, 2011.

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, J.J. e Inmaculada Medina-Bulo. Estudio preliminar de la calidad de los mutantes. En: *II Jornadas Predoctorales de la Escuela Superior de Ingeniería, Cádiz, Andalucía, España*, 2010.

- Inmaculada Medina-Bulo, Lorena Gutiérrez-Madroñal y Domínguez-Jiménez, J.J. Propuesta de optimización en la prueba de mutaciones en Java. En: *V Taller sobre pruebas en Ingeniería del Software (PRIS 2010), Valencia, Comunidad Valenciana, España.*

- Antonia Estero-Botaro, Inmaculada Medina-Bulo, Domínguez-Jiménez, J.J. y Lorena Gutiérrez-Madroñal. GAmera: una herramienta para la generación y selección mediante algoritmos genéticos de mutantes WS-BPEL. En: *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), San Sebastián, Spain.*

Capítulo 11

Conclusions and Future Work

This chapter is the English translation for chapter 10, where the accomplished goals, contributions and conclusions from this thesis are described. Likewise, the future work lines derived from the current research are exposed.

11.1. Contributions and Conclusions

In the first chapter of this dissertation, the goals of this thesis were exposed, leading to the definition of a methodology to obtain test cases for IoT applications which process and analyse a vast amount of information in real time, and the mutation testing application to EPL programming language.

To achieve these goals, a division of this main goal into smaller and simpler goals was executed, and completed successfully afterwards. The conclusions obtained by the achievement of these goals are now exposed.

11.1.1. State of the art

The first goal presented in this document was the obtaining of information in the following subjects:

- The state of the art of studies about IoT testing, CEP testing, taking a deeper look into EPL and its expansion.
- Research about the existing event generators.
- Studies about different programming languages and real time systems.

All of these were accomplished by an exhaustive literature review (chapter 3).

The IoT studies discuss how tests have to be done, the differences with the web or desktop tests and the kind of tests that should be done. The security tests are very important in this kind of applications, there are guides and projects where they are presented.

With reference to the tests in CEP applications, multiple studies which used the different tests in CEP systems in order to obtain different information about specific environments of this type of systems were found. Most of these tests are really specific depending on the environment of the CEP application.

Another relevant point in the state of the art was to review the different Event Processing Languages types, highlight the most relevant studies in each case. Among all of them the Esper EPL goodness are highlighted. Its free license and the fact that is a stream oriented EPL have been determinant to be chosen for this thesis.

Most of the test cases generators are intended for specific fields, there is not a wide variety of this type of tool because the IoT platforms are much more accessible to regular users. These IoT platforms, like the previously commented generators, can be public or commercial, letting the user to monitor whatever activity they want.

Regarding to the application of mutation testing in different programming languages, different works about “traditional” and “new” programming languages were found (the latter especially in recent publications). Most of these works not only include the definition of the mutation operator in the given language, but they also include a tool were these definitions are implemented, as it happens in this thesis.

The different works found about mutation testing in real time systems are contributions that study traditional programming languages like: Java, C, and Ada. Even though these languages are not specifically designed for these types of systems, they have evolved in order to be used to implement them.

Finally, the found related works are focused on testing EPL queries. Despite of their flexibility in terms of configuration, the events have to be defined manually. The proposed method validates the event type definitions in order to automatically generate the test suite (events).

11.1.2. Proposing a method to automatically generate test events

The chapter 4 introduces the method which automates the event generation to test event processing systems.

This method allows to generate events of any event type, these steps must be followed:

1. Define the event type structure using the proposed specification.
2. Validate the event type definition.
3. Generate the events, according to the validated event type definition, running the event generator IoT-TEG.

Each phase is described in detail in the chapters 5 and 6.

11.1.3. Proposed specification to define event types

The proposed specification (see chapter 5 and appendix A) has been created following Luckham's recommendations [93] (among others): it has been used a strongly typed language (XML) and it has been taken into account the data types of event attributes (complex and simple).

Thanks to the flexibility of this specification, a comprehensive variety of event types can be defined. More than 450 event types from ThingSpeak platform were tested using the specification, as well as programs which process, in real time, events. That tests helped not only to validate the specification, but also to improve it.

11.1.4. Tool implementation to generate event processing languages test cases

IoT-TEG automates the test events generation for event processing programs:

1. IoT-TEG validates the event type definition.
2. IoT-TEG generates the required number of events to test (see chapter 6).

The event type definition must follow the proposed specification to define valid event types (see chapter 5).

IoT-TEG has been created following Saboor and Rengasamy recommendations [92], as a consequence, IoT-TEG can be adapted to the user and tests requirements. This tool works similar to the canals IoT platforms but it has the following advantages:

- The user can define the number of event to generate (no limits).
- Given that the event type is defined by the user, it is focused to cover the user necessities.

- There is not need to wait for the source of the events, the event obtainment is faster.

The event outputs formats from the populars IoT platforms have been checked, the three common ones, `{JSON, XML, CSV}`, can be used in IoT-TEG.

The properties of the proposed specification help not only to define a large variety of event types, but also to limit and assign specific values to the event type attributes. There is a funtionality which enables the generation of events with values according to the EPL queries which will process them. This functionality covers the relational and logical operations restrictions from EPL queries, and the generated events meet these restrictions. The specification properties and this query functionality create test cases with higher quality.

Only the authors of the IoT tool know the range of values, so an accurate event type definition can be done by them to test the IoT application. The more accurate event type definition, the better coverage of values. The test event generation depend on the type of test to do, so the tester (the programmer or a member of the development group) will define the event type and the range of values of its attributes according to the test. So the full coverage of event attribute values depends on the tester event type definition.

11.1.5. Defining a mutation operator set adapted to EPL characteristics

In order to apply the mutation testing to programs implemented in a specific language, a set of mutation operators adapted the characteristics of the given language is needed [123]. One of the contributions of this thesis is the definition of this set, given that the mutation testing has not been applied to EPL before. Specifically, 34 mutation operators which model the most common mistakes a programmer can make when implementing an EPL query have been defined. These operators have been classified in 4 categories:

1. Window operators
2. Replacement operators
3. Injection attacks operators
4. Pattern expression operators

The works [158, 159] describe the defined operators, as it has been done in chapter 7.

The mutation operators defined by Tuya and col[82] have been used as a basis. In this work the mutation operators for the SQL language, a very similar language to EPL, are defined. Of the 21 mutation operators defined, some of them have been applied to EPL without modifications, others needed some adaptation to EPL syntax and another group of them did not apply.

11.1.6. EPL mutants generation and execution tool implementation

The defined EPL mutation operators are implemented and integrated in MuEPL. MuEPL is an automatic mutant generation and execution tool (see chapter 8).

MuEPL automates the mutation testing to Java programs which run EPL queries. MuEPL captures the EPL queries when they are integrated in the code, generates the mutants from the queries applying the mutation operators, executes them against the test suite and compares the mutant outputs with the original one (thanks to the execution matrix generated if a mutant is killed or not it can be known).

An execution system is included in this tool which uses Java `CyclicBarrier` class. The Java `CyclicBarrier` class allows parallel executions of the generated mutants and the original program. The application of this class palliates one of the main problems of the mutation testing: computational cost.

11.1.7. Evaluation of case studies results

The last objective was the evaluation of the obtained results after the mutation testing application to different cases studies. The original events and generated events have been used (using the proposed method of this thesis). On the other hand, the execution times using and not using the Java `CyclicBarrier` class was checked. Moreover, thanks to the obtained results, the EPL mutant operators (chapter 7) can be justified. The evaluation of the results allows us to establish the following conclusions:

- The proposed method to automatically generate test events allows to generate a huge variety of event types thanks to its components: the specification to define event types and IoT-TEG.
- Mutation testing can be applied using MuEPL to any EPL queries from any Java implemented event processing program.
- Using the proposed method to generate events and applying the mutation testing afterwards, the maximum number of killed mutants is achieved in 7 of the 9 applications; this is not always get with the original events. One of the other 2 applications

the number of killed mutants is similar to the obtained using the original ones. And with the last one the number is lower. This is because the mutations are applied to event patterns with specific conditions. The original events localize the mutations, but it has to be taken into account that the original events were "forced" by the programmer to have specific values (see appendix B.5).

- Using the advanced functionality, the IoT-TEG generated events, the more query conditions they meet, the more likely are the mutants to be killed. The test suite has a highlight quality.

- `CyclicBarrier` execution times are significantly lower than `Non-CyclicBarrier` execution times. That means that the execution computational cost is reduced, so this mutation testing drawback is mitigated.

- Of the defined EPL mutation operators, those considered as typical of the programming language, can be applied to the majority of the test because their frequency in queries. Talking about the 9 applications where the tests have been done, the "Patterns operators" can be applied to 8 of them, where at least 3 of the 6 operators appears. The "Windows operators", like the RTU operator, can be applied to 5 of the 9 applications, where at least a 50 % of the "Windows operators" operators appears. Finally, the IWR operator can be applied to 7 of the 9 applications. The RRR_1 and RRR_2 operators do not appear, this could be because the "single row functions" are not very popular among the new programmers, or they are not frequently used to be considered like a common programming errors.

  The found equivalent mutants are because the context of the application. Only 3 of the 11 operators which generate equivalents, are considered specific of EPL EsperTech.

11.2. Future Research Lines

In addition to the contributions made on this thesis, several future research lines remain open as summarized in the following lines.

11.2.1. Extending the queries reading functionality

After analysing the obtained results, it looks necessary to extend the IoT-TEG queries reading functionality. The functionality has to include logical and relational operations where complex data types are involved and also patterns which have to be met by the events. The obtained test suite will have better quality thanks to this functionality.

On the other hand, this functionality will be implemented in other way. The limit values will be the ones which determine the event attributes values. Limit value is the *value* in a relational operation *field relOp value*. The *field* values in the generated events will be *value + 1*, *value - 1*, *value + 10*, *value - 10* (meeting the event type definition restrictions). The *field* values will be up and down the limit value.

11.2.2. Adding a sequential generation

The obtained results by IoT-TEG are random (within a range); given that sequential values for certain tests could be necessary, a property for the simple data types: Integer, Float, Long, String, Alphanumeric, Date and Time to be sequentially generated, wants to be included. For instance, the Integer data type will be sequentially generated from 1 to 10 increasing one unit if the range is [1, 10]. The String data type will be sequentially generated from L to R following the alphabetical order if the range is [L, R]. The Date data type will be sequentially generated from 12/04/2017 to 04/28/2018 increasing day by day according to the Gregorian calendar, etc. This sequential generation of values could help to do some tests where sequential values are needed.

11.2.3. Adding multiple event type generations

IoT-TEG generates events given an event type definition. The Domotic 9.3.4 and DEN-MEvaP 9.3.5 case studies process events from several event types. Due to IoT-TEG is able to generate events according to an event type definition, we decided to make the following solution; to unify the relevant attributes of the different involved event types and define a new event type. The event type definition will be upgraded in order to consider multiple definitions of event types, without the necessity of joining all the relevant event attributes in a single definition. This means that IoT-TEG will read individually different event type definitions and will generate a mix of events from each one, but according to their required number.

11.2.4. Adding the conditional generation

The conditional generation is interesting to be include because the following situations could be covered; if an event attribute value is unique (if it has been used, it can not be used again) or, in a multiple event type generation case, if the event types share an event attribute, use the same value in both. This conditioned value generation should be implemented differently depending on the conditioned element (event types, event attributes, etc).

11.2.5. Time distribution between events generation

IoT-TEG is able to generate events to do different kind of tests according to an event type definition. This would be like having one device sending events into an IoT channel. In order to simulate a channel with several devices connected to it over a long period of time, the events sent by IoT-TEG have to be distributed over time. IoT-TEG has to generate N events every T milliseconds, seconds, minutes..., where N is the number of events and T the time unit. Additionally, a connection mechanism has to be implemented to notify that IoT-TEG has generated new events of multiple types.

11.2.6. Output validation component implementation

IoT-TEG generates test events to any event processing program which process `XML`, `JSON` or `CSV` as event format. The output of the application under test, will determine if the program works correctly according to the input event stream. An *output validation component* will be implemented to check the outputs. The expected activities and complex event will be indicated by the user. The correct functioning of each software component will be verified thanks the IoT-TEG test generated events and the *output validation component*.

11.2.7. MuEPL and GAmera connection

GAmera [166, 167] is a mutant generation tool in which MuEPL has been based on to be implemented. GAmera is based on the evolutionary mutation technique [168], which reduces the computational cost involved in mutation testing. GAmera generates a subset of mutants which are selected by a Genetic Algorithm. Due to MuEPL is based on this tool and its Genetic Algorithm was implemented to be easily used with any programming language, MuEPL could be connected to GAmera in order to generate a subset of mutants and reduce more the computational cost.

11.2.8. Improving some mutation operator definitions

After analysing the mutation operators which generate equivalent mutants, the following operator definitions need to be improve in order to avoid the equivalences: RAF, RJR, RNW, RSC, RTU, WBT and WTM. To achieve it, the EsperTech EPL syntax will be studied again, and new studies cases will be used to analyse the operators application and mutation. The others mutation operators (according to the studies cases in this thesis) generate around, or less, a 5 % equivalent mutants.

11.2.9. Improving the Capturing component

The Capturing component (see section 8.2.1) was implemented modifying the EsperTech EPL libraries. As a consequence, the programmer instead of using the original EsperTech libraries, they have to use the ones given with MuEPL. This will be changed in the future, allowing the compatibility between MuEPL and the original EsperTech libraries, saving the extra work and the updates.

11.3. Publications

11.3.1. International Publications

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, Juan José and Inmaculada Medina-Bulo. IoT-TEG: Event Generator System. In: *Journal of Systems and Software JSS. Special Issue on Software Reliability Engineering* (Under first revision to be published).
- Lorena Gutiérrez-Madroñal. Method to Test Internet of Things Systems. In: *3dr Spanish-German Symposium on Applied Computer Science SGSOACS 2016. Cádiz, Spain.*
- Lorena Gutiérrez-Madroñal. Mutation Testing in Event Programming Language. In: *7th European Symposium on Computational Intelligence and Mathematics ES-CIM 2015 Cádiz, Spain.*
- Lorena Gutiérrez-Madroñal. Applying Mutation Testing to Event Processing Language. In: *11th International Summer School on Training And Research On Testing, TAROT 2015 Cádiz, Spain.*
- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, Juan José and Inmaculada Medina-Bulo. Mutation Testing: Guideline and Mutation Operator Classification. In: *ICC-GI 2014, The Ninth International Multi-Conference on Computing in the Global Information Technology*, pags. 171-179.
- Lorena Gutiérrez-Madroñal. Mutation Testing and Event Processing Language (EPL). In: *Queen's Graduate Computing Society Conference (QGCSC 2012), Kingston, Ontario, Canadá.*
- Lorena Gutiérrez-Madroñal, H. Shahriar, M. Zulkernine, Domínguez-Jiménez, J.J. and Inmaculada Medina-Bulo. In: *23rd IEEE International Symposium on Software Reliability Engineering ISSRE 2012, Dallas, Texas, Estados Unidos de América.*

11.3.2. National Publications

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, Juan José and Inmaculada Medina-Bulo. Generación automática de eventos de prueba para sistemas de IoT. In: *XXI Jornadas de Ingeniería del Software y Bases de Datos (JISBD - CEDI 2016), Salamanca, Spain.*

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, J.J. and Inmaculada Medina-Bulo. Prueba de mutaciones sobre consultas de procesamiento de eventos en aplicaciones en tiempo real. In: *XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD - SISTEDES 2012), Almería, Spain.*

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, J.J. and Inmaculada Medina-Bulo. Análisis del lenguaje EPL para la prueba de mutaciones. In: *III Jornadas Predoctorales de la Escuela Superior de Ingeniería, Cádiz, Andalucía, España*, 2011.

- Antonia Estero-Botaro, Domínguez-Jiménez, J.J., Lorena Gutiérrez-Madroñal and Inmaculada Medina-Bulo. Evaluación de la calidad de los mutantes en la prueba de mutaciones. In: *Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos, A Coruña, Spain*, 2011.

- Lorena Gutiérrez-Madroñal, Domínguez-Jiménez, J.J. and Inmaculada Medina-Bulo. Estudio preliminar de la calidad de los mutantes. In: *II Jornadas Predoctorales de la Escuela Superior de Ingeniería, Cádiz, Andalucía, España*, 2010.

- Inmaculada Medina-Bulo, Lorena Gutiérrez-Madroñal y Domínguez-Jiménez, J.J. Propuesta de optimización en la prueba de mutaciones en Java. In: *V Taller sobre pruebas en Ingeniería del Software (PRIS 2010), Valencia, Comunidad Valenciana, España.*

- Antonia Estero-Botaro, Inmaculada Medina-Bulo, Domínguez-Jiménez, J.J. y Lorena Gutiérrez-Madroñal. GAmera: una herramienta para la generación y selección mediante algoritmos genéticos de mutantes WS-BPEL. In: *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), San Sebastián, Spain.*

Apéndice A

XML Schema de la Propuesta de Especificación para la Definición de Tipos de Eventos

A.1. XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="event" type="EventType"/>
 <xs:complexType name="EventType">
  <xs:sequence>
    <xs:element name="block" type="BlockType"/>
  </xs:sequence>
 <xs:attribute name="name" type="xs:string" use="required"/>
 </xs:complexType>

 <xs:complexType name="BlockType">
  <xs:all>
    <xs:element name="field" type="IntegerFieldType"/>
    <xs:element name="field" type="FloatFieldType"/>
    <xs:element name="field" type="LongFieldType"/>
    <xs:element name="field" type="StringFieldType"/>
    <xs:element name="field" type="AlphanumericFieldType"/>
    <xs:element name="field" type="BooleanFieldType"/>
    <xs:element name="field" type="TimeFieldType"/>
    <xs:element name="field" type="DateFieldType"/>
    <xs:element name="optionalfields" type="OptfieldType"/>
  </xs:all>
```

```
<xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
<xs:attribute name="repeat" type="xs:integer" use="required" minOccurs="1"
   maxOccurs="1"/>
<xs:attribute name="value" type="xs:string" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="OptfieldType">
 <xs:all>
   <xs:element name="field" type="IntegerFieldType"/>
   <xs:element name="field" type="FloatFieldType"/>
   <xs:element name="field" type="LongFieldType"/>
   <xs:element name="field" type="StringFieldType"/>
   <xs:element name="field" type="AlphanumericFieldType"/>
   <xs:element name="field" type="BooleanFieldType"/>
   <xs:element name="field" type="TimeFieldType"/>
   <xs:element name="field" type="DateFieldType"/>
   <xs:element name="field" type="ComplexFieldType"/>
 </xs:all>
<xs:attribute name="mandatory" type="xs:boolean" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="IntegerFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="Integer" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="min" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="max" type="xs:integer" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="FloatFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="Float" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:float" maxOccurs="1"/>
 <xs:attribute name="precision" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="min" type="xs:float" maxOccurs="1"/>
 <xs:attribute name="max" type="xs:float" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="LongFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="Long" use="required" maxOccurs="1"/>
```

```
 <xs:attribute name="value" type="xs:long" maxOccurs="1"/>
 <xs:attribute name="min" type="xs:long" maxOccurs="1"/>
 <xs:attribute name="max" type="xs:long" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="StringFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="String" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:string" maxOccurs="1"/>
 <xs:attribute name="length" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="case" value="low" maxOccurs="1"/>
 <xs:attribute name="endcharacter" type="xs:char" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="AlphanumericFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="Alphanumeric" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:string" maxOccurs="1"/>
 <xs:attribute name="length" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="case" value="low" maxOccurs="1"/>
 <xs:attribute name="endcharacter" type="xs:char" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="BooleanFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="Alphanumeric" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:boolean" maxOccurs="1"/>
 <xs:attribute name="isnumeric" type="xs:boolean" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="TimeFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
 <xs:attribute name="type" value="Time" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:long" maxOccurs="1"/>
 <xs:attribute name="format" type="xs:string" use="required" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="DateFieldType">
 <xs:attribute name="name" type="xs:string" use="required" maxOccurs="1"/>
 <xs:attribute name="quotes" type="xs:boolean" use="required" maxOccurs="1"/>
```

```
 <xs:attribute name="type" value="Date" use="required" maxOccurs="1"/>
 <xs:attribute name="value" type="xs:long" maxOccurs="1"/>
 <xs:attribute name="format" type="xs:string" use="required" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="ComplexFieldType">
 <xs:all>
   <xs:element name="field" type="IntegerFieldType"/>
   <xs:element name="field" type="FloatFieldType"/>
   <xs:element name="field" type="LongFieldType"/>
   <xs:element name="field" type="StringFieldType"/>
   <xs:element name="field" type="AlphanumericFieldType"/>
   <xs:element name="field" type="BooleanFieldType"/>
   <xs:element name="field" type="TimeFieldType"/>
   <xs:element name="field" type="DateFieldType"/>
   <xs:element name="attribute" type="IntegerAttType"/>
   <xs:element name="attribute" type="FloatAttType"/>
   <xs:element name="attribute" type="LongAttType"/>
   <xs:element name="attribute" type="StringAttType"/>
   <xs:element name="attribute" type="AlphanumericAttType"/>
   <xs:element name="attribute" type="BooleanAttType"/>
   <xs:element name="attribute" type="TimeAttType"/>
   <xs:element name="attribute" type="DateAttType"/>
 </xs:all>
<xs:attribute name="chooseone" type="xs:boolean" maxOccurs="1"/>
<xs:attribute name="dependence" type="xs:boolean" maxOccurs="1"/>
<xs:attribute name="dependence" type="xs:string" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="IntegerAttType">
 <xs:attribute name="value" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="min" type="xs:integer" maxOccurs="1"/>
 <xs:attribute name="max" type="xs:integer" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="FloatAttType">
 <xs:attribute name="value" type="xs:float" maxOccurs="1"/>
 <xs:attribute name="min" type="xs:float" maxOccurs="1"/>
 <xs:attribute name="max" type="xs:float" maxOccurs="1"/>
 <xs:attribute name="precision" type="xs:integer" maxOccurs="1"/>
</xs:complexType>

<xs:complexType name="LongAttType">
 <xs:attribute name="value" type="xs:long" maxOccurs="1"/>
```

```
  <xs:attribute name="min" type="xs:long" maxOccurs="1"/>
  <xs:attribute name="max" type="xs:long" maxOccurs="1"/>
 </xs:complexType>

 <xs:complexType name="StringAttType">
  <xs:attribute name="value" type="xs:string" maxOccurs="1"/>
  <xs:attribute name="length" type="xs:integer" maxOccurs="1"/>
  <xs:attribute name="case" value="low" maxOccurs="1"/>
  <xs:attribute name="endcharacter" type="xs:char" maxOccurs="1"/>
 </xs:complexType>

 <xs:complexType name="AlphanumericAttType">
  <xs:attribute name="value" type="xs:string" maxOccurs="1"/>
  <xs:attribute name="length" type="xs:integer" maxOccurs="1"/>
  <xs:attribute name="case" value="low" maxOccurs="1"/>
  <xs:attribute name="endcharacter" type="xs:char" maxOccurs="1"/>
 </xs:complexType>

 <xs:complexType name="BooleanAttType">
  <xs:attribute name="value" type="xs:boolean" maxOccurs="1"/>
  <xs:attribute name="isnumeric" type="xs:boolean" maxOccurs="1"/>
 </xs:complexType>

 <xs:complexType name="TimeAttType">
  <xs:attribute name="value" type="xs:long" maxOccurs="1"/>
  <xs:attribute name="format" type="xs:string" use="required" maxOccurs="1"/>
 </xs:complexType>

 <xs:complexType name="DateAttType">
  <xs:attribute name="value" type="xs:long" maxOccurs="1"/>
  <xs:attribute name="format" type="xs:string" use="required" maxOccurs="1"/>
 </xs:complexType>
</xs:schema>
```

Código A.1: XML Schema de la Propuesta de Especificación para la Definición de Tipos de Eventos

Apéndice B

Casos de Estudio

En este apéndice se presentan y describen los casos de estudio en los que se aplica la herramienta MuEPL descrita en el capítulo 8 utilizando los “eventos originales”[1] de la propia herramienta y los generados a través de la metodología propuesta en la presente tesis, véase capítulo 4. Los casos de estudio analizados van desde programas distribuidos por la propia compañía EsperTech, a aplicaciones elaboradas para la docencia e investigación. Además se indican las diferentes modificaciones o adaptaciones que han sido implementadas para poder realizar las pruebas.

B.1. Caso de Estudio - Self-Service Terminal

Dentro de las distribuciones de la compañía EsperTech [9], ofrecen diversos casos de estudio para comprobar la potencia y versatibilidad de EPL. Uno de los programas de ejemplos al que se le ha aplicado MuEPL es el denominado “Self-Service Terminal” [160]. El ejemplo está basado en J2EE y simula un sistema de auto servicio de un aeropuerto, el cual obtiene un gran cantidad de eventos de las terminales que están conectadas al mismo. La media de eventos obtenidos por segundo esta sobre 500. Algunos eventos indican situaciones anormales como “paper low” (poco papel), “terminal out of order” (terminal fuera de servicio), etc. Otros eventos contemplan actividades de los clientes: a la hora de hacer “check in”, imprimir las tarjetas de embarque, etc.

Cada terminal conectada al sistema de auto servicio puede publicar cualquiera de los siguientes seis eventos:

1. **Checkin**: Indica que un cliente ha comenzado un diálogo para un check-in.

[1] Se llaman “eventos originales“ a aquellos que originalmente utiliza el caso de estudio; bien sea eventos generados por un generador de eventos propio, canales de eventos, ficheros con eventos, etc.

2. **Cancelled**: Indica que un cliente ha cancelado el diálogo del check-in.
3. **Completed**: Indica que un cliente ha completado el diálogo del check-in.
4. **OutOfOrder**: Indica que una terminal ha detectado un problema de hardware.
5. **LowPaper**: Indica que la terminal se está quedando sin papel.
6. **Status**: Indica el estado de la terminal. Este es publicado cada minuto independientemente de la actividad, es un "sistema de latidos" de la terminal.

Todos los eventos proporcionan información sobre la terminal que publicó el evento y una marca de tiempo (*timestamp*). La información de la terminal está soportada en la propiedad "term", la cual proporciona el id de la terminal. A pesar de que todos los eventos ofrecen una información similar, estos se han modelado como subtipos de la clase "BaseTerminalEvent", lo que permite tratar todos los eventos de las terminales de manera polimórfica y simplificar las consultas.

Todas las terminales conectadas al sistema publican su "Status" cada minuto. En situaciones normales, el "Status" indicaría que la terminal está activa y *online*. La ausencia de este evento podría indicar que una terminal está *offline* por alguna razón que necesita ser investigada.

Como se ha comentado previamente, en este caso de estudio se utilizan unos eventos basados en la clase "BaseTerminalEvent", estos eventos son obtenidos por el generador de eventos que el propio caso de estudio tiene implementado. Dado que es un caso de estudio proporcionado por la compañía EsperTech, es lógico que el propio ejemplo incluya un generador de eventos para que el usuario pueda analizar la aplicación y estudiar cómo realizar estas operaciones. Cada uno de los subtipos han sido implementados en diferentes clases (una por cada tipo de evento): {Cancelled, Checkin, Completed, LowPaper, OutOfOrder y Status}. En la tabla B.1 se presenta la normalización de cada uno de los subtipos, donde se especifica el nombre del dato (evento), junto con su tipo y una descripción del mismo.

Dato	Tipo	Descripción
event_type	String	Tipo de evento que lanza la terminal.
id	Integer	Identificador de la terminal.
timestamp	Long	Fecha y hora de registro del dato.

Tabla B.1: Eventos para el caso de estudio *Self-Service Terminal*.

El nombre del dato representado como *event_ type* nos indica el tipo de evento que lanza la terminal. Aunque el tipo de este campo sea de tipo *String*, los valores que puede tener este campo son los diferentes subtipos implementados basados en la clase "BaseTerminalEvent" {Cancelled, Checkin, Completed, LowPaper, OutOfOrder y Status}. El campo

id es de tipo *Integer*, y sus valores están comprendidos entre 0 y 99 (ambos incluidos). Finalmente, el campo *timestamp* es de tipo Long para representar la fecha y la hora del sistema.

Para hacer las pruebas con la metodología propuesta en la presente tesis (véase capítulo 5), se define el tipo de evento "TerminalEvent", que engloba los atributos presentados en la tabla B.1. Esta definición (véase la sección C.1) se emplea para generar eventos a través de IoT-TEG (véase capítulo 6). Adicionalmente se modifica la aplicación para que en vez de obtener los eventos del generador propio de la aplicación, estos sean obtenidos de un fichero en formato `CSV` que ha sido generado por IoT-TEG.

B.2. Caso de Estudio - Transaction

Otro de los casos de estudios que ofrece la compañía EsperTech, y al que también se le ha aplicado MuEPL, es "Transaction 3-Event Challenge" [162]. Este ejemplo consiste en hacer el seguimiento de los tres componentes de una transacción. En este caso de estudio se le da mucha importancia al uso de al menos tres componentes, representados como eventos, ya que algunos motores actúan o tienen una codificación diferente para transacciones donde tienen lugar dos eventos. Como se ha comentado, cada componente llega al motor como un evento con los campos que se muestran en la tabla B.2, del mismo modo se muestran los campos extras que incluyen algunos de estos componentes:

Evento	Dato	Tipo	Descripción
Todos	id de la transacción	String	Identificador de la terminal.
Todos	timestamp	Long	Fecha y hora de registro del dato.
Tipo A	customerID	String	Identificador del cliente.
Tipo C	supplierID	String	Identificador del proveedor.

Tabla B.2: Eventos para el caso de estudio *Transaction 3-Event Challenge*.

En este estudio se reúnen todos los tipos de eventos para obtener uno solo con los siguientes campos:

- transaction id
- customer id
- timestamp del evento tipo A
- timestamp del evento tipo B
- timestamp del evento tipo C

Se unifican los campos *transaction id* de cada evento, para formar un evento de agregación. Si todos estos eventos estuviesen en una base de datos relacional, la unión se podría hacer con una simple consulta SQL utilizando la cláusula JOIN. Pero si tenemos en cuenta que se esta trabajando con 10000 eventos por segundo, el hardware de la base de datos tendría que tener unas características muy excepcionales.

Lo que se pretende localizar en este caso de estudio son las transacciones que no logran realizarse con los tres componentes. Por ejemplo, una transacción con eventos de tipo A o de tipo B y no de tipo C. Nótese que, en este caso, lo que interesa son los eventos de tipo C. La carencia de eventos de tipo A o B podrían indicar un fallo en el transporte de eventos, este tipo de fallo podría ser ignorado. Sin embargo, la falta de eventos de tipo C podría ser un fallo en el transporte, lo cual ha de analizarse.

Al igual que el caso de estudio B.1, esta aplicación también cuenta con un generador de eventos. Este nos permite cambiar el tamaño del bloque de eventos {tiniest, tiny, small, medium, large, larger, largerer}, lo cual permite aumentar o disminuir el número de eventos que recibirá el programa.

Para hacer las pruebas con la metodología propuesta en la presente tesis (véase capítulo 5), se define el tipo de evento "TransactionEvent", que engloba los atributos presentados en la tabla B.2. Esta definición (véase la sección C.2) se emplea para generar eventos a través de IoT-TEG (véase capítulo 6). Adicionalmente se modifica la aplicación para que en vez de obtener los eventos del generador propio de la aplicación, estos sean obtenidos de un fichero en formato `CSV` que ha sido generado por IoT-TEG.

B.3. Caso de Estudio - Ecological Island

Este caso de estudio es un Trabajo Fin de Grado [163] que ha sido desarrollado con el objetivo de contribuir en el desarrollo de las *Smart Cities*, dotando a las islas ecológicas de cierta "inteligencia" para reducir costes en la empresa recolectora, reducir la contaminación medioambiental y aumentar la eficiencia energética así como velar para el cuidado del patrimonio de la ciudad siendo alertados los servicios de emergencia en su caso y, en general, hacerle al ciudadano la vida más placentera.

Las actividades que se notifican en esta aplicación son:

1. Detección de fuego: Si la temperatura tomada en un momento incrementa mucho en un intervalo de tiempo de 1 segundo.
2. Detección de que el contenedor está por la mitad: Si el contenido de uno de los contenedores de la isla está al 50 % de su capacidad.

3. La entrada del contenedor está bloqueada: Se detecta que la entrada del contenedor no puede moverse.

4. Detección de que el contenedor está casi lleno: Si el contenido de uno de los contenedores de la isla está al 90 % de su capacidad.

Como fuente de eventos se utiliza la plataforma IoT "ThinkSpeak", en concreto de los canales: 16302, 16306, 16307 y 16308. Toda la información sobre los atributos del tipo de evento que se emplea se muestran en la tabla B.3

Atributo	Tipo	Descripción
created_at	String	Fecha y hora a la que se captura el evento
entry_id	Integer	Identificador de la entrada
field1	Boolean	Lectura del Sensor_L0_N0
field2	Boolean	Lectura del Sensor_L0_N1
field3	Boolean	Lectura del Sensor_L0_N2
field4	Boolean	Lectura del Sensor_L1_N0
field5	Boolean	Lectura del Sensor_L1_N1
field6	Boolean	Lectura del Sensor_L1_N2
field7	Integer	Temperatura
field8	Boolean	Se indica si está o no bloqueado el contenedor

Tabla B.3: Información sobre los atributos del tipo de evento utilizado en el caso de estudio de EcologicalIsland.

La aplicación está desarrollada para leer en formato JSON los eventos que se capturan de los diferentes canales, para ello el autor implementó una clase en la que se leen los eventos en formato `JSON` (`JsonToSnifferEventTransformer`) y que asigna los valores según la estructura del tipo de evento "Island". Para para hacer las pruebas con la metodología propuesta en la presente tesis (véase capítulo 5), se define el tipo de evento "IslandEvent", que contiene los atributos presentados en la tabla B.3. Esta definición (véase la sección C.4) se emplea para generar eventos a través de IoT-TEG (véase capítulo 6) y para poder utilizarlos se reutiliza la clase `JsonToSnifferEventTransformer` comentada anteriormente.

B.4. Caso de Estudio - Domótica

Este caso de estudio es una versión del que está propuesto por Boubeta et al. en [164]. Este ejemplo simula los sensores de una casa inteligente que dependiendo de la información que reciben en tiempo real, actúan en consecuencia. Los patrones implementados que incluye la aplicación, activa las siguientes acciones automáticas:

1. Consumo irresponsable de energía: Se detectan situaciones en las que se alcanza un alto consumo de energía (más de 1500 vatios) en intervalos de tiempo de 10 minutos.

2. Detección de fuego: Si la temperatura tomada en un momento determinado se incrementa en 20 grados centígrados al minuto de ser tomada, se indica que hay fuego.

3. Fallo de potencia: Cuando el consumo de potencia es mayor que 0 vatios y pasa directamente a un consumo igual a 0, asumiendo que el consumo básico de una casa nunca será igual a 0.

4. Uso irresponsable de la televisión: Se detectan situaciones en las que se ha olvidado apagar el televisor, si este lleva encendido 6 horas, suponiendo que las personas no pasan más de 6 horas viendo la televisión.

Como fuente de eventos se utiliza la plataforma IoT “Xively”, una plataforma que ya no es abierta. El acceso a los eventos se ha realizado gracias a la cuenta de Boubeta en la plataforma que previamente se había creado. Esta plataforma ofrece los eventos de uno en uno, sin posibilidad de adquirir un lote de eventos, además de que no todos los canales de eventos se actualizan a la vez. Son tres los canales que se emplean en esta aplicación como fuentes de eventos, véase tabla B.4.

Nº	Tipo de evento	País	URL
F1	Residential information	The Netherlands	https://xively.com/feeds/62988
F2	Home Automation	The Netherlands	https://xively.com/feeds/71257
F3	Current Cost Bridge	Spain	https://xively.com/feeds/89125

Tabla B.4: Información sobre los canales utilizados como fuentes de eventos en el caso de estudio de Domótica.

Se tuvo que tener en cuenta que de cada uno de los canales, los valores de los atributos de los eventos podrían tener valores en distintas unidades (la temperatura podría venir en grados Celsius o Fahrenheit), así que se normalizaron los datos. En la tabla B.5 se muestran los atributos de los eventos, los tipos de los atributos, su descripción y se marca con una “X” si estos están disponibles en la fuente de eventos.

La aplicación está desarrollada para leer en formato `JSON` los eventos que se capturan de los diferentes canales, para ello se incluye una clase que diferencia el tipo de evento[2] según el canal del que le llegue la información. Para evitar hacer distinciones entre los tipos de eventos y para hacer las pruebas con la metodología propuesta en la presente tesis (véase capítulo 5), se define el tipo de evento “DomoticEvent”, que engloba los atributos

[2]Recuérdese que los atributos definen el tipo de evento.

Atributo	Tipo	Descripción	F1	F2	F3
Home	String	Nombre del tipo de evento	X	X	X
Sensor	String	URL de la fuente	X	X	X
Location	String	Ciudad, país	X	X	X
Latitude	Float	Latitud del sensor	X	X	X
Longitude	Float	Longitud del sensor	X	X	X
Timestamp	String	Timestamp	X	X	X
EnergyConsumption	Float	Consumo de energía (W)	X	X	X
Temperature	Float	Temperatura (C)	X	X	X
Humidity	Float	Humedad (%)	X	X	
TVConnection	Float	Conexión del televisor (on/off)			X

Tabla B.5: Información sobre los atributos de los tipos de eventos utilizados en el caso de estudio de Domótica.

presentados en la tabla B.5. Esta definición (véase la sección C.4) se emplea para generar eventos a través de IoT-TEG (véase capítulo 6) y para poder utilizarlos se implementa la clase `JsonToSnifferEventTransformer` en la que se leen los eventos en formato `JSON` y que asigna los valores según la estructura del tipo de evento “DomoticEvent”.

B.5. Caso de Estudio - DENMEvaP

Distributed Event-driven Network Monitoring Evaluation Prototype (DENMEvaP) [165] es un caso de estudio que consiste en un pequeño prototipo que consistente en dos equipos:

1. Un nodo sensor que actúa como un productor de eventos y adquiere los datos en formato raw de la red monitorizada.
2. Un nodo de procesamiento de datos que prepara la infraestructura de comunicación, el motor CEP y el consumidor de eventos que se utiliza para medir el rendimiento.

Esta aplicación está compuesta por diversos módulos que están implementados para detectar diferentes comportamientos en una infraestructura de comunicaciones. Debido a que está implementado con el lenguaje de programación Clojure[3] y por recomendación de su autor, Rüediger Gad, se han ejecutado cada módulo de forma independiente, dando lugar a los siguientes programas:

- Brute Force
- Sniffer Flat
- Sniffer Flood

[3]Lenguaje de programación funcional (dialecto de Lisp), que entre sus características incluye una sintaxis concisa para la mejora de la concurrencia.

- Simple Sniffer
- Sniffer Congestion

Adicionalmente, el autor recomienda extraer de cada módulo los patrones EPL y no ejecutar su aplicación para realizar las pruebas. Así que para poder utilizar los patrones EPL se implementa una aplicación Java (para cada módulo) que captura los eventos que serán procesados por los patrones. Cada uno de los programas implementados, que representan a cada módulo del caso de estudio DENMEvaP, incluyen:

- `EsperRuntimeFacade`: Una clase para abstraerse de Esper (EPServiceProvider y EPRuntime). La intención de esta clase es facilitar el uso de Esper en otro código.
- `CustomAwareListener`: Una clase que incluye *listeners* que permite extraer información sobre los eventos que cumplen los requisitos de los patrones EPL y que vienen en flujos de eventos.
- `ClasePrincipal`: En esta clase se obtienen los eventos, se incluyen los patrones EPL en el motor CEP y se lanzan los eventos al motor CEP para que sean procesados.
- `JsonToSnifferEventTransformer`: Clase en la que se leen los eventos en formato `JSON` y que asigna los valores según la estructura del tipo de evento que se esté estudiando.

El tipo de evento que manejan estas aplicaciones se denomina "Sniffer_flat", los atributos de que definen a este tipo de evento se pueden ver en la tabla B.6, así como sus descripciones.

El formato original de los eventos que maneja este caso de estudio es `pcap` (Packet CAPture). El conjunto de casos de pruebas utilizado es el siguiente:

- `arp_spoofing_attack_attacker_2012-10-11`: Los eventos se generaron a través de la herramienta Ettercap [169], se grabaron a través del ordenador del autor en el que se estaba haciendo el ataque de suplantación. El conjunto de eventos de este caso de prueba es de 6970.
- `arp_spoofing_attack_victim_2012-11-13`: También se generaron con Ettercap, en este caso se guardaron los eventos con el rol de víctima que estaba siendo suplantada en una red local. Este caso de prueba contiene 4712 eventos. El propósito general del ataque (en ambos casos) era re-direccionar el tráfico entre la víctima y el router (gateway estándar) en la red local, de tal forma que el atacante se estuviese entre ambos.

Atributo	Tipo	Descripción
EthernetSrc	Complejo	Dirección Ethernet origen
EthernetDst	Complejo	Dirección Ethernet destino
udpSrc	Integer	User Datagram Protocol origen
udpDst	Integer	User Datagram Protocol destino
len	Integer	Longitud
timestamp	Long	Timestamp
sequence	Integer	Secuencia
icmpType	Complejo	El tipo de Internet Control Message Protocol `{echo request, echo reply, destination unreachable}`
ipTtl	Integer	Tiempo de vida, número de nodos por los que pasa un paquete antes de ser descartado.
ipVer	Integer	Versión del Internet Protocol (IP)
ipChecksum	Integer	Suma de verificación
tcpFlags	Integer	Número de bandera del Protocolo de Transmisión
sourcePort	Integer	Número del puerto de origen
destinationPort	Integer	Número del puerto de destino
source	Complejo	Dirección del origen
destination	Complejo	Dirección del destino
sourceIp	Complejo	Dirección IP del origen
targetIp	Complejo	Dirección IP del destino
arpOpDesc	Complejo	Tipo de paquete del Protocolo de Resolución de Direcciones `{REQUEST, REPLY}`
sourceMac	Complejo	Dirección MAC del origen
targetMac	Complejo	Dirección MAC del destino

Tabla B.6: Información sobre los atributos del tipo de evento "Sniffer_flat".

- `dns-http-icm_wlan_2013-11-30`: Estos eventos fueron generados a través de scripts creados por el autor, estos seguían el siguiente proceso: generar tráfico DNS resolviendo las direcciones IP de un listado de nombres de dominio populares, luego se descargaban de forma automática las webs de un listado de páginas web populares y por último se realizaba `"ping"` para generar tráfico. Todo realizado a través de WLAN. El número de eventos de este caso de prueba es de 133.

- `ssh_brute_force_sshpass_wlan0_2012-11-26`: Los eventos se generaron a través de scripts utilizando sshpass [170]. El tráfico de eventos se creó intentando, de forma repetitiva, la conexión a un servidor SSH a través de un cliente SSH. Las llamadas del cliente ejecutadas a través de sshpass, con unos credenciales (nombre de usuario y contraseña) equivocados (algo realizado de forma intencionada) para simular accesos fallidos al servidor. La idea era simular un ataque por fuerza bruta mediante conexiones SSH, donde el atacante intenta adivinar el nombre de usuario y contraseña, probando varias combinaciones. El caso de prueba tiene 6917 eventos.

- `syn_flood_sender_non-rand_2012-11-07`: El autor pretendía simular un ataque TCP SYN por saturación con el paquete generador Mausezahn [171], en este caso las direcciones IP origen no fueron aleatorias. El conjunto de este caso de prueba tiene 9439 eventos.

- `syn_flood_sender_rand_2012-11-26`: Se realiza la misma simulación de ataque TCP SYN por saturación con el paquete generador Mausezahn, pero ahora las direcciones IP origen en la simulación fueron aleatorias. El conjunto de eventos de este caso de prueba es 10010.

Para la lectura de los eventos en este formato, se utiliza en la `ClasePrincipal` de cada programa, una clase implementada por el autor que permite extraer en forma de `Map` los valores de los ficheros `pcap` (`CljNetPcapJavaAdapter`). Una vez que están almacenados en estas estructuras, se pueden lanzar los eventos al motor CEP. Para hacer las pruebas con la metodología propuesta en la presente tesis (véase capítulo 5), se define el tipo de evento "Sniffer_flat", que engloba los atributos presentados en la tabla B.6. Esta definición (véase la sección C.5) se emplea para generar eventos a través de IoT-TEG (véase capítulo 6) y para poder utilizarlos se implementa la clase `JsonToSnifferEventTransformer` en la que se leen los eventos en formato `JSON` y que asigna los valores según la estructura del tipo de evento "Sniffer_flat".

Apéndice C

Definición de Eventos de los Casos de Estudio

En este apéndice se definen, con la propuesta de especificación de definición de tipos de eventos, los tipos de eventos utilizados en la presente tesis (véase Apéndice B).

C.1. Caso de Estudio - Terminal

Definición del tipo de evento “TerminalEvent” del caso de estudio “Terminal” (véase anexo B.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<event name="TerminalEvent">
  <block name="feeds" repeat="100">
    <field name="status" quotes="true" type="ComplexType"
                                                        chooseone="true">
      <attribute type="String" value="OutOfOrder"></attribute>
      <attribute type="String" value="Checkin"></attribute>
      <attribute type="String" value="Cancelled"></attribute>
      <attribute type="String" value="Completed"></attribute>
    </field>
    <field name="terminal_id" quotes="false" type="Integer" min="0"
                                                        max="99"></field>
    <field name="timestamp" quotes="false" type="Long"
                         min="1000000000000" max="9999999999999"></field>
  </block>
</event>
```

C.2. Caso de Estudio - Transaction

Definición del tipo de evento “TransactionEvent” del caso de estudio “Transaction” (véase anexo B.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<event name="TransactionEvent">
  <block name="feeds" repeat="100">
    <field name="transaction_id" quotes="false" type="Long"
                              min="1000000000" max="9999999999"></field>
    <field name="timestamp" quotes="false" type="Long"
                         min="1000000000000" max="9999999999999"></field>
    <optionalfields possibleempty="true">
      <field name="customerId" quotes="true" type="ComplexType"
                                                   chooseone="true">
        <attribute type="String" value="YELLOW"></attribute>
        <attribute type="String" value="ORANGE"></attribute>
        <attribute type="String" value="GREEN"></attribute>
        <attribute type="String" value="INDIGO"></attribute>
        <attribute type="String" value="RED"></attribute>
        <attribute type="String" value="BLUE"></attribute>
      </field>
      <field name="supplierId" quotes="true" type="ComplexType"
                                                   chooseone="true">
        <attribute type="String" value="ADAMS"></attribute>
        <attribute type="String" value="WASHINGTON"></attribute>
        <attribute type="String" value="MADISON"></attribute>
        <attribute type="String" value="JEFFERSON"></attribute>
      </field>
    </optionalfields>
  </block>
</event>
```

C.3. Caso de Estudio - Ecological Island

Definición del tipo de evento “IslandEvent” del caso de estudio “Ecological Island” (véase anexo B.3).


```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="IslandEvent">
 <block name="head" value="&quot;id&quot;:16306,&quot;name&quot;:
 &quot;Island03&quot;,&quot;description&quot;:&quot;Island03&quot;,
 &quot;field1&quot;:&quot;Sensor_L0_N0&quot;,&quot;field2&quot;:&quot;
 Sensor_L0_N1&quot;,&quot;field3&quot;:&quot;Sensor_L0_N2&quot;,&quot;
 field4&quot;:&quot;Sensor_L1_N0&quot;,&quot;field5&quot;:&quot;
 Sensor_L1_N1&quot;,&quot;field6&quot;:&quot;Sensor_L1_N2&quot;,&quot;
 field7&quot;:&quot;Temperature&quot;,&quot;field8&quot;:&quot;
 Blocked&quot;,&quot;created_at&quot;:&quot;2014-09-22T21:36:59Z&quot;,
 &quot;updated_at&quot;:&quot;2015-11-01T17:42:14Z&quot;,&quot;
 last_entry_id&quot;:427088">
 </block>
 <block name="feeds" repeat="100">
   <field name="create_at" quotes="true" type="ComplexType">
     <attribute type="Date" format="yyyy-MM-dd"></attribute>
     <attribute type="String" value="T"></attribute>
     <attribute type="Time" format="hh:mm:ss"></attribute>
     <attribute type="String" value="Z"></attribute>
   </field>
   <field name="entry_id" quotes="false" type="Integer" min="100000"
                                                  max="999999"></field>
   <field name="field1" quotes="true" type="Boolean" isnumeric="true">
   </field>
   <field name="field2" quotes="true" type="Boolean" isnumeric="true">
   </field>
   <field name="field3" quotes="true" type="Boolean" isnumeric="true">
   </field>
   <field name="field4" quotes="true" type="Boolean" isnumeric="true">
   </field>
   <field name="field5" quotes="true" type="Boolean" isnumeric="true">
   </field>
   <field name="field6" quotes="true" type="Boolean" isnumeric="true">
   </field>
```

```
    <field name="field7" quotes="true" type="Integer" min="0"
                                                       max="99"></field>
    <field name="field8" quotes="true" type="Boolean" isnumeric="true">
    </field>
  </block>
</event>
```

C.4. Caso de Estudio - Domotic

Definición del tipo de evento “DomoticEvent” del caso de estudio “Domotic” (véase B.4).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="DomoticEvent">
 <block name="feeds" repeat="100">
   <field name="home" quotes="true" type="String" length="10"></field>
   <field name="sensor" quotes="true" type="ComplexType">
     <attribute type="String" value="http://www.sensor"></attribute>
     <attribute type="String" length="6"></attribute>
     <attribute type="String" value=".com"></attribute>
   </field>
   <field name="location" quotes="false" type="ComplexType">
     <field name="name" quotes="true" type="String" length="10"></field>
     <field name="latitud" quotes="false" type="Float"
                                                    precision="2"></field>
     <field name="longitud" quotes="false" type="Float"
                                                    precision="2"></field>
   </field>
   <field name="timestamp" quotes="true" type="Time"
                                                format="hh:mm:ss"></field>
   <field name="energyConsumption" quotes="false" type="Float"
                                                    precision="2"></field>
   <field name="temperature" quotes="false" type="Float"
                                                    precision="2"></field>
   <field name="humidity" quotes="false" type="Float" precision="2">
   </field>
   <field name="tvConsumption" quotes="false" type="Float"
                                                    precision="2"></field>
 </block>
</event>
```

C.5. Caso de Estudio - DENMEvaP

C.5.1. Módulo - Brute Force

Definición del tipo de evento “BruteForce” para el módulo “Brute Force” del caso de estudio “DENMEvaP” (véase B.5).

```
<?xml version="1.0" encoding="UTF-8"?>
<event name="BruteForceEvent">
  <block name="info" value="&quot;name&quot;:&quot;Sniffer&quot;,&quot;
  description&quot;:&quot;Sniffer&quot;,&quot;ethSrc&quot;:&quot;
  ethSrc&quot;,&quot;ethDst&quot;:&quot;ethDst&quot;,&quot;udpDst&quot;:
  &quot;udpDst&quot;,&quot;udpSrc&quot;:&quot;udpSrc&quot;,&quot;len&quot
  ;:&quot;len&quot;,&quot;ts&quot;:&quot;ts&quot;,&quot;icmpEchoSeq&quot
  ;:&quot;icmpEchoSeq&quot;,&quot;icmpType&quot;:&quot;icmpType&quot;,
  &quot;ipTtl&quot;:&quot;ipTtl&quot;,&quot;ipVer&quot;:&quot;ipVer&quot;
  ,&quot;ipChecksum&quot;:&quot;ipChecksum&quot;,&quot;ipId&quot;:&quot;
  ipId&quot;,&quot;tcpFlags&quot;:&quot;tcpFlags&quot;,&quot;tcpSrc&quot;
  :&quot;tcpSrc&quot;,&quot;tcpDst&quot;:&quot;tcpDst&quot;,&quot;
  ipSrc&quot;:&quot;ipSrc&quot;,&quot;arpSourceIp&quot;:&quot;
  arpSourceIp&quot;,&quot;arpTargetIp&quot;:&quot;arpTargetIp&quot;,&quot
  ;arpOpDesc&quot;:&quot;arpOpDesc&quot;,&quot;arpSourceMac&quot;:&quot;
  arpSourceMac&quot;,&quot;arpTargetMac&quot;:&quot;arpTargetMac&quot;">
  </block>
  <block name="feeds" repeat="10000">
    <field name="ethSrc" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
```

```
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="ethDst" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="udpDst" quotes="false" type="Integer" min="0" max="99">
  </field>
  <field name="udpSrc" quotes="false" type="Integer" min="0" max="99">
  </field>
  <field name="len" quotes="false" type="Integer" min="1" max="99">
  </field>
  <field name="timestamp" quotes="false" type="Long"
          min="1300000000000000000" max="1399999999999999999"></field>
  <field name="sequence" quotes="false" type="Integer" min="1000"
                                              max="1050"></field>
  <field name="icmpType" quotes="true" type="ComplexType"
                                                chooseone="true">
    <attribute type="String" value="echo request"></attribute>
```

```
    <attribute type="String" value="echo reply"></attribute>
    <attribute type="String" value="destination unreachable">
    </attribute>
  </field>
  <field name="ipTtl" quotes="false" type="Integer" min="100"
                                               max="500"></field>
  <field name="ipVer" quotes="false" type="Integer" min="1" max="5">
  </field>
  <field name="ipChecksum" quotes="false" type="Integer" min="10000"
                                             max="40000"></field>
  <field name="ipId" quotes="false" type="Integer" min="10000"
                                             max="40000"></field>
  <field name="tcpFlags" quotes="false" type="ComplexType"
                                                chooseone="true">
    <attribute type="Integer" value="1"></attribute>
    <attribute type="Integer" value="2"></attribute>
  </field>
  <field name="sourcePort" quotes="false" type="Integer" min="0"
                                                  max="30"></field>
  <field name="destinationPort" quotes="false" type="Integer" min="0"
                                                  max="30"></field>
  <field name="source" quotes="true" type="ComplexType"
                                                 chooseone="true">
    <attribute type="String" value="45.128.192.192"></attribute>
    <attribute type="String" value="32.25.151.255"></attribute>
    <attribute type="String" value="124.256.182.74"></attribute>
    <attribute type="String" value="1.12.32.255"></attribute>
    <attribute type="String" value="122.203.198.10"></attribute>
    <attribute type="String" value="182.255.202.16"></attribute>
    <attribute type="String" value="200.134.8.72"></attribute>
    <attribute type="String" value="25.75.198.46"></attribute>
  </field>
  <field name="destination" quotes="true" type="ComplexType"
                                                 chooseone="true">
    <attribute type="String" value="45.128.192.192"></attribute>
    <attribute type="String" value="32.25.151.255"></attribute>
    <attribute type="String" value="124.256.182.74"></attribute>
    <attribute type="String" value="1.12.32.255"></attribute>
    <attribute type="String" value="122.203.198.10"></attribute>
```

```
    <attribute type="String" value="182.255.202.16"></attribute>
    <attribute type="String" value="200.134.8.72"></attribute>
    <attribute type="String" value="25.75.198.46"></attribute>
  </field>
  <field name="sourceIp" quotes="true" type="ComplexType">
    <attribute type="Integer" value="192"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="targetIp" quotes="true" type="ComplexType">
    <attribute type="Integer" value="192"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="arpOpDesc" quotes="true" type="ComplexType"
                                            chooseone="true">
    <attribute type="String" value="REPLY"></attribute>
    <attribute type="String" value="REQUEST"></attribute>
  </field>
  <field name="sourceMac" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
```

```
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
    <field name="targetMac" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
  </block>
</event>
```

C.5.2. Módulo - Sniffer Flat

Definición del tipo de evento “SnifferFlat” para el módulo “Sniffer Flat” del caso de estudio “DENMEvaP” (véase B.5).


```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="SnifferFlatEvent">
  <block name="info" value="&quot;name&quot;:&quot;Sniffer&quot;,&quot;
  description&quot;:&quot;Sniffer&quot;,&quot;ethSrc&quot;:&quot;
  ethSrc&quot;,&quot;ethDst&quot;:&quot;ethDst&quot;,&quot;udpDst&quot;:
  &quot;udpDst&quot;,&quot;udpSrc&quot;:&quot;udpSrc&quot;,&quot;len&quot
  ;:&quot;len&quot;,&quot;ts&quot;:&quot;ts&quot;,&quot;icmpEchoSeq&quot
  ;:&quot;icmpEchoSeq&quot;,&quot;icmpType&quot;:&quot;icmpType&quot;,
  &quot;ipTtl&quot;:&quot;ipTtl&quot;,&quot;ipVer&quot;:&quot;ipVer&quot;
  ,&quot;ipChecksum&quot;:&quot;ipChecksum&quot;,&quot;ipId&quot;:&quot;
  ipId&quot;,&quot;tcpFlags&quot;:&quot;tcpFlags&quot;,&quot;tcpSrc&quot;
  :&quot;tcpSrc&quot;,&quot;tcpDst&quot;:&quot;tcpDst&quot;,&quot;
  ipSrc&quot;:&quot;ipSrc&quot;,&quot;arpSourceIp&quot;:&quot;
  arpSourceIp&quot;,&quot;arpTargetIp&quot;:&quot;arpTargetIp&quot;,&quot
  ;arpOpDesc&quot;:&quot;arpOpDesc&quot;,&quot;arpSourceMac&quot;:&quot;
  arpSourceMac&quot;,&quot;arpTargetMac&quot;:&quot;arpTargetMac&quot;">
  </block>
  <block name="feeds" repeat="10000">
    <field name="ethSrc" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
```

```
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="ethDst" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="udpDst" quotes="false" type="Integer" min="0" max="99">
  </field>
  <field name="udpSrc" quotes="false" type="Integer" min="0" max="99">
  </field>
  <field name="len" quotes="false" type="Integer" min="1" max="99">
  </field>
  <field name="timestamp" quotes="false" type="Long"
        min="1300000000000000000" max="1399999999999999999"></field>
  <field name="sequence" quotes="false" type="Integer" min="1000"
                                                  max="1050"></field>
  <field name="icmpType" quotes="true" type="ComplexType"
                                                  chooseone="true">
    <attribute type="String" value="echo request"></attribute>
    <attribute type="String" value="echo reply"></attribute>
    <attribute type="String" value="destination unreachable">
    </attribute>
```

```
</field>
<field name="ipTtl" quotes="false" type="Integer" min="100"
                                                        max="500"></field>
<field name="ipVer" quotes="false" type="Integer" min="1" max="5">
</field>
<field name="ipChecksum" quotes="false" type="Integer" min="10000"
                                                      max="40000"></field>
<field name="ipId" quotes="false" type="Integer" min="10000"
                                                      max="40000"></field>
<field name="tcpFlags" quotes="false" type="ComplexType"
                                                        chooseone="true">
  <attribute type="Integer" value="1"></attribute>
  <attribute type="Integer" value="2"></attribute>
  <attribute type="Integer" value="16"></attribute>
  <attribute type="Integer" value="18"></attribute>
  <attribute type="Integer" value="20"></attribute>
</field>
<field name="sourcePort" quotes="false" type="Integer" min="0"
                                                        max="10"></field>
<field name="destinationPort" quotes="false" type="Integer" min="0"
                                                        max="10"></field>
<field name="source" quotes="true" type="ComplexType"
                                                        chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
</field>
<field name="destination" quotes="true" type="ComplexType"
                                                        chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
```

```
    <attribute type="String" value="182.255.202.16"></attribute>
    <attribute type="String" value="200.134.8.72"></attribute>
    <attribute type="String" value="25.75.198.46"></attribute>
  </field>
  <field name="sourceIp" quotes="true" type="ComplexType">
    <attribute type="Integer" value="192"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="targetIp" quotes="true" type="ComplexType">
    <attribute type="Integer" value="192"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="arpOpDesc" quotes="true" type="ComplexType"
                                              chooseone="true">
    <attribute type="String" value="REPLY"></attribute>
    <attribute type="String" value="REQUEST"></attribute>
  </field>
  <field name="sourceMac" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
```

```
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
    <field name="targetMac" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
  </block>
</event>
```

C.5.3. Módulo - Sniffer Flood

Definición del tipo de evento “SnifferFlood” para el módulo “Sniffer Flood” del caso de estudio “DENMEvaP” (véase B.5).


```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="SnifferFloodEvent">
  <block name="info" value="&quot;name&quot;:&quot;Sniffer&quot;,&quot;
  description&quot;:&quot;Sniffer&quot;,&quot;ethSrc&quot;:&quot;
  ethSrc&quot;,&quot;ethDst&quot;:&quot;ethDst&quot;,&quot;udpDst&quot;:
  &quot;udpDst&quot;,&quot;udpSrc&quot;:&quot;udpSrc&quot;,&quot;len&quot
  ;:&quot;len&quot;,&quot;ts&quot;:&quot;ts&quot;,&quot;icmpEchoSeq&quot
  ;:&quot;icmpEchoSeq&quot;,&quot;icmpType&quot;:&quot;icmpType&quot;,
  &quot;ipTtl&quot;:&quot;ipTtl&quot;,&quot;ipVer&quot;:&quot;ipVer&quot;
  ,&quot;ipChecksum&quot;:&quot;ipChecksum&quot;,&quot;ipId&quot;:&quot;
  ipId&quot;,&quot;tcpFlags&quot;:&quot;tcpFlags&quot;,&quot;tcpSrc&quot;
  :&quot;tcpSrc&quot;,&quot;tcpDst&quot;:&quot;tcpDst&quot;,&quot;
  ipSrc&quot;:&quot;ipSrc&quot;,&quot;arpSourceIp&quot;:&quot;
  arpSourceIp&quot;,&quot;arpTargetIp&quot;:&quot;arpTargetIp&quot;,&quot
  ;arpOpDesc&quot;:&quot;arpOpDesc&quot;,&quot;arpSourceMac&quot;:&quot;
  arpSourceMac&quot;,&quot;arpTargetMac&quot;:&quot;arpTargetMac&quot;">
  </block>
  <block name="feeds" repeat="10000">
    <field name="ethSrc" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
```

```
  </attribute>
  <attribute type="String" value=":"></attribute>
</field>
<field name="ethDst" quotes="true" type="ComplexType">
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
</field>
<field name="udpDst" quotes="false" type="Integer" min="0"
                                            max="99"></field>
<field name="udpSrc" quotes="false" type="Integer" min="0"
                                            max="99"></field>
<field name="len" quotes="false" type="Integer" min="1"
                                            max="99"></field>
<field name="timestamp" quotes="false" type="Long"
        min="1300000000000000000" max="1399999999999999999"></field>
<field name="sequence" quotes="false" type="Integer" min="1000"
                                                 max="1050"></field>
<field name="icmpType" quotes="true" type="ComplexType"
                                                   chooseone="true">
  <attribute type="String" value="echo request"></attribute>
  <attribute type="String" value="echo reply"></attribute>
  <attribute type="String" value="destination unreachable">
  </attribute>
```

```
</field>
<field name="ipTtl" quotes="false" type="Integer" min="100"
                                                   max="500"></field>
<field name="ipVer" quotes="false" type="Integer" min="1" max="5">
</field>
<field name="ipChecksum" quotes="false" type="Integer" min="10000"
                                                 max="40000"></field>
<field name="ipId" quotes="false" type="Integer" min="10000"
                                                 max="40000"></field>
<field name="tcpFlags" quotes="false" type="Integer" min="0"
                                                    max="20"></field>
<field name="sourcePort" quotes="false" type="Integer" min="0"
                                                    max="30"></field>
<field name="destinationPort" quotes="false" type="Integer" min="0"
                                                    max="30"></field>
<field name="source" quotes="true" type="ComplexType"
                                                    chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
</field>
<field name="destination" quotes="true" type="ComplexType"
                                                    chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
</field>
<field name="sourceIp" quotes="true" type="ComplexType">
  <attribute type="Integer" value="192"></attribute>
```

```
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="targetIp" quotes="true" type="ComplexType">
    <attribute type="Integer" value="192"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="arpOpDesc" quotes="true" type="ComplexType"
                                          chooseone="true">
    <attribute type="String" value="REPLY"></attribute>
    <attribute type="String" value="REQUEST"></attribute>
  </field>
  <field name="sourceMac" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
```

```
      <attribute type="String" value=":"></attribute>
    </field>
    <field name="targetMac" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
  </block>
</event>
```

C.5.4. Módulo - Simple Sniffer

Definición del tipo de evento "SimpleSniffer" para el módulo "Simple Sniffer" del caso de estudio "DENMEvaP" (véase B.5).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="SimpleSnifferEvent">
  <block name="info" value="&quot;name&quot;:&quot;Sniffer&quot;,&quot;
  description&quot;:&quot;Sniffer&quot;,&quot;ethSrc&quot;:&quot;
  ethSrc&quot;,&quot;ethDst&quot;:&quot;ethDst&quot;,&quot;udpDst&quot;:
  &quot;udpDst&quot;,&quot;udpSrc&quot;:&quot;udpSrc&quot;,&quot;len&quot
  ;:&quot;len&quot;,&quot;ts&quot;:&quot;ts&quot;,&quot;icmpEchoSeq&quot
  ;:&quot;icmpEchoSeq&quot;,&quot;icmpType&quot;:&quot;icmpType&quot;,
  &quot;ipTtl&quot;:&quot;ipTtl&quot;,&quot;ipVer&quot;:&quot;ipVer&quot;
  ,&quot;ipChecksum&quot;:&quot;ipChecksum&quot;,&quot;ipId&quot;:&quot;
  ipId&quot;,&quot;tcpFlags&quot;:&quot;tcpFlags&quot;,&quot;tcpSrc&quot;
  :&quot;tcpSrc&quot;,&quot;tcpDst&quot;:&quot;tcpDst&quot;,&quot;
  ipSrc&quot;:&quot;ipSrc&quot;,&quot;arpSourceIp&quot;:&quot;
  arpSourceIp&quot;,&quot;arpTargetIp&quot;:&quot;arpTargetIp&quot;,&quot
  ;arpOpDesc&quot;:&quot;arpOpDesc&quot;,&quot;arpSourceMac&quot;:&quot;
  arpSourceMac&quot;,&quot;arpTargetMac&quot;:&quot;arpTargetMac&quot;">
  </block>
  <block name="feeds" repeat="10000">
    <field name="ethSrc" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
```

```
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="ethDst" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="udpDst" quotes="false" type="Integer" min="0"
                                              max="99"></field>
  <field name="udpSrc" quotes="false" type="Integer" min="0"
                                              max="99"></field>
  <field name="len" quotes="false" type="Integer" min="1"
                                              max="99"></field>
  <field name="timestamp" quotes="false" type="Long"
     min="1300000000000000000" max="1399999999999999999"></field>
  <field name="sequence" quotes="false" type="Integer" min="1000"
                                            max="1050"></field>
  <field name="icmpType" quotes="true" type="ComplexType"
                                              chooseone="true">
    <attribute type="String" value="echo request"></attribute>
    <attribute type="String" value="echo reply"></attribute>
    <attribute type="String" value="destination unreachable">
    </attribute>
```

```
</field>
<field name="ipTtl" quotes="false" type="Integer" min="100"
                                        max="500"></field>
<field name="ipVer" quotes="false" type="Integer" min="1" max="5">
</field>
<field name="ipChecksum" quotes="false" type="Integer" min="10000"
                                        max="40000"></field>
<field name="ipId" quotes="false" type="Integer" min="10000"
                                        max="40000"></field>
<field name="tcpFlags" quotes="false" type="ComplexType"
                                        chooseone="true">
  <attribute type="Integer" value="1"></attribute>
  <attribute type="Integer" value="2"></attribute>
</field>
<field name="sourcePort" quotes="false" type="Integer" min="0"
                                        max="30"></field>
<field name="destinationPort" quotes="false" type="Integer" min="0"
                                        max="30"></field>
<field name="source" quotes="true" type="ComplexType"
                                        chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
</field>
<field name="destination" quotes="true" type="ComplexType"
                                        chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
```

```
</field>
<field name="sourceIp" quotes="true" type="ComplexType">
  <attribute type="Integer" value="192"></attribute>
  <attribute type="String" value="."></attribute>
  <attribute type="Integer" value="168"></attribute>
  <attribute type="String" value="."></attribute>
  <attribute type="Integer" value="20"></attribute>
  <attribute type="String" value="."></attribute>
  <attribute type="Integer" min="0" max="255"></attribute>
</field>
<field name="targetIp" quotes="true" type="ComplexType">
  <attribute type="Integer" value="192"></attribute>
  <attribute type="String" value="."></attribute>
  <attribute type="Integer" value="168"></attribute>
  <attribute type="String" value="."></attribute>
  <attribute type="Integer" value="20"></attribute>
  <attribute type="String" value="."></attribute>
  <attribute type="Integer" min="0" max="255"></attribute>
</field>
<field name="arpOpDesc" quotes="true" type="ComplexType"
                                             chooseone="true">
  <attribute type="String" value="REPLY"></attribute>
  <attribute type="String" value="REQUEST"></attribute>
</field>
<field name="sourceMac" quotes="true" type="ComplexType">
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
  <attribute type="String" value=":"></attribute>
  <attribute type="Alphanumeric" length="2" endcharacter="F">
  </attribute>
```

```
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
    <field name="targetMac" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
  </block>
</event>
```

C.5.5. Módulo - Sniffer Congestion

Definición del tipo de evento “SnifferCongestion” para el módulo “Sniffer Congestion” del caso de estudio “DENMEvaP” (véase B.5).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="SnifferCongestionEvent">
  <block name="info" value="&quot;name&quot;:&quot;Sniffer&quot;,&quot;
  description&quot;:&quot;Sniffer&quot;,&quot;ethSrc&quot;:&quot;
  ethSrc&quot;,&quot;ethDst&quot;:&quot;ethDst&quot;,&quot;udpDst&quot;:
  &quot;udpDst&quot;,&quot;udpSrc&quot;:&quot;udpSrc&quot;,&quot;len&quot
  ;:&quot;len&quot;,&quot;ts&quot;:&quot;ts&quot;,&quot;icmpEchoSeq&quot
  ;:&quot;icmpEchoSeq&quot;,&quot;icmpType&quot;:&quot;icmpType&quot;,
  &quot;ipTtl&quot;:&quot;ipTtl&quot;,&quot;ipVer&quot;:&quot;ipVer&quot;
  ,&quot;ipChecksum&quot;:&quot;ipChecksum&quot;,&quot;ipId&quot;:&quot;
  ipId&quot;,&quot;tcpFlags&quot;:&quot;tcpFlags&quot;,&quot;tcpSrc&quot;
  :&quot;tcpSrc&quot;,&quot;tcpDst&quot;:&quot;tcpDst&quot;,&quot;
  ipSrc&quot;:&quot;ipSrc&quot;,&quot;arpSourceIp&quot;:&quot;
  arpSourceIp&quot;,&quot;arpTargetIp&quot;:&quot;arpTargetIp&quot;,&quot
  ;arpOpDesc&quot;:&quot;arpOpDesc&quot;,&quot;arpSourceMac&quot;:&quot;
  arpSourceMac&quot;,&quot;arpTargetMac&quot;:&quot;arpTargetMac&quot;">
  </block>
  <block name="feeds" repeat="10000">
    <field name="ethSrc" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
```

```
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="ethDst" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
  </field>
  <field name="udpDst" quotes="false" type="Integer" min="0" max="99">
  </field>
  <field name="udpSrc" quotes="false" type="Integer" min="0" max="99">
  </field>
  <field name="len" quotes="false" type="Integer" min="1" max="99">
  </field>
  <field name="timestamp" quotes="false" type="Long"
          min="1300000000000000000" max="1399999999999999999"></field>
  <field name="sequence" quotes="false" type="Integer" min="1000"
                                                   max="1050"></field>
  <field name="icmpType" quotes="true" type="ComplexType"
                                                     chooseone="true">
    <attribute type="String" value="echo request"></attribute>
    <attribute type="String" value="echo reply"></attribute>
    <attribute type="String" value="destination unreachable">
    </attribute>
```

```
</field>
<field name="ipTtl" quotes="false" type="Integer" min="100"
                                                max="500"></field>
<field name="ipVer" quotes="false" type="Integer" min="1" max="5">
</field>
<field name="ipChecksum" quotes="false" type="Integer" min="10000"
                                                max="40000"></field>
<field name="ipId" quotes="false" type="Integer" min="10000"
                                                max="40000"></field>
<field name="tcpFlags" quotes="false" type="Integer" min="0"
                                                max="50"></field>
<field name="sourcePort" quotes="false" type="Integer" min="0"
                                                max="30"></field>
<field name="destinationPort" quotes="false" type="Integer" min="0"
                                                max="30"></field>
<field name="source" quotes="true" type="ComplexType"
                                                chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
</field>
<field name="destination" quotes="true" type="ComplexType"
                                                chooseone="true">
  <attribute type="String" value="45.128.192.192"></attribute>
  <attribute type="String" value="32.25.151.255"></attribute>
  <attribute type="String" value="124.256.182.74"></attribute>
  <attribute type="String" value="1.12.32.255"></attribute>
  <attribute type="String" value="122.203.198.10"></attribute>
  <attribute type="String" value="182.255.202.16"></attribute>
  <attribute type="String" value="200.134.8.72"></attribute>
  <attribute type="String" value="25.75.198.46"></attribute>
</field>
<field name="sourceIp" quotes="true" type="ComplexType">
  <attribute type="Integer" value="192"></attribute>
```

```
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="targetIp" quotes="true" type="ComplexType">
    <attribute type="Integer" value="192"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="168"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" value="20"></attribute>
    <attribute type="String" value="."></attribute>
    <attribute type="Integer" min="0" max="255"></attribute>
  </field>
  <field name="arpOpDesc" quotes="true" type="ComplexType"
                                              chooseone="true">
    <attribute type="String" value="REPLY"></attribute>
    <attribute type="String" value="REQUEST"></attribute>
  </field>
  <field name="sourceMac" quotes="true" type="ComplexType">
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
    <attribute type="String" value=":"></attribute>
    <attribute type="Alphanumeric" length="2" endcharacter="F">
    </attribute>
```

```
      <attribute type="String" value=":"></attribute>
    </field>
    <field name="targetMac" quotes="true" type="ComplexType">
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
      <attribute type="Alphanumeric" length="2" endcharacter="F">
      </attribute>
      <attribute type="String" value=":"></attribute>
    </field>
  </block>
</event>
```

Bibliografía

[1] R.Vijaya Arjunan and Nagapandu Potti. Framework to monitor big data processing in the cloud. *Networking and Communication Engineering*, 6(8), 2014. ISSN 0974 – 9616.

[2] Dain Hansen. *Oracle fast data: Real-time strategies for big data and business analytics*. An Oracle white paper, Oracle, 2013.

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

[4] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[5] Pierre Bourque and Fairley Richard E. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.

[6] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972. ISBN 0-12-200550-3.

[7] Gregory M. Kapfhammer. Software testing. In *The Computer Science Handbook*. CRC Press, 2004.

[8] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 654–665, Hong Kong, November 18–20, 2014.

[9] EsperTech. Espertech website. `http://www.espertech.com/esper/index.php`, . Accedida 02-18-2016.

[10] D Giusto. A. lera, g. morabito, l. atzori (eds.) the internet of things, 2010.

[11] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[12] T McCourt, Simon Leopold, F Louthan, Hans Mosesmann, S Smigie, T Tilman, Daniel Toomey, Georgios Kyriakopoulos, Eric Lemus, Brian Peterson, et al. The internet of things: A study in hype, reality, disruption, and growth. *Raymond James & Associates, USA*, 2014.

[13] Ovidiu Vermesan and Peter Friess. *Internet of things-from research and innovation to market deployment.* River Publishers Aalborg, 2014.

[14] ITU-T. Internet of things global standards initiative. `http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx`.

[15] VM Larios, JG Robledo, L Gómez, and R Rincon. Ieee-gdl ccd smart buildings introduction. *online at http://smartcities. ieee. org/images/files/images/pdf/whitepaper phi smartbuildingsv6. pdf*, 2014.

[16] Event-driven architecture. In *Enterprise Service Oriented Architectures*, pages 317–355. Springer Netherlands, 2006. ISBN 978-1-4020-3704-7. doi: 10.1007/1-4020-3705-8_8.

[17] K.Mani Chandy. Event driven architecture. In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 1040–1044. Springer US, 2009. ISBN 978-0-387-35544-3. doi: 10.1007/978-0-387-39940-9_570.

[18] Grygoriy Zholtkevych, Boris Novikov, and Volodymyr Dorozhinsky. Pre-automata and complex event processing. In *International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications*, pages 100–116. Springer, 2014.

[19] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.

[20] Opher Etzion and Peter Niblett. *Event Processing in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN 1935182218, 9781935182214.

[21] David C. Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise.* Wiley, Nueva Jersey, Estados Unidos, 2012. ISBN 978-0-470-53485-4.

[22] K. Mani Chandy and W. Roy Schulte. *Event Processing - Designing IT Systems for Agile Companies.* McGraw-Hill, 2010. ISBN 978-0-07-163350-5.

[23] Ruediger Gad, Juan Boubeta-Puig, Martin Kappes, and Inmaculada Medina-Bulo. Hierarchical events for efficient distributed network analysis and surveillance. In *Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups*,

WAS4FI-Mashups '12, page 5–11, Bertinoro, Italy, sep 2012. ACM, ACM. ISBN 978-1-4503-1566-1. doi: http://doi.acm.org/10.1145/2377836.2377839.

[24] Ruediger Gad, Martin Kappes, Juan Boubeta-Puig, and Inmaculada Medina-Bulo. Employing the CEP paradigm for network analysis and surveillance. In *Proceedings of The Ninth Advanced International Conference on Telecommunications*, page 204–210, Rome, Italy, 2013. IARIA, IARIA. ISBN 978-1-61208-279-0.

[25] Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo. *Approaching the Internet of Things through Integrating SOA and Complex Event Processing*, page 304–323. IGI Global book series Advances in Web Technologies and Engineering (AWTE). IGI Global, 2014. ISBN 978-1-4666-5885-1.

[26] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy, and Frank Eliassen. The digihome service-oriented platform. *Software: Practice and Experience*, 2011.

[27] Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo. An approach of early disease detection using CEP and SOA. In *Proceedings of The Third International Conferences on Advanced Service Computing*, pages 143–148, Rome, Italy, sep 2011. IARIA, IARIA. ISBN 978-1-61208-152-6.

[28] Konstantin Vikhorev, Richard Greenough, and Neil Brown. An advanced energy management framework to promote energy awareness, 2013.

[29] Oracle. Oracle event processing. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`, 2014. Accedida septiembre 2015.

[30] StreamBase. Streambase studio. `http://www.streambase.com/products/streambasecep/streambase-studio/`, 2014. Accedida septiembre 2015.

[31] SyBase. Sap sybase event stream processor. `http://www.sybase.com/products/financialservicessolutions/complex-event-processing`, 2014. Accedida septiembre 2015.

[32] JBoss Community. Drools. `http://www.jboss.org/drools/drools-fusion.html`, 2014. Accedida septiembre 2015.

[33] IBM. Operational decision manager. `http://www-03.ibm.com/software/products/en/odm`, 2014. Accedida septiembre 2015.

[34] Darko Anicic and Paul Fodor. Etalis - event-drivent transaction logic inference system. `https://code.google.com/p/etalis/`, 2014. Accedida septiembre 2015.

[35] Software AG. Apama analytics and decisions platform. `http://www.softwareag.com/corporate/products/bigdata/apama$_$analytics/overview/`, 2014. Accedida septiembre 2015.

[36] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012. ISSN 0360-0300.

[37] Michelangelo L Mangano and Timothy J Stelzer. Tools for the simulation of hard hadronic collisions. *Annu. Rev. Nucl. Part. Sci.*, 55:555–588, 2005.

[38] Matt A Dobbs, Stefano Frixione, Eric Laenen, Kirsten Tollefson, H Baer, E Boos, B Cox, R Engel, W Giele, J Huston, et al. Les houches guidebook to monte carlo generators for hadron collider physics. *arXiv preprint hep-ph/0403045*, 2004.

[39] IoT-Analytics. White paper - iot platforms - the central backbone for the internet of things. `https://iot-analytics.com/5-things-know-about-iot-platform/`. Accedida noviembre 2015.

[40] editor ISO/IEC/IEEE. Iso/iec/ieee 29119 2013. systems and software engineering - software testing - part 1: Concepts and definitions, 2013.

[41] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4):465–470, 1982.

[42] John B Goodenough and Susan L Gerhart. Toward a theory of test data selection. *Software Engineering, IEEE Transactions on*, (2):156–173, 1975.

[43] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, (3):236–246, 1980.

[44] John B Goodenough and Susan L Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, (2):156–173, 1975.

[45] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590.

[46] R.G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, SE-3(4):279–290, July 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.231145.

[47] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136.

[48] Martin R Woodward. Mutation testing—its origin and evolution. *Information and Software Technology*, 35(3):163–169, 1993.

[49] A Jefferson Offutt and Ronald H Untch. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century. Kluwer Academic Publishers, Norwell, MA, USA*, pages 34–44, 2001.

[50] Kamel Ayari, Salah Bouktif, and Giuliano Antoniol. Automatic mutation test input data generation via ant colony. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1074–1081, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-697-4. doi: 10.1145/1276958.1277172.

[51] A Jefferson Offutt. *Automatic test data generation*. PhD thesis, Atlanta, GA, USA, 1988.

[52] A Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[53] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.

[54] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.62.

[55] Macario Polo Usaola and Pedro Reales Mateo. Mutation testing cost reduction techniques: A survey. *IEEE Softw.*, 27(3):80–86, May 2010. ISSN 0740-7459. doi: 10.1109/MS.2010.79.

[56] Allen Troy Acree, Jr. *On Mutation.* PhD thesis, Atlanta, GA, USA, 1980. AAI8107280.

[57] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, New Haven, Connecticut, 1980.

[58] Shamaila Hussain. Mutation clustering. *Ms. Th., King's College London, Strand, London*, 2008.

[59] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. A novel method of mutation clustering based on domain analysis. *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09)*, 2009.

[60] A.P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Computer Software and Applications Conference, 1991. COMPSAC '91.,*

Proceedings of the Fifteenth Annual International, pages 604–605, Sep 1991. doi: 10.1109/CMPSAC.1991.170248.

[61] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996. ISSN 1049-331X. doi: 10.1145/227607.227610.

[62] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c†. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[63] A.S. Namin, J.H. Andrews, and D. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 351–360, May 2008. doi: 10.1145/1368088.1368136.

[64] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 435–444. ACM, 2010.

[65] Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Softw. Test. Verif. Reliab.*, 19(2):111–131, June 2009. ISSN 0960-0833. doi: 10.1002/stvr.v19:2.

[66] Yue Jia and Mark Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51 (10):1379–1393, October 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.04.016.

[67] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, and Inmaculada Medina-Bulo. Evolutionary mutation testing. *Information and Software Technology*, 53(10):1108–1123, 2011.

[68] B.J.M. Grun, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 192–199, April 2009. doi: 10.1109/ICSTW.2009.37.

[69] Peter Varhol Gerie Owen. Testing the internet of things. InfoQ Interview by Ben Linders `https://www.infoq.com/news/2014/09/testing-iot`, sep 2014.

[70] P. Giménez, B. Molína, C. E. Palau, and M. Esteve. Swe simulation and testing for the iot. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pages 356–361, Oct 2013. doi: 10.1109/SMC.2013.67.

[71] OWASP. Owasp main page. `https://www.owasp.org/index.php/Main_Page`, 2001.

[72] OWASP. Owasp internet of things project. `https://www.owasp.org/index.php?title=OWASP_Internet_of_Things_Project&setlang=es#tab=Main`, 2015.

[73] João Varajão, Manuela Cunha, Phillip Yetton, Rui Rijo, Isabel Laranjo, Joaquim Macedo, and Alexandre Santos. 4th conference of enterprise information systems – aligning technology, organizations and people (centeris 2012) internet of things for medication control: Service implementation and testing. *Procedia Technology*, 5:777 – 786, 2012. ISSN 2212-0173. doi: http://dx.doi.org/10.1016/j.protcy.2012.09.086.

[74] Brian Bailey. How to cut verification costs for iot. `http://semiengineering.com/how-to-cut-verification-costs-for-iot/`, 2014.

[75] 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.

[76] Jürgen Dunkel, Alberto Fernández, Rubén Ortiz, and Sascha Ossowski. Injecting semantics into event-driven architectures. In *ICEIS (1)*, pages 70–75, 2009.

[77] David Luckham and Roy Schulte. Event processing technical society. *Event Processing Glossary—Version*, 2, 2011.

[78] D Robins. Complex event processing. In *Second International Workshop on Education Technology and Computer Science. Wuhan*, 2010.

[79] David C Luckham and Brian Frasca. Complex event processing in distributed systems. *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford*, 28, 1998.

[80] Louis Perrochon, Stephane Kasriel, and David C Luckham. Managing event processing networks. *Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, USA*, pages 1–62, 1999.

[81] Sungjin Ahn and Daeyoung Kim. Proactive context-aware sensor networks. In *Wireless Sensor Networks*, pages 38–53. Springer, 2006.

[82] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):414–454, 2014.

[83] Elahe Habibi and Seyed-Hassan Mirian-Hosseinabadi. Event-driven web application testing based on model-based mutation testing. *Information and Software Technology*, 67:159–179, 2015.

[84] Johannes Weiss, Peter Mandl, and Alexander Schill. Introducing the qcep-testing system for executable acceptance test driven development of complex event processing applications. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, pages 13–18. ACM, 2013.

[85] Fevzi Belli and Michael Linschulte. Event-driven modeling and testing of web services. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1168–1173. IEEE, 2008.

[86] Lee J White. Regression testing of gui event interactions. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 350–358. IEEE, 1996.

[87] Xun Yuan, Myra B Cohen, and Atif M Memon. Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574, 2011.

[88] Marcelo Mendes, Pedro Bizarro, and Paulo Marques. A framework for performance evaluation of complex event processing systems. In *Proceedings of the second international conference on Distributed event-based systems*, pages 313–316. ACM, 2008.

[89] Marcelo RN Mendes, Pedro Bizarro, and Paulo Marques. A performance study of event processing systems. In *Performance Evaluation and Benchmarking*, pages 221–236. Springer, 2009.

[90] Chunhui Li and Robert Berry. Cepben: a benchmark for complex event processing systems. In *Performance Characterization and Benchmarking*, pages 125–142. Springer, 2013.

[91] Chunhui Li. *Performance management of event processing systems*. PhD thesis, Aston University, 2014.

[92] M Saboor and R Rengasamy. Designing and developing complex event processing applications. *Sapient Global Markets*, 2013.

[93] David C. Luckham. *Event processing for business : organizing the real-time enterprise*. Hoboken, N.J. John Wiley & Sons, 2012. ISBN 978-0-470-53485-4. URL `http://opac.inria.fr/record=b1133466`.

[94] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.

[95] Piyanath Mangkorntong and FethiA. Rabhi. A domain-driven approach for detecting event patterns in e-markets: A case study in financial market surveillance. In Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq, and Claude Godart, editors, *Web Information Systems Engineering – WISE 2007*, volume 4831 of *Lecture Notes in Computer Science*, pages 147–158. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-76992-7. doi: 10.1007/978-3-540-76993-4_13.

[96] Bilgin AVENOöLUa and P Erhan Eren. A complex event processing based framework for intelligent environments. In *Workshop Proceedings of the 9th International Conference on Intelligent Environments*, volume 17, page 12. IOS Press, 2013.

[97] Darko Anicic, Paul Fodor, Nenad Stojanovic, and Roland Stühmer. Computing complex events in an event-driven and logic-based approach. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 36. ACM, 2009.

[98] Darko Anicic. Event processing and stream reasoning with etalis. *PhD, Karlsruher Institut für Technologie (KIT)*, 2011.

[99] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in etalis. *Semantic Web*, 3(4):397–407, 2012.

[100] Darko Anicic. *Event Processing and Stream Reasoning with ETALIS: From Concept to Implementation.* Suedwestdeutscher Verlag fuer Hochschulschriften, Germany, 2012. ISBN 3838131738, 9783838131733.

[101] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[102] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rahul Khandekar, Vipin Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[103] Louis Lovas. A transparent new year. `http://apama.typepad.com/my_weblog/2009/01/a-transparent-new-year.html`, 10-2015.

[104] Louis Lovas. Structured programming in the apama epl. http://apama.typepad.com/my_weblog/2009/02/structured-programming-in-the-apama-epl-.html, 10-2015.

[105] Micro-Research Finland Oy. Event generator. http://www.mrf.fi/index.php/timing-system/71-event-generator. Accedida 03-18-2016.

[106] Starcom Systems. The event generator. https://www.starcomsystems.com/ru/the-event-generator. Accedida 03-18-2016.

[107] Oracle. Introducing the weblogic integration administration console. https://docs.oracle.com/cd/E13214_01/wli/docs85/manage/intro.html. Accedida 03-18-2016.

[108] SV Chekanov. Hepsim: a repository with predictions for high-energy physics experiments. *Advances in High Energy Physics*, 2015, 2015.

[109] ThingSpeak. Thingspeak website. https://thingspeak.com. Accedida 01-24-2016.

[110] Lelylan. Lelylan website. http://www.lelylan.com/. Accedida 02-16-2016.

[111] Particle. Particle website. https://www.particle.io/. Accedida 02-16-2016.

[112] Buglabs. Buglabs website. http://buglabs.net/bugswarm. Accedida 02-25-2016.

[113] Zettajs. Zettajs website. http://www.zettajs.org/. Accedida 02-25-2016.

[114] Grovestreams. Grovestreams website. https://grovestreams.com/index.html. Accedida 02-25-2016.

[115] Plotly. Plotly website. https://plot.ly/. Accedida 02-16-2016.

[116] Sensorcloud. Sensorcloud website. http://sensorcloud.com/. Accedida 02-25-2016.

[117] Xively. Xively website. http://xively.com/. Accedida 02-25-2016.

[118] Carriots. Carriots website. https://www.carriots.com/. Accedida 02-16-2016.

[119] Nodejs. Nodejs website. https://nodejs.org/en/. Accedida 02-25-2016.

[120] atultyodhi geek123 praneethz1994 Anders, niteshvijay. Simple epl query tester. https://java.net/projects/simpleeplquerytester/pages/Home, 2014.

[121] EsperTech. Esper epl online. http://www.esper-epl-tryout.appspot.com/epltryout/index.html, .

[122] Tagomori. Norikra stream processing with sql for everybody. `https://norikra.github.io/index.html`, 2014.

[123] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381.

[124] Hiralal Agrawal, Richard A DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. 1999.

[125] Márcio Eduardo Delamaro and José Carlos Maldonado. Proteum-A tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.

[126] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Softw. Pract. Exper.*, 21(7):685–718, June 1991. ISSN 0038-0644. doi: 10.1002/spe.4380210704.

[127] 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.

[128] Sunwoo Kim, John A. Clark, and John A. McDermid. The rigorous generation of java mutation operators using hazop. In *IN 12TH INTERNATIONAL CONFERENCE ON SOFTWARE & SYSTEMS ENGINEERING AND THEIR APPLICATIONS (ICSSEA'99*, 1999.

[129] Roger T Alexander, James M Bieman, Sudipto Ghosh, and Bixia Ji. Mutation of java objects. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 341–351. IEEE, 2002.

[130] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 352–363. IEEE, 2002.

[131] 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.

[132] Jeremy S Bradbury, James R Cordy, and Juergen Dingel. Mutation operators for concurrent java (j2se 5.0). In *Mutation Analysis, 2006. Second Workshop on*, pages 11–11. IEEE, 2006.

[133] P.R. Mateo and M.P. Usaola. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 646–649, Sept 2012. doi: 10.1109/ICSM.2012.6405344.

[134] Pedro Reales Mateo and Macario Polo Usaola. Parallel mutation testing. *Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[135] Javier Tuya, Ma José Suárez-Cabal, and Claudio de la Riva. Mutating database queries. *Inf. Softw. Technol.*, 49(4):398–417, April 2007. ISSN 0950-5849. doi: 10.1016/j.infsof.2006.06.009.

[136] A Derezińska. Specification of mutation operators specialized for c# code. *Inst. of Computer Science Research Raport*, 2(05).

[137] Anna Derezińska. Advanced mutation operators applicable in c# programs. In *Software Engineering Techniques: Design for Quality*, pages 283–288. Springer US, 2007.

[138] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez, Antonio García-Domínguez, and Francisco Palomo-Lozano. Class mutation operators for c++ object-oriented systems. *annals of telecommunications - annales des télécommunications*, 70(3):137–148, 2015. ISSN 1958-9395. doi: 10.1007/s12243-014-0445-4.

[139] Marcio E. Delamaro, Jos'e C. Maldonado, and Aditya P. Mathur. Integration testing using interface mutations. In *In Proceedings of International Symposium on Software Reliability Engineering (ISSRE '96*, pages 112–121. Society Press, 1996.

[140] Sudipto Ghosh and Aditya P. Mathur. Interface mutation. *Software Testing, Verification and Reliability*, 11(4):227–247, 2001.

[141] P. Ammann and P.E. Black. Abstracting formal specifications to generate software tests via model checking. In *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, volume 2, pages 10.A.6–1–10.A.6–10 vol.2, 1999. doi: 10.1109/DASC.1999.822091.

[142] Simone Do Rocio Senger De Souza, José Carlos Maldonado, Sandra Camargo Pinto Ferraz Fabbri, and Wanderley Lopes De Souza. Mutation testing applied to estelle specifications. *Software Quality Control*, 8(4):285–301, December 1999. ISSN 0963-9314. doi: 10.1023/A:1008978021407.

[143] ISO/IEC 9074 1989. Estelle – a formal description technique based on an extended state transition model, 1989.

[144] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, September 2004. ISSN 0163-5948. doi: 10.1145/1022494.1022529.

[145] Wuzhi Xu, Jeff Offutt, and Juan Luo. Testing web services by xml perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.

[146] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Mutation operators for WS-BPEL 2.0. In *ICSSEA'08: 21th International Conference on Software & Systems Engineering and their Applications*, 2008.

[147] Salem F Adra and Phil McMinn. Mutation operators for agent-based models. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 151–156. IEEE, 2010.

[148] Vilas Jagannath, Milos Gligoric, Steven Lauterburg, Darko Marinov, and Gul Agha. Mutation operators for actor systems. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 157–162. IEEE, 2010.

[149] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. Xacmut: Xacml 2.0 mutants generator. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 28–33. IEEE, 2013.

[150] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 667–676. ACM, 2007.

[151] Tejeddine Mouelhi, Franck Fleurey, and Benoit Baudry. A generic metamodel for security policies mutation. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 278–286. IEEE, 2008.

[152] 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.

[153] 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.

[154] 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.

[155] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[156] L. Gutiérrez-Madroñal, J.J. Dominguez-Jimenez, and I. Medina-Bulo. Generación automática de eventos de prueba para sistemas de i.o.t. In *XXI Jornadas de Ingeniería del Software y Bases de Datos (JISBD - SISTEDES 2016), Salamanca, Spain*, 2016.

[157] L. Gutiérrez-Madroñal, J.J. Dominguez-Jimenez, and I. Medina-Bulo. Iot-teg: Event generator system. *Journal of Systems and Software JSS. Special Issue on Software Reliability Engineering, (en primera revisión para ser publicado)*, 2016.

[158] 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. doi: 10.1109/ISSRE.2012.20.

[159] 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.

[160] EsperTech. Espertech self-service terminal. `http://www.espertech.com/esper/release-5.2.0/esper-reference/html/examples.html#examples-terminalsvc-J2EE`, . Accedida 01-07-2016.

[161] EsperTech. Gc:esperepl2grammarlexer. `http://grepcode.com/file/repo1.maven.org/maven2/com.espertech/esper/4.9.0/com/espertech/esper/epl/generated/EsperEPL2GrammarLexer.java?av=f`, . Accedida 10-25-2014.

[162] EsperTech. Espertech transaction 3-event challenge. `http://www.espertech.com/esper/release-5.2.0/esper-reference/html/examples.html#examples-transaction-3-event-challenge`, . Accedida 01-07-2016.

[163] Daniel Jesús Rosa Gallardo. Sistema sostenible para la recolección de basura en smart cities con procesamiento de eventos complejos. Universidad de Cádiz, Puerto Real, 10-02-2016.

[164] Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo. Approaching the Internet of Things through Integrating SOA and Complex Event Processing.

In Zhaohao Sun and John Yearwood, editors, *Handbook of Research on Demand-Driven Web Services: Theory, Technologies, and Applications*, IGI Global book series Advances in Web Technologies and Engineering (AWTE), pages 304–323. IGI Global, March 2014. ISBN 978-1-4666-5885-1.

[165] Ruediger Gad. Distributed event-driven network monitoring evaluation prototype (denmevap). `https://github.com/fg-netzwerksicherheit/DENMEvaP`. Accedida 01-07-2016.

[166] J.J. Dominguez Jiménez. *Búsquedas genéticas: Métodos de optimización global y optimización combinatoria*. PhD thesis, Escuela Superior de Ingeniería, Universidad de Cádiz, 2008.

[167] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, and Inmaculada Medina-Bulo. Gamera: an automatic mutant generation system for ws-bpel compositions. In *Web Services, 2009. ECOWS'09. Seventh IEEE European Conference on*, pages 97–106. IEEE, 2009.

[168] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, and Inmaculada Medina-Bulo. Evolutionary mutation testing. *Information and Software Technology*, 53(10):1108–1123, 2011.

[169] Ettercap. Ettercap website. `https://ettercap.github.io/ettercap/index.html`. Accedida 02-16-2016.

[170] Shachar Shemesh (thesun). sshpass website. `https://sourceforge.net/projects/sshpass/`. Accedida 06-23-2016.

[171] Herbert Haas. Mausezahn website. `http://www.perihel.at/sec/mz/mzguide.html`. Accedida 02-16-2016.