Prueba de mutaciones sobre consultas de procesamiento de eventos en aplicaciones en tiempo real

Lorena Gutiérrez Madroñal, Juan José Domínguez Jiménez, e Inmaculada Medina Bulo

Dpto. de Lenguajes y Sistemas Informáticos
Escuela Superior de Ingeniería, Universidad de Cádiz
11002 Cádiz
{lorena.gutierrez, juanjose.dominguez, inmaculada.medina}@uca.es

Abstract. La prueba de mutaciones es una técnica de prueba de software que ha sido usada con éxito en la prueba de lenguajes de programación clásicos. Sin embargo, no se ha empleado en la prueba de aplicaciones en tiempo real que procesen un gran número de flujos de eventos y en las que se realicen consultas de procesos de eventos. Un error mientras se está diseñando la consulta para procesar un flujo de eventos, puede ocasionar un comportamiento anómalo del sistema. En este trabajo, proponemos la prueba de mutaciones para controlar las consultas en aplicaciones en tiempo real realizadas en el lenguaje EPL de procesamiento de eventos. Se presentan y definen los operadores de mutación para EPL, comparándolos con los operadores de mutación del lenguaje SQL. Definimos los criterios necesarios para matar mutantes en EPL. Finalmente, se presenta una arquitectura para la generación automática de dichos mutantes.

Keywords: Lenguaje para el Procesamiento de Eventos, Operadores de mutación para EPL, Aplicaciones en tiempo real, Casos de prueba adecuados

1 Introducción

La prueba de mutaciones es una técnica de prueba que introduce fallos simples en el programa a probar que puede medir la efectividad de un conjunto de casos de prueba para la localización de esos fallos. Los *mutantes* son el resultado de introducir esos fallos, aplicando *operadores de mutación*, en el programa a probar. Cada operador de mutación se corresponde con una categoría de error típico que el desarrollador podría cometer. Los pequeños cambios sintácticos que contienen los mutantes con respecto al programa original deben ser detectados por el conjunto de casos de prueba.

Si un programa contiene la instrucción x = 60 y disponemos de operadores de mutación sobre los operadores relacionales (se cambia un operador relacional por otro), el mutante resultante podría tener como instrucción, por ejemplo, $x \neq 60$. Si un caso de prueba es capaz de diferenciar entre el programa original y el mutante, es decir, sus salidas son diferentes, se dice que el caso de prueba mata al mutante. Si ningún caso de prueba es capaz de diferenciar las salidas de ambos, el mutante sigue vivo.

A. Ruíz, L. Iribarne (Eds.): JISBD'2012, pp. 369-382, ISBN:978-84-15487-28-9.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Una de las dificultades de aplicar la prueba de mutaciones es la existencia de *mutantes equivalentes*. Éstos tienen el mismo comportamiento que el programa original, es decir, siempre producen la misma salida, por lo que no pueden ser diferenciados del programa original. No deben confundirse estos mutantes con los mutantes que sobreviven porque el conjunto de casos de prueba no es adecuado para detectarlos.

La prueba de mutaciones ha demostrado ser efectiva en diversos lenguajes [1 – 8, 18] para conseguir la calidad de un software implementado. Desafortunadamente, no existe suficiente información sobre la prueba de mutaciones en aplicaciones de tiempo real [22 - 25] que procesen un gran número de flujos de eventos. Éstas tienen procesos de eventos complejos que aceptan como entrada flujos de eventos que luego redirigen a "listeners" que se basan en reglas o en eventos que se activan comparando expresiones de patrones. Estas consultas son parcialmente similares a las consultas tradicionales del lenguaje de programación SQL, pero a diferencia de éstas, proporcionan medios para la expresión de algunas características (por ejemplo, expresión de patrones). Un error en el diseño de este tipo de consultas puede provocar desde un anormal comportamiento del programa a la pérdida de muchas oportunidades de negocio. Además, las consultas, al poder ser generadas dinámicamente, pueden disponer de entradas que no han sido filtradas apropiadamente. Esto hace necesario que se preste atención a las pruebas en este tipo de consultas en aplicaciones de tiempo real.

En este artículo proponemos un enfoque basado en la prueba de mutaciones para evaluar las consultas de aplicaciones en tiempo real realizadas en el lenguaje EPL (Event Processing Language) [9] de procesamiento de eventos. A pesar de que EPL es similar al lenguaje SQL (Structured Query Language), existe un número considerable de diferencias entre ambos. Hemos definido y diseñado un conjunto de operadores de mutación para este lenguaje, así como los criterios que se deben emplear para considerar a un mutante muerto. Los operadores que proponemos introducen fallos en las consultas EPL (expresiones de patrones, ventanas de longitud y tiempo, proceso de eventos por lotes), así como en las cláusulas, operadores y en el uso de la palabra reservada NULL. Además también presentamos una arquitectura para generar de manera automática los mutantes para EPL.

Los operadores propuestos y los criterios para matar los mutantes que se presentan pueden ser usados para controlar la calidad de algunos de los aspectos más comunes de las aplicaciones en tiempo real, como errores o retrasos en la notificación de eventos, e inesperadas e incorrectas secuencias de eventos.

Las siguientes secciones se organizan de la siguiente forma: en la Sección 2 se presenta el lenguaje EPL y las diferencias y similitudes con SQL. En la Sección 3 se definen los operadores de mutación, los criterios para matar los mutantes y algunos ejemplos. La Sección 4 describe el framework para EPL. La Sección 5 presenta los trabajos relacionados y la Sección 6 las conclusiones y el trabajo futuro.

2 Esper, EPL y SQL Database

El motor de Esper ha sido desarrollado para satisfacer los requisitos de las aplicaciones que analizan y reaccionan a los eventos. Algunos ejemplos típicos de este tipo de

aplicaciones son la gestión y automatización de procesos de negocio, las finanzas, la monitorización de aplicaciones y redes y las aplicaciones de sensores de redes. Lo que tienen en común todas estas aplicaciones es que necesitan procesar eventos o mensajes en tiempo real, no en un tiempo cercano al real. Esto significa que a veces se tienen que procesar eventos complejos (CEP) y analizar el flujo de estos eventos. Las consideraciones claves para este tipo de aplicaciones son: el rendimiento, la latencia y la complejidad de la lógica necesaria.

Esper es un motor implementado en Java que permite especificar y ejecutar consultas EPL [9]. Esto permite al programador implementar aplicaciones en tiempo real capaces de enviar y escuchar a la vez continuos flujos de eventos. Como los flujos de eventos están disponibles en la aplicación, el programador tiene la opción de implementar filtros para escuchar eventos específicos de interés, relacionados con eventos del mundo real.

Existen diferencias entre Esper y los sistemas de bases de datos tradicionales. Primero, Esper almacena consultas y las bases de datos almacenan datos. Segundo, mientras que una sentencia de SQL puede ser invocada por una aplicación (dependiendo de funcionalidades específicas), una sentencia EPL se ejecuta de inmediato y de forma continua tras haber sido creada durante la ejecución de la aplicación. Además las consultas EPL también se ejecutan cuando se reciben eventos predefinidos en la aplicación o cuando se dispara el temporizador. Tercero, en Esper, el tiempo es una propiedad, a diferencia de las bases de datos que es un tipo de dato. Luego, eventos, flujos de eventos y consultas EPL, pueden ser consideradas análogas a filas, tablas y consultas SQL, respectivamente.

El lenguaje EPL es un lenguaje similar a SQL con las cláusulas: SELECT, FROM, WHERE, GROUP BY, HAVING y ORDER BY. Las cadenas reemplazan a las tablas como fuente de datos y los eventos reemplazan a las filas como unidad básica de datos [9]. La figura 1 muestra un ejemplo de una consulta EPL que informa de un evento si el precio de compra de un artículo supera los 5€, y el precio medio de todos los artículos comprados supera los 8€ en un plazo de 10 minutos.

```
select * from
pattern [every (Order.price > 5.0 -> avg (Order.price) > 8.0))] where timer:within (10 min)]
```

Fig. 1. Ejemplo de un patron de eventos en EPL.

Esper ofrece dos métodos para procesar los eventos: patrones de eventos y consultas de eventos en cadena. Para el análisis de las aplicaciones CEP (aplicaciones que procesan eventos complejos), las consultas de eventos en cadena de Esper son la mejor opción a usar con estos eventos encadenados. Éstas siguen la sintaxis de EPL y proporcionan: ventanas, agregaciones, uniones y análisis de funciones.

3 Operadores propuestos para EPL y criterios para matar a un mutante en EPL

En esta sección se proponen los operadores de mutación para evaluar la calidad de los flujos de eventos que proporcionan las consultas EPL. Los operadores están diseñados

para introducir fallos en determinadas características: en expresiones de patrones, en ventanas de longitud y tiempo variable, y en la capacidad de procesamiento por lotes (Véanse de 3.2 a 3.5). Como EPL también es vulnerable a los ataques de entradas SQL [26], también se diseñan este tipo de operadores (Véase 3.6). Dado que EPL está basado en SQL, además desarrollamos operadores análogos a los ya existentes para SQL [7, 11, 12] (Véanse de 3.7 a 3.9). Del mismo modo se indican los operadores que generan mutantes equivalentes. Pero antes de presentarlos, destacaremos los criterios para matar los mutantes que se van a generar según estos operadores.

3.1 Criterios para matar a un mutante en EPL

En general, las aplicaciones en tiempo real implementan métodos para manejar los eventos e informan sobre ellos. Asumimos que estos métodos están bien implementados mientras que se aplican nuestros operadores de mutación para evaluar si son adecuados los casos de prueba. Vamos a comparar el número total de los eventos obtenidos por la consulta original (O), y por la consulta mutada (M) para decidir si el mutante está vivo o muerto, véase tabla 1. Un mutante está muerto si el número de eventos obtenidos no es el mismo que el del original, siendo estos eventos obtenidos los mismos para ambas consultas.

Categorías	**Operadores**	**Criterios para matar**
Pattern expression (PEP)	RREP, CEOP, RLOP	Número de eventos entre *O* y *M*
	RGEP, OEDIP,OEDDP	Latencia entre *O* y *M*
Window of length (WOL)	LINC, LDEC	Número de eventos entre *O* y *M*
Window of time (WOT)	TINC, TDEC, TRUN	Latencia entre *O* y *M*
Batch processing of event (BOE)	BATL	Número de eventos entre *O* y *M*
	BATT	Latencia entre *O* y *M*
SQL injection attack (SQIJ)	WCRW, WCNG, WCFD	Número de eventos entre *O* y *M*
EPL clause (EPLC)	EAGR, ESEL, EGRU, EJOI, EORD, ESUB	Número de eventos entre *O* y *M*
	ESIR	Número de eventos entre *O* y *M* Latencia entre *O* y *M*
Operator replacement (ORP)	EBTW, EABS, ELKE, EAOR, EROR, EUOI	Número de eventos entre *O* y *M* Latencia entre *O* y *M*.
Null mutation operator (NOP)	ENLF, ENLI, ENLO	Número de eventos entre *O* y *M*

Table 1. Operadores y criterios para matar los mutantes

Algunos de los operadores (WCNG y WCFD), necesitan un criterio diferente para matar los mutantes. Para ellos, un mutante está muerto si el número de eventos obtenidos por la consulta original y la mutada coincide. Con los operadores que introducen fallos en parámetros de tiempo, comparamos la latencia entre la consulta original y la mutada. La latencia de la ejecución de una consulta se calcula basándose en la diferencia entre el tiempo de suministro de las entradas de eventos necesarios y el momento en el que se activa un método de devolución. Luego el retraso de la notifi-

cación de un evento entre una consulta original y una mutada es un criterio para matar un mutante.

3.2 Operadores de mutación para expresiones de patrones

Las consultas EPL pueden incluir expresiones de patrones que se comparan con uno o muchos flujos de eventos de entrada y dependiendo del tiempo, podrían anidarse. Se proponen 6 operadores (véase tabla2):

1. RREP - Remove repetition in pattern expression: Se elimina el operador "*every*" de una expresión. Este permite una comprobación continua de los patrones, y eliminándolo, conseguimos que solamente se haga una comprobación.
2. CEOP - Change event order in pattern expression: Se cambia el orden de la expresión. El orden de los eventos viene definido con el operador "→". Se intercambian los eventos a ambos lados del operador "→".
3. RLOP - Replace logical operator in pattern expression: Se sustituyen, cada uno de los operadores lógicos "*and*" y "*or*" por el otro.
4. RGEP - Remove guard expression: Se elimina la expresión "*guard*" presente en las condiciones WHERE. Estas controlan el ciclo de vida de las expresiones de patrones. Las consultas mutadas, modifican el ciclo de vida del patrón.
5. OEDIP - Observer expression's time delay increment: Se incrementa en uno el valor del tiempo en el observador del patrón. En "*timer:at*" como en "*timer:interval*".
6. OEDDP - Observer expression's time delay decrement: Igual que el operador OEDIP, pero se decrementa en uno el valor.

Nombre	Consulta original	Consulta mutada
RREP	every (p=Order(price>5))	(p=Order(price>5))
CEOP	(a=Order (price > 5 and symbol= "AAPL") → o = Order (price > 8 and symbol="ORNG"))	(o = Order (price > 8 and symbol ="ORNG") → a=Order (price > 5 and symbol= "AAPL"))
RLOP	Order (price > 5 and symbol="AAPL")	Order (price > 5 or symbol="AAPL")
RGEP	select avg(price) from Order where timer.within (10 sec)	select avg(price) from Order
OEDIP	every timer:interval (20 sec)	every timer:interval (21 sec)
OEDDP	every timer:interval (20 sec)	every timer:interval (19 sec)

Table 2. Ejemplos de aplicación de los operadores en expresiones de patrones.

3.3 Operadores de mutación para ventanas de longitud variable

Se pueden incluir ventanas de longitud variable, que indiquen al motor de búsqueda los N últimos eventos a mantener. Se proponen 2 operadores (véase tabla 3).

1. LINC – Increase data window length by one: Una ventana de longitud indica al motor de búsqueda que solo ha de mantener los N últimos eventos para un flujo. El operador LINC, incrementa la longitud de la ventana en uno.

2. LDEC – Decrease data window length by one: Este operador reduce el valor de la longitud de la ventana en uno.

Nombre	Consulta original	Consulta mutada
LINC	select * fromOrder.win:length(5)	select * from Order.win:length (6)
LDEC	select * from Order.win:length (5)	select * from Order.win:length (4)

Table 3. Ejemplos de aplicación de los operadores para ventanas de longitud variable.

3.4 Operadores de mutación para ventanas de tiempo variable

Las ventanas de tiempo variable de EPL permiten restringir el número de eventos a procesar por consulta. Se proponen tres operadores (véase tabla 4).

1. TINC – Increase time window by one: Una ventana de tiempo es una ventana que varía según el tiempo del sistema. Esto permite al programador restringir el número de eventos a procesar por consulta. El operador TINC, incrementa el tiempo especificado en la ventana en una unidad.
2. TDEC – Decrease time window by one: El operador TDEC decrementa el tiempo de la ventana en una unidad.
3. TRUN – Replace timer unit: El operador reemplaza la unidad de tiempo especificada por otra. El conjunto de unidades de tiempo son: milisegundos, segundos, minutos, horas y días.

Nombre	Consulta original	Consulta mutada
TINC	select avg(price) from Order.win:time (4 sec)	select avg(price) from Order.win:time (5 sec)
TDEC	select avg(price) from Order.win:time (4 sec)	select avg(price) from Order.win:time (3 sec)
TRUN	select avg(price) from Order.win:time (4 min)	select avg(price) from Order.win:time (4 sec)

Table 4. Ejemplos de aplicación de los operadores para ventanas de tiempo variable.

3.5 Operadores de mutación para el procesamiento de eventos por lotes

Las consultas EPL pueden incluir el procesamiento de eventos por lotes basados en el tiempo y la longitud de las ventanas. Se proponen dos operadores (véase tabla 5):

4. BATL – Replace batch processing length with ordinary window length: La función "*win:length_batch*" nos permite obtener los datos de un número específico de eventos. Una vez que se obtienen los valores, el evento que está escuchando, obtiene esta información. El operador reemplaza "*win:length_batch*" por "*win:length*".
5. BATT – Replace batch processing time with ordinary window time: La función "*win:time_batch*" nos permite obtener los datos de un específico periodo de tiem-

po. Una vez que se obtienen los valores, el evento que está escuchando, obtiene esta información. El operador reemplaza "*win:time_batch*" por "*win_time*".

Nombre	Consulta original	Consulta mutada
BATL	select * from Order.win:length_batch(5)	select * from Order.win:length (5)
BATT	select * from Order.win:time_batch(5)	select * from Order.win:time(5)

Table 5. Ejemplos de aplicación de los operadores para el procesamiento de eventos por lotes.

3.6 Operadores de mutación para entradas SQL

Para las consultas EPL que sufran ataques de entradas SQL, proponemos tres operadores (véase tabla 6):

1. WCRW - Remove SQL Where Conditions: Eliminación de condiciones WHERE. Esto nos permite generar entradas maliciosas relacionadas con ataques SQL. El mutante muere si observamos resultados similares entre la consulta original y la mutada (ataque tautológico [10]).
2. WCNG - Negation of Expression in Where Conditions: Negación de la expresión WHERE. La consulta mutada muere si al pasar el caso de prueba, el número de eventos no coincide con el de la consulta original.
3. WCFD - Prepend 'false and' after the where keyword: Antepone "*false and*" en las condiciones WHERE. La consulta mutada muere si al pasar un caso de prueba genera los mismos eventos que la consulta original.

Nombre	Consulta original	Consulta mutada	Salida (O)	Salida (M)
WCRW	select * from Order (symbol='' or 1=1) where timer.within(10 sec)	select * from Order (symbol='' or 1=1)	(1, AAPL, 5), (2, ORNG, 2), (3, ORNG, 6), (4, AAPL, 2)	(1, AAPL, 5), (2, ORNG, 2), (3, ORNG, 6), (4, AAPL, 2)
WCNG	select * from Order (symbol='' or 1=1) where timer.within(10 sec)	select * from Order (not symbol='' or 1=1) where timer.within(10 sec)	(1, AAPL, 5), (2, ORNG, 2), (3, ORNG, 6), (4, AAPL, 2)	Nada
WCFD	select * from Order (symbol='' or 1=1) where timer.within(10 sec)	select * from Order (symbol='' or 1=1) false and where timer.within(10 sec)	(1, AAPL, 5), (2, ORNG, 2), (3, ORNG, 6), (4, AAPL, 2)	(1, AAPL, 5), (2, ORNG, 2), (3, ORNG, 6), (4, AAPL, 2)

Table 6. Ejemplos de aplicación de los operadores de entradas SQL.

3.7 Operadores de mutación para cláusulas EPL

1. EAGR - Agregate functions: Cada función de agregación es reemplazada por otra, tanto si aparecen tras SELECT como en HAVING. Las funciones de agregación en SQL son: *"min", "max", "avg(distinct)", "sum", "sum(distinct)", "count", "count(distinct)"*. Los cambios tienen que tener en cuenta el tipo de los argumen-

tos. Las funciones de agregación en EPL: *"sum", "avg", "count", "max", "min", "median", "stddev", "avedev"*. Según la sintaxis de EPL (1):

Aggregate-function ([all | distinct] expression) (1)

Se incluyen en los cambios anteponer o quitar las palabras claves "*all*", "*distinct*".

2. ESIR - Single-row functions: Cada función single-row es reemplazada por otra (de tipo compatible). Las funciones "single-row": "*case", "cast", "coalesce", "current_timestamp", "exists", "instanceof", "max", "min", "prev", "prior", "prevwindow", "prevcount"* y *"typeof"*. Ejemplos de aplicación de este operador se puede ver en la tabla 7.

Consulta original	**Consulta mutada**
select prev(2, price) from Order retain 10 events	select prior(2, price) from Order retain 10 events
select prevwindow(price) from Trade.win:length(10)	select prevcount(price) from Trade.win:length(10)

Table 7. Ejemplo de aplicación del operador ESIR

3. GRU - Groupings. Eliminación de expresiones GROUP BY: Si la expresión se encuentra en el SELECT o en ORDER BY, tiene que incluirse en una función de agregación para evitar consultas sintácticamente incorrectas. Las funciones de agregación son: *max* y *min*. En el caso de que sólo exista una expresión GROUP BY, la cláusula entera se elimina.
4. EJOI - JOIN clause: Cada palabra clave: "*inner joing", "left outer join", "right outer join", "full outer join", "cross join"*, es reemplazada por otra. En EPL no existe la palabra reservada *"cross join"*.
5. EORD - Ordering of the result set: Las expresiones ORDER BY son eliminadas. Si existe solamente una expresión, se elimina al completo. Cada par de expresiones adjuntas ORDER BY, es intercambiada, al igual que se intercambian las palabras clave "*asc*" y "*desc*". En el caso de no existir una palabra clave, se añade "*desc*".
6. ESEL - Select clause: Intercambia las palabras clave SELECT y SELECT DISTINCT. En EPL además se incluyen, según su sintaxis (2), los intercambios de las palabras claves: *istream* (eventos de inserción), *rstream* (eventos de eliminación) y *irstream* (eventos de inserción y eliminación).

select [istream | irstream | rstream] [distinct]* | expression_list… (2)

Ejemplos de aplicación de este operador se puede ver en la tabla 8. Este operador genera mutantes equivalentes cuando hablamos de la mutación con la palabra clave istream, palabra clave por defecto. Si una consulta no tiene ninguna de las palabras claves istream, rstream e irstream, los eventos son de inserción. Luego los mutantes generados son equivalentes a la consulta original.

Consulta original	Consulta mutada
select rstream * from Order.win:time(30 sec)	select istream * from Order.win:time(30 sec)
select distinct price from Order	select rstream distinct price from Order

Table 8. Ejemplo de aplicación del operador ESEL

7. SUB - Subquery predicates: Considerando un predicado que normalmente utiliza subconsultas como *"e ℜ k(Q)"*, siendo *"e"* el valor de la fila (atributo o expresión), *"ℜ"*representa un operador relacional, *"k"* es la palabra clave que representa el predicado y *"Q"* la subconsulta. Dependiendo de *"k"*, encontramos tres tipos de predicados:

   - Tipo 1 ($k \in$ {all, any, some}): De la forma *e ℜ k(Q)*
   - Tipo 2 ($k \in$ {in, not in}): De la forma *ek(Q)*
   - Tipo 3 ($k \in$ {exists, no exists}): De la forma *k(Q)*

   Cada palabra clave es reemplazada por otra del mismo tipo, evitando cambios equivalentes. Los predicados de tipo 1, son cambiados a predicados de tipo 2 y 3. Y los de tipo 2 son cambiados a predicados de tipo 1 (combinando todos los operadores relacionales) y 3.

3.8 Operadores de mutación de reemplazamiento.

1. EABS - Absolute Value Insertion: Cada expresión aritmética o referencia a un número "*e"*, es reemplazada por "*abs(e)"* y "*-abs(e)"*. No se realizan estos cambios si "*e"* se encuentra en expresiones GROUP BY, ORDER BY o, dentro de subconsultas EXISTS, en SELECT. Este operador genera mutantes equivalentes, ya que tanto si el número *e* es positivo, como negativo, el reemplazo por *abs(e)*, o *–abs(e),* respectivamente, genera un mutante equivalente con respecto a la consulta original.
2. EAOR - Arithmetic operator replacement: Cada operador aritmético *"=", "-", "*", "/", "%"* es reemplazado por otro.
3. EBTW - Between predícate: Cada condición "*a between x and y*" es reemplazado por "$a > x$ *and* $a \leq y$", y por "$a \geq x$ *and* $a < y$". Ejemplos de aplicación de este operador se puede ver en la tabla 9.

Consulta original	Consulta mutada
select * from Order where price between 50 and 60	select * from Order where price > 50 and price <= 60
select * from Order where price between 50 and 60	select * from Order where price >= 50 and price < 60

Table 9. Ejemplo de aplicación del operador EBTW

4. ELKE - Like predicate: Modifica las condiciones en expresiones LIKE. Se eliminan, reemplazan o añaden caracteres especiales: "*%", "_}", "...)"*. Los caracteres especiales en EPL son: "*_"* y "*%"*.

5. EROR - Relational operator replacement: Cada operador relacional *"="*, *"<>"*, *"<"*, *"≤"* *">"*, *"≥"* es reemplazado por otro.
6. EUOI - Unary operator insertion. Cada expresión aritmética o referencia a un número "*e*" es reemplazado por *"-e"*, *"e + 1"* y "*e – 1*".

3.9 Operadores de mutación para la palabra reservada NULL

1. ENLF - Null check predicates: Intercambia los predicados "*is null*" y" *is not null*".
2. ENLI - Include nulls: Establece el valor de la condición a "true" donde hay un valor NULL. Cada atributo *a* en una condición *C* es reemplazado por "*C or a is null*".
3. ENLO - Other nulls. Cada atributo a en condición *C* es reemplazado por: "*not C or a is null"*, *"a is null"* y *"a is not null"*.

4 Arquitectura para la prueba de mutaciones en EPL

Recientemente en el grupo de investigación UCASE1, hemos desarrollado la herramienta MuBPEL [8], para realizar la prueba de mutaciones en el lenguaje Web Services Business Process Execution 2.0. (WSBPEL). MuBPEL se utiliza para evaluar la calidad de un conjunto de casos de prueba, comprobando si el mutante puede ser considerado diferente al programa original.

Los operadores de mutación diseñados anteriormente (sección 3) se probarán en una herramienta que automatice la generación de mutantes en EPL cuyo esquema (Figura 2) es similar al de MuBPEL.

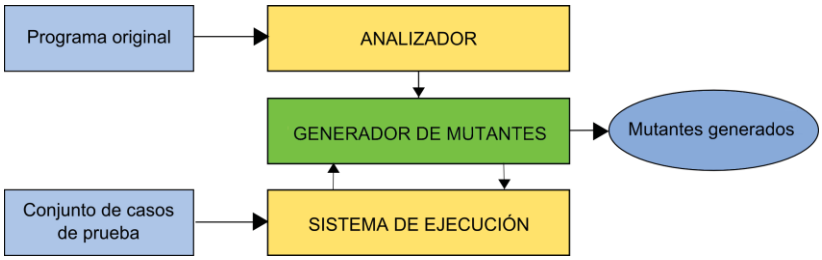

Fig. 2. Arquitectura de la herramienta para la generación de mutantes EPL

Esta arquitectura consta de tres partes: análisis del programa original, generación de mutantes y ejecución de los mutantes. Se recibe como entrada el lenguaje original y un conjunto de casos de prueba. El analizador, toma el programa original y determina los operadores de mutación que se pueden aplicar. Esta información la recibe el segundo componente, el generador de mutantes. Finalmente el sistema de ejecución determinará, con un conjunto de casos de prueba suministrado por el usuario, qué mutantes están vivos o muertos, y con ello podrá comprobar la calidad de los mutantes generados. La calidad de los casos de prueba es evaluada basándose en la puntua-

[1] https://neptuno.uca.es/redmine/

ción de mutación (relación entre el número de mutantes no equivalentes y el número de mutantes muertos). Esta arquitectura acepta dos tipos de casos de prueba: flujos de eventos de datos y entradas que pueden ser parte de consultas dinámicas en EPL. Para la validación de los operadores de mutación de EPL aplicaremos en MuEPL un conjunto de casos de prueba que abarque todos los operadores definidos. Actualmente no existe un conjunto de casos de prueba de consultas EPL para la prueba de mutaciones. A pesar de encontrarse disponibles en la web de Esper consultas EPL [9], éstas utilizan eventos de tipos distintos, siendo inconsistentes y no siendo útiles para un conjunto de casos de prueba. Por esto, y tras estudiar nuestros operadores, se ha tenido que elaborar un nuevo conjunto de casos de prueba que contempla las características del lenguaje EPL2.

5 Trabajos relacionados

Comenzamos esta sección con algunos trabajos relacionados que emplean operadores de mutación para probar aplicaciones en tiempo real detallando más aquellos relacionados con nuestro trabajo. Mark et al [22] generan mutantes utilizando documentos gráficos de aplicaciones en tiempo real para generar casos de prueba adecuados de tal modo que pueden solventar algunos de los problemas de las aplicaciones en tiempo real. Los mutantes se generan en la transición relacionada con los diagramas de estado de modo que las relaciones se realizan con una mayor demora. Nosotros proponemos operadores que aumentan y disminuyen el tiempo de retardo en la ejecución de la consulta. Sin embargo, nuestro objetivo es generar casos de prueba capaces de revelar fallos en los datos y brechas en la seguridad. Geist et al. [23] implementan un conjunto de operadores de mutación para generar casos de prueba de tal modo que las distribuciones de software en tiempo de ejecución se diversifican. Se introducen faltas en las entradas de los datos de los sensores para permitir la diversidad en la ejecución del software en tiempo real. Del mismo modo nosotros diseñamos operadores para modificar las entradas de los usuarios, por lo que las entradas maliciosas y los flujos de eventos diversos pueden revelar fallos en la seguridad y el rendimiento de las aplicaciones en tiempo real. Nilson et al. [24] desarrollan tres operadores de mutación para comprobar el control de los plazos del sistema en tiempo real. Estos permiten un control para implementar casos de prueba por lo que la violación de los plazos puede verse fácilmente. Del mismo modo Sung et al. [25] implementa operadores de mutación para probar la interfaz entre sistemas en tiempo real autómatas. En vez de tanto esfuerzo, nosotros implementamos operadores de mutación para generar casos de prueba, que revelarán cualquiera de estos fallos.

Ahora vamos a hablar sobre algunos trabajos relacionados con las pruebas de mutaciones. Varios trabajos presentan el desarrollo de herramientas para automatizar la generación de mutantes: Mothra [4] para Fortran, MuJava [5] para Java, Proteum [6] para C, SQLMutation [7] para SQL. Su principal característica es que generan au-

[2] https://neptuno.uca.es/svn/sources-fm/trunk/src/muepl/src/test/java/sourcefiles/FilterQueries/epl-benchmark-May2012

tomáticamente todos los posibles mutantes para todos los operadores de mutación proporcionados, sin realizar ninguna mutación selectiva en la generación.

Uno de los principales inconvenientes de la prueba de mutaciones es el coste computacional que supone la ejecución de la gran cantidad de mutantes que se generan a partir del programa original. Para reducir el tiempo de ejecución se han propuesto diversos tipos de técnicas que recopilan Jia y Harman [16] y por Polo y Reales [17]. Algunas de éstas se basan en la reducción del número de mutantes que se generan, así tenemos: muestreo de mutantes [2, 18], agrupamiento de mutantes [3], mutación selectiva [19], mutación de orden superior [2] y mutación evolutiva [20].

En el grupo de investigación UCASE hemos desarrollado la técnica de prueba Mutación Evolutiva (ME) [20], la cual genera y ejecuta solo un subgrupo de todos los mutantes. Los mutantes son seleccionados mediante algoritmos genéticos, sin pérdida de efectividad en las pruebas. En trabajos previos, se ha presentado GAmera [14, 15], un sistema para la generación automática de mutantes para composiciones WS-BPEL. GAmera emplea la ME y puede generar, ejecutar, evaluar y clasificar los mutantes según la calidad de éstos. GAmera está desarrollada de tal forma que puede convertirse en un sistema para la generación automática de mutantes para cualquier lenguaje de programación. Incorpora un algoritmo genético que selecciona sólo los mutantes de mayor calidad, reduciendo el coste computacional que supondría la ejecución de todos los mutantes. Debido a sus características, se quiere adaptar este sistema de generación automática de mutantes a otros lenguajes de programación, como EPL. Con esto se consigue que la técnica de prueba ME se utilice con este nuevo lenguaje, abriendo un nuevo camino para comprobar la efectividad de la misma.

La motivación de nuestro trabajo viene en gran parte del gran número de trabajos que proponen operadores de mutación para las consultas SQL [1, 7, 10 – 15]. Sin embargo, esos trabajos no consideran las características propias de las consultas EPL, tales como las ventanas de eventos tanto de tiempo como de longitud. Los criterios para matar a los mutantes propuestos en estos trabajos no pueden ser aplicados directamente en las consultas EPL. Además, los casos de prueba que proponen, están descritos basándose en las tablas y filas de las bases de datos. Nuestro trabajo propone nuevos operadores y criterios para matar a los mutantes para las consultas EPL. Del mismo modo, se generan casos de prueba específicos para EPL (flujos de eventos y entradas maliciosas), que no se contemplan en los trabajos anteriores.

En [21] se presenta una primera comparación entre los operadores de mutación de los lenguajes (SQL y EPL) y se describen algunos cambios que tienen que sufrir los operadores de SQL para ser adaptados al nuevo lenguaje. Este artículo describe de forma detallada un conjunto de operadores de mutación para EPL, sus criterios para matar a un mutante, junto con la generación de los casos de prueba. Shahriar et al. [10] proponen operadores de mutación que introducen ataques SQL relacionados con los casos de prueba en aplicaciones web. Sin embargo, el trabajo se enfoca en introducir ataques SQL con consultas SQL comunes y con llamadas a eventos que relacionan la API de la base de datos. Este trabajo introduce ataques SQL para consultas EPL en el contexto de aplicaciones en tiempo real.

6 Conclusiones y trabajo futuro

Probar aplicaciones en tiempo real, las cuales consumen y procesan grandes flujos de eventos es un reto. La prueba de mutaciones puede ser una técnica complementaria para evaluar la calidad de las aplicaciones en tiempo real. Los problemas de calidad pueden resolverse realizando pruebas en las consultas EPL de las aplicaciones. Desafortunadamente las investigaciones existentes no están enfocadas en esta dirección.

Este trabajo desarrolla operadores de mutación nóveles y los criterios para matar a un mutante, para evaluar la calidad de las consultas EPL. Los operadores que proponemos pueden ser aplicados para evaluar y generar casos de prueba efectivos, de forma que en los flujos de eventos de datos, se pueden revelar errores de implementación en específicas expresiones de patrones, en ventanas de tiempo y longitud variable y en los eventos procesados por lotes. Desarrollamos operadores para generar entradas dinámicas maliciosas que resultan tras entradas SQL en consultas EPL. También se habla sobre los operadores que son necesarios para probar la exactitud de las cláusulas, operadores y el uso correcto de la palabra reservada NULL en las consultas EPL. Por último, se presenta una arquitectura para la prueba sistemática de consultas EPL.

Nuestro trabajo futuro incluye abarcar diversas direcciones. Se está implementando una aplicación similar a MuBPEL [8] para EPL, la cual se llama MuEPL. MuEPL, analizará, generará todos los mutantes implicados y los ejecutará contra el programa original. Actualmente MuEPL analiza las consultas EPL determinando los operadores de mutación que se pueden aplicar. Tras analizar las consultas, y según los operadores de mutación que se apliquen, se generan los mutantes. MuEPL también podrá ser usada para hacer la evaluación experimental en las aplicaciones en tiempo real que ejecutan consultas EPL, evaluar la calidad de los flujos de eventos y la efectividad de la técnica de prueba ME. Estudiaremos el desarrollo de operadores basados en la calidad específica de aplicaciones en tiempo real, que pueden relacionarse con el rendimiento y la ejecución de las consultas EPL. Un inconveniente de la prueba de mutaciones es la existencia de mutantes equivalentes; también planeamos identificar los operadores que generan este tipo de mutantes para probar las aplicaciones matando a un menor número de mutantes.

7 Agradecimientos

Este trabajo ha sido financiado por el proyecto MoDSOA (TIN2011-27242) del Programa Nacional de Investigación, Desarrollo e Innovación del Ministerio de Ciencia e Innovación y por el proyecto PR2011-004 del Plan de Promoción de la Investigación de la Universidad de Cádiz.

8 Referencias

1. W. K. Chan, S. C. Cheung, T. H. Tse “Fault-based testing of database application programs with conceptual data model”, Proc. 5th Annual International Conference on Quality Software (QSIC 2005) IEEE Computer Society Press, pp. 187 – 198

2. T.A. Budd, Mutation Analysis of Program Test Data. Ph.D. Thesis, Yale University, 1980.
3. S. Hussain, Mutation Clustering. Master's thesis, King's College London 2008.
4. N. King, J. Offutt "A FORTRAN language system for mutation-based software testing" Software - Practice and Experience 21(7), 1991, pp 685 – 718.
5. Y. Ma, J. Offutt, Y. R. Kwon "MuJava: An automated class mutation system" Software Testing, Verification and Reliability 15 (2) 2005, pp 97 - 133.
6. M.E. Delamaro, J.C. Maldonado, "Proteum - a tool for the assessment of test adequacy for C programs", Proc. Conf. on Performability in Computing Systems (PCS 96), pp 79 - 95.
7. J. Tuya, M. Suárez-Cabal, C. de la Riba, "SQLMutation: a Tool to Generate Mutants of SQL Database Queries", 2nd Workshop on Mutation Analysis, 2006, pp 1.
8. MuBPEL website: https://neptuno.uca.es/redmine/projects/ sources-fm/wiki/MuBPEL.
9. EsperTech: http://esper.codehaus.org/.
10. H. Shahriar, M. Zulkernine, "MUSIC: Mutation-based SQL injection vulnerability checking," Proc. of the 8th IEEE International Conf. on Quality Software, pp. 77-86.
11. A. Derezinska "An Experimental Case Study to Applying Mutation Analysis for SQL Queries" Computer Science and Information Technology, 2009.
12. J. Tuya, M. Suárez-Cabal, C. de la Riva, "Mutating database queries", Information and Software Technology, Vol. 49, Issue 4, 2007, pp 398 - 417.
13. H. Shahriar, Mutation-based testing of buffer overflows, SQL injections, and format string bugs, Thesis, Queen's University, 2008 http://qspace.library.queensu.ca/handle/1974/1359.
14. J.J. Domínguez-Jiménez, A. Estero-Botaro, I. Medina-Bulo,"A framework for mutant genetic generation for WS-BPEL", SOFSEM 2009, Proc. of the 35th Conference on Current Trends in Theory and Practice of Computer Science, (5404), Springer-Verlag, pp 229-240.
15. A. Estero Botaro, J. Domínguez Jiménez, L. Gutiérrez Madroñal, I. Medina Bulo "Evaluación de la calidad de los mutantes en la prueba de mutaciones" Proc. XVI JISBD 2011.
16. Y. Jia, M. Harman, "Higher order mutation testing" Information and Software Technology, Vol. 51, Issue 10, 2009, pp. 1379-1393.
17. M. Polo Usaola, P. Reales Mateo, "Mutation testing cost reduction techniques: A survey" IEEE Software, Vol 27, No.3, 2010, pp. 80 - 86.
18. A.T. Acree, On Mutation, Ph.D. Thesis, Georgia Institute of Technology, 1980.
19. A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, "An experimental determination of sufficient mutant operators" ACM Transactions on Software Engineering. and Methodology. 5, 1996, pp 99-118.
20. J.J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, I. Medina-Bulo, "Evolutionary mutation testing" Information and Software Technology 2011. http://dx.doi.org/10.1016/j.infsof.2011.03.00.
21. L. Gutierrez-Madroñal, J.J. Domínguez-Jiménez, I. Medina-Bulo, "Análisis del lenguage EPL para la prueba de mutaciones", Proc. 3rd JORPRESI, 2011, pp. INF37-INF40.
22. M. Trakhtenbrot, "Implementation-Oriented Mutation Testing of Statechart Models," Proc. 3rd ICSTW, 2010, pp. 120-125.
23. R. Geist, A. Offutt, and F. Harris, "Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis," IEEE Transactions on Computers -Special issue on Fault-tolerant Computing, Vol. 41, Issue 5, 1992, pp. 550-558.
24. R. Nilsson and J. Offutt, "Automated Testing of Timeliness: A Case Study," Proc. of the 2nd International Workshop on Automation of Software Test, 2007, pp. 11-18.
25. A. Sung, J. Jang, and B. Choi, "Fault-Based Interface Testing Between Real-Time Operating System and Application," Proc. 2nd ISSRE Workshop Mutation Analysis, 2006, pp. 8.
26. SQL Injection – OWASP, Accessed: https://www.owasp.org/index.php/SQL_Injection

Testing in Service Oriented Architectures with dynamic binding: A mapping study

Marcos Palacios, José García-Fanjul, Javier Tuya

Department of Computing, University of Oviedo
Campus Universitario de Gijón. 33204, Asturias, Spain

{palaciosmarcos, jgfanjul, tuya}@uniovi.es

1 Summary

This article aims at identifying the state of the art in the research on testing Service Oriented Architectures (SOA) with dynamic binding. Testing SOA presents new challenges to researchers because some traditional testing techniques need to be suitably adapted due to the unique features of this new paradigm, for example, the dynamic behavior that allows discovering, selecting and binding a service at runtime. Testing this dynamic binding is one of the most challenging tasks in SOA because the final bound services cannot be known until the moment of the invocations. Hence, there have been a number of recent studies that aim at improving the quality of the dynamic binding using testing approaches.

The objective of this review is to search, analyze and synthesize the different approaches that have been previously proposed. Thus, following the guidelines proposed by Prof. Barbara Kitchenham we have performed a mapping study, which is a particular form of systematic literature review (SLR) that contributes to identify and categorise the available research on a specific topic.

The mapping study has been carried out following a protocol we have developed to guide the search, selection and synthesis of the studies. This protocol includes a set of research questions and a three-phased strategy that allows searching in a broad number of journals and conferences/workshops proceedings. With the aim of selecting the most relevant studies, we have devised study selection criteria that allow deciding whether a study is finally included or excluded in the set of primary studies. Before applying these criteria, we found 392 papers. Removing the duplicates and excluding such papers that do not pass the different criteria, a set of 33 primary studies were finally selected.

As a result of this review, the objectives of the different approaches are grouped into two categories: studies that aim to detect faults in the service oriented application (19 studies) or studies that make a decision about the service to be invoked based on the test results (14 studies). The proposed testing approaches focus more on non-functional characteristics rather than on functional.

Regarding the applied testing techniques, the results of this review show that, currently, two thirds of the studies apply monitoring approaches to improve the dynamic binding. These approaches check properties of the executing system in order to perform an adaptive action (for example, rebind to another service) when a

A. Ruíz, L. Iribarne (Eds.): JISBD'2012, pp. 383-384, ISBN:978-84-15487-28-9.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.