Revista
Española de

Innovación, Calidad e Ingeniería del Software

REICIS

Volumen 7, No. 1, abril, 2011

Web de la editorial: www.ati.es
Web de la revista: www.ati.es/reicis
E-mail: calidadsoft@ati.es
ISSN: 1885-4486

Publicado por la Asociación de Técnicos de Informática (ATI), Via Laietana, 46, 08003 Barcelona.

Secretaría de dirección: ATI Madrid, C/Padilla 66, 3º dcha., 28006 Madrid

Revista Española de Innovación, Calidad e Ingeniería del Software (REICIS)

Contenidos

REICIS

Equivalencias entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes

Juan Boubeta-Puig, Inmaculada Medina-Bulo y Antonio García-Domínguez
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Cádiz
{juan.boubeta, inmaculada.medina, antonio.garciadominguez}@uca.es

Resumen

Para aplicar prueba de mutaciones a un programa escrito en un lenguaje concreto se requiere un conjunto de operadores de mutación definido para el mismo. Además, es necesario que dichos operadores cubran adecuadamente todas las características del lenguaje para que sean efectivos. En este artículo, se evalúan cualitativamente los operadores de mutación definidos para el lenguaje de ejecución de procesos de negocio WS-BPEL 2.0 y se estudian las analogías y diferencias entre éste y otros lenguajes. Se revisan los operadores existentes para varios lenguajes de propósito general así como para otros específicos del dominio. Los resultados confirman que WS-BPEL es muy diferente a otros lenguajes, ya que aproximadamente sólo la mitad de sus operadores es equivalente a los de otros. Por tanto, es posible mejorar su conjunto de operadores de mutación.

Palabras clave: prueba de software, análisis de mutaciones, servicios web, WS-BPEL.

Equivalences between mutation operators defined for WS-BPEL 2.0 and other languages

Abstract

Applying mutation testing to a program written in a certain language requires that a set of mutation operators is defined for that language. The mutation operators need to adequately cover the features of that language in order to be effective. In this work, we evaluate qualitatively the operators defined for the Web Services Business Process Execution Language 2.0 (WS-BPEL) and study the differences and similarities between WS-BPEL and other languages. We review the existing operators for several structured and object-oriented general-purpose programming languages, and for several domain-specific languages. Results confirm that WS-BPEL is very different from other languages, as near half of the mutation operators for this language are equivalent to those of other languages. Our study concludes that the set of WS-BPEL mutation operators can be improved.

Key words: software testing, mutation analysis, web services, WS-BPEL.

Boubeta-Puig, J., Medina-Bulo, I. y García-Domínguez, A., "Equivalencias entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes", REICIS, vol. 7, no.1, 2011, pp.23-46. Recibido: 15-6-2010; revisado: 12-7-2010; aceptado:30-3-2011

1. Introducción y motivación

La prueba de mutaciones es una técnica de prueba de software basada en fallos, que genera un conjunto de mutantes al aplicar uno o más operadores de mutación. Cada operador realiza un cambio sintáctico en el programa que se desea probar. Los mutantes de primer orden o *First Order Mutants* (FOMs) aplican un único operador por mutante, mientras que los mutantes de orden superior o *Higher Order Mutants* (HOMs) aplican más de uno por mutante [1]. Algunos operadores modelan los errores típicos que suelen cometer los programadores mientras que otros miden ciertos criterios de cobertura.

El lenguaje de ejecución de procesos de negocio o *Web Services Business Process Execution Language* (WS-BPEL) [2] permite a los programadores definir nuevos servicios web o *Web Services* (WS) a partir de otros más simples, esto es, una composición de servicios. Este lenguaje soporta la invocación de WS externos tanto de forma síncrona como asíncrona así como la paralelización de bucles, entre otros. Por ello, es un lenguaje muy potente para ciertas aplicaciones, como la implementación de procesos de negocio basados en flujos de trabajo sobre WS existentes; aunque también presenta nuevos retos para las pruebas.

Estero et al. han definido un conjunto de operadores de mutación para WS-BPEL en [3] y los han evaluado cuantitativamente en [4]. Nuestro trabajo se diferencia de este en que se realiza una comparativa cualitativa entre los operadores definidos para WS-BPEL y los definidos para otros lenguajes. Los objetivos son comprobar si falta por definir alguno para WS-BPEL y establecer las diferencias y analogías entre WS-BPEL y otros lenguajes, ya que en la actualidad existen pocos trabajos sobre estos operadores.

El resto del artículo se estructura de la siguiente forma. En la sección 2 se especifican las características principales de WS-BPEL. En la sección 3 se presentan algunos trabajos que definen operadores de mutación para C, Fortran, Ada, C++, C#, ASP.NET, Java, SQL y XSLT. En la sección 4 se definen los operadores de mutación para WS-BPEL. En las secciones 5, 6, 7 y 8 se describe, respectivamente, qué operadores de mutación para otros lenguajes son comparables a los de WS-BPEL, los operadores de WS-BPEL que no son aplicables a otros lenguajes, los operadores de mutación para otros lenguajes que no son aplicables a WS-BPEL y los que podrían aplicarse a programas WS-BPEL y que, sin embargo, no están definidos para este lenguaje. En la sección 9 se presentan los resultados

obtenidos. Finalmente, en la sección 10 se presentan algunas conclusiones y las líneas de trabajo futuras.

2. El lenguaje WS-BPEL

WS-BPEL es un lenguaje basado en XML que permite implementar procesos de negocio utilizando WS externos. El proceso de negocio resultante es un WS más complejo, conocido como una composición de servicios. Cada proceso WS-BPEL se divide en cuatro secciones:

1. Definición de relaciones con los socios externos, que son el cliente que utiliza el proceso de negocio y los WS a los que llama el proceso.
2. Definición de las variables que emplea el proceso, basada en *Web Services Description Language* (WSDL) y *XML Schema*.
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso: manejadores de fallos (indican las acciones a realizar en caso de producirse un fallo interno o en un WS al que se llama), manejadores de eventos (especifican las acciones a realizar en caso de que el proceso reciba una petición durante su flujo normal de ejecución), manejadores de terminación (gestionan la terminación del proceso) y manejadores de compensación (permiten deshacer una invocación previa de WS).
4. Descripción del comportamiento del proceso de negocio; esto se logra a través de las actividades que proporciona el lenguaje.

Todos los elementos comentados anteriormente son globales por defecto. Sin embargo, también existe la posibilidad de declararlos de forma local mediante el elemento XML *scope*, que permite dividir el proceso de negocio en diferentes ámbitos.

Los principales elementos constructivos de un proceso WS-BPEL son las actividades, que pueden ser de dos tipos: básicas y estructuradas. Las actividades básicas son las que realizan una determinada labor (recepción de un mensaje de un socio externo, envío de un mensaje a un socio externo, etc.). Las actividades estructuradas pueden contener otras actividades y definen la lógica de negocio.

Cada actividad se representa mediante un elemento XML. A las actividades pueden asociarse un conjunto de atributos y un conjunto de contenedores. Estos últimos pueden incluir diferentes elementos, que a su vez pueden tener atributos asociados.

WS-BPEL permite realizar acciones en paralelo y de forma sincronizada. Por ejemplo, la actividad *flow* permite ejecutar un conjunto de actividades concurrentemente especificando las condiciones de sincronización entre ellas. En la figura 1 se muestra un ejemplo en el que *flow* invoca a tres WS en paralelo, comprobarVuelo, comprobarHotel y comprobarAlquilerCoche. Además, existe otro WS, reservarVuelo, que solo se invocará si se completa comprobarVuelo. Esta sincronización entre actividades se consigue estableciendo un enlace o *link*; por lo que la actividad objetivo del enlace se ejecutará solo si se ha completado la actividad fuente de ese enlace.

```
<flow> ← Actividad estructurada
  <links> ← Contenedor
    <link name="comprobarVuelo-reservarVuelo" ← Atributo /> ← Elemento
  </links>
  <invoke name="comprobarVuelo" ... > ← Actividad básica
    <sources> ← Contenedor
      <source linkName="comprobarVuelo-reservarVuelo" />
    </sources>
  </invoke>
  <invoke name="comprobarHotel" ... />
  <invoke name="comprobarAlquilerCoche" ... />
  <invoke name="reservarVuelo" ... >
    <targets> ← Contenedor
      <target linkName="comprobarVuelo-reservarVuelo" /> ← Elemento
    </targets>
  </invoke>
</flow>
```

Figura 1. Ejemplo de una actividad *flow* para WS-BPEL 2.0

En WS-BPEL pueden utilizarse distintos lenguajes de expresiones. Todos los motores WS-BPEL estándar soportan *XML Path Language* 1.1 (XPath). XPath es un lenguaje declarativo que permite realizar consultas sobre documentos XML y que dispone de los operadores aritméticos, relacionales y lógicos, con una sintaxis similar a la de los lenguajes tradicionales.

3. Trabajos relacionados

Se han publicado muchos trabajos sobre la aplicación de la prueba de mutaciones [5]. En este artículo se seleccionan los trabajos que definen los operadores de mutación para los lenguajes de propósito general más conocidos de programación estructurada y orientada a objetos, y para algunos de los más populares específicos del dominio. Todos los operadores comentados en este trabajo se recogen en la tabla 1.

En cuanto a los operadores de mutación definidos para los lenguajes de programación estructurada, Agrawal et al. definen un conjunto de 77 operadores para C [6], que son implementados más tarde en la herramienta Proteum [7], y los comparan con los operadores de mutación definidos para Fortran.

En un trabajo posterior, Barbosa et al. [8] proponen un procedimiento que permite seleccionar sistemáticamente un conjunto de operadores suficientes para C. La técnica de mutación selectiva trata de encontrar un conjunto de operadores de mutación que genere un subconjunto de todos los posibles mutantes sin pérdida significativa de la efectividad de los casos de prueba [5]. Algunos autores han propuesto los operadores de mutación suficientes para un lenguaje específico. Sin embargo, la mayoría de los estudios se han realizado sobre los operadores de mutación tradicionales.

Los 10 operadores de mutación propuestos como suficientes a partir de los 77 operadores para C son: SWDD (sustituye *while* por *do-while*), SMTC (introduce una guarda antes del cuerpo de un bucle), SSDL (elimina una sentencia), OLBN (sustituye un operador lógico por un operador de bit), OASN (sustituye un operador aritmético por uno de desplazamiento), ORRN (sustituye un operador relacional por otro), VTWD (incrementa o decrementa en 1 el valor de una variable escalar), VDTR (proporciona cobertura de dominio para variables escalares), Cccr (sustituye una constante por otra) y Ccsr (sustituye una constante por un escalar).

King y Offutt definen un conjunto de 22 operadores de mutación para Fortran y los implementan en la herramienta Mothra [9]. Más tarde, Offutt et al. [10] enumeran los 5 operadores que son suficientes: ABS (antes de una expresión o subexpresión, inserta el operador unario de valor absoluto, valor absoluto negativo o el que comprueba si la expresión o subexpresión es cero), UOI (inserta un operador unario), LCR (sustituye un

operador lógico por otro), AOR (sustituye un operador aritmético por otro) y ROR (sustituye un operador relacional por otro).

Nombre	*Lenguaje(s) y referencia(s)*	*Sección*	*Nombre*	*Lenguaje(s) y referencia(s)*	*Sección*
ABS	Fortran [9]; Java [24]; SQL [25]	3	OLBN	C [6]	3
AOR	Fortran [9]; C++ [12]; Java [19], [24]; SQL [25]; XSLT [27]	3	OLLN	C [6]	5.2
CBD	Java [23]	5.4	OIPM	C [6]	7
Cccr	C [6]	3	ORO	ASP.NET [16]	5.2
Ccsr	C [6]	3	ORRN	C [6]	3
CMC	Ada [11]	3	OVV	Ada [11]; C++ [12]	5.1
CRCR	C [6]	5.2	PPD	Java [18]	3
CRP	Fortran [9]	5.2	ROR	Fortran [9]; Java [19]; SQL [25]; XSLT [27]	3
DNR	XSLT [27]	8	SAN	Fortran [9]	5.3
DSI	XSLT [27]	8	SDL	Fortran [9]	5.3
EAI	Ada [11]	3	SDWD	C [6]	5.3
EDT	Ada [11]	5.2	SEE	Ada [11]	5.3
EEO	Ada [11]	3	SER	Ada [11]	5.4
EEU	Ada [11]	3	SMO	ASP.NET [16]	5.3
EEZ	Ada [11]	3	SMTC	C [6]	3
EHR	C# [14]; Java [28]	5.4	SMVB	C [6]	7
ELR	Ada [11]	3	SRN	Ada [11]	5.3
EMO	ASP.NET [16]	5.2	SRSR	C [6]	7
EMR	Ada [11]	3	SRW	Ada [11]	5.3
ENI	Ada [11]	3	SSDL	C [6]	3
EOA	Java [18]	3	SSWM	C [6]	7
EOR	Ada [11]	3	STRP	C [6]	5.3
ERR	Ada [11]	3	SVR	Fortran [9]	5.1
ESR	Ada [11]	3	SWDD	C [6]	3
EUI	Ada [11]	3	SWR	Ada [11]	5.3
EUR	Ada [11]	3	UOD	Java [24]	5.2
IBO1	C++ [12]	5.2	UOI	Fortran [9]; C++ [12]; Java [24]; SQL [25]	3
IBO2	C++ [12]	5.2	Uuor	C [6]	8
IBO3	C++ [12]	5.2	VDTR	C [6]	3
IHD	Java [18]	3	VGAR	C [6]	7
IRP	SQL [25]	5.1	VGSR	C [6]	5.1
ISD	Java [18]	3	VGTR	C [6]	7
LCR	Fortran [9]; SQL [25]; Java [20] (como COR), [24]; XSLT [27]	3	VLSR	C [6]	5.1
MXT	Java [22]	5.2	VR	XSLT [27]	5.1
OAAN	C [6]	5.2	VTWD	C [6]	3
OASN	C [6]	3	XER	XSLT [27]	5.2

Tabla 1. Índice de los operadores de mutación para otros lenguajes comentados en este estudio.

Por otro lado, Offutt et al. definen 65 operadores de mutación para Ada [11] y realizan una comparativa entre los operadores definidos para Ada, C y Fortran. Además,

proponen CMC (cobertura de condición múltiple) como el único operador suficiente de mutación de cobertura y 12 operadores suficientes de expresiones: EAI (inserta el valor absoluto), ENI (inserta el valor absoluto negativo), EEZ (inserta el subprograma *Except_on_Zero* antes de cada expresión y subexpresión aritmética), EOR (sustituye un operador aritmético por otro), ERR (sustituye un operador relacional por otro), EMR (sustituye cada IN por NOT IN y viceversa), ELR (sustituye un operador lógico por otro), EUI (inserta el operador menos unario), EUR (sustituye el operador unario por otro), ESR (sustituye cada nombre de función y subrutina por otro que tenga la misma signatura y pertenezca al mismo paquete), EEO (inserta el subprograma *Except_on_OverFlow* antes de cada expresión aritmética) y EEU (inserta el subprograma *Except_on_UnderFlow* antes de cada expresión aritmética).

Otros autores definen operadores de mutación para lenguajes de programación orientados a objetos como C++, C#, ASP.NET o Java.

Zhang [12] define 24 operadores para C++. Dereziňska define 40 operadores de mutación para C# en [13] y [14], que son implementados en la herramienta CREAM [15]. Mansour y Houri [16] han propuesto un conjunto de 15 operadores de mutación para evaluar la calidad de las pruebas realizadas a aplicaciones web escritas en ASP.NET.

Existen muchos trabajos sobre operadores de mutación para Java. Ma et al. definen 26 operadores de mutación de clases [17] (redefinidos en trabajos posteriores como en [18]) y de métodos [19]; estos operadores han sido integrados en la herramienta MuJava [20]. Más tarde, estudian la efectividad de los operadores de mutación de clases que producen pocos mutantes e identifican los siguientes [21]: IHD (elimina una variable oculta), ISD (elimina la palabra reservada *super*), PPD (declara un parámetro variable con un tipo de clase hija) y EOA (sustituye una asignación de referencia y una asignación de contenido). Estos autores planean ampliar su estudio para determinar un conjunto de operadores de mutación suficientes.

Bradbury et al. [22] definen 24 operadores de mutación específicos para el comportamiento concurrente de Java. Ji et al. [23] han definido 5 operadores para la captura de excepciones. Madeyski y Radyk [24] han seleccionado 16 operadores de mutación de entre todos los definidos por Ma et al. [17] y Offutt et al. [18], y los han integrado en la herramienta Judy [24]. Otros trabajos definen operadores de mutación para SQL, un

lenguaje específico de dominio para operar sobre bases de datos relacionales. Tuya et al. [25] definen 22 operadores suficientes para consultas SELECT, integrados en la herramienta SQLMutation [26], que son evaluados posteriormente por Dereziňska [1].

Lonetti y Marchetti [27] proponen 21 operadores de mutación para *Extensible Stylesheet Language Transformations* (XSLT), un lenguaje que permite transformar un documento XML en otros formatos. Los autores han adaptado al lenguaje XSLT algunos de los operadores de variables y expresiones implementados en las herramientas Mothra [9] y MuJava [20]. Además, definen operadores específicos para la manipulación de elementos XSLT, como expresiones XPath y reglas de plantillas. Una colección de reglas es un programa XSLT básico.

4. Operadores de mutación para WS-BPEL

Estero et al. [3] han definido 26 operadores de mutación para WS-BPEL (ver tabla 2). Los nombres de estos operadores están compuestos por tres letras en mayúsculas. La primera letra identifica su categoría. Existen cuatro categorías, de acuerdo con el tipo de elemento sintáctico de WS-BPEL afectado por el operador: mutación de Identificadores (I), de Expresiones (E), de Actividades (A), y de condiciones eXcepcionales y eventos (X). Las últimas dos letras representan la acción realizada sobre el elemento sintáctico.

Es importante tener en cuenta que estos operadores solo reemplazan o eliminan código: ninguno añade código. Los autores consideran que es complicado añadir código por error, puesto que los procesos WS-BPEL son codificados normalmente mediante herramientas gráficas (excepto las expresiones XPath). Cabe destacar que, aunque se usen herramientas gráficas, los programadores podrían escoger el elemento equivocado seleccionando, por ejemplo, una actividad *while* en lugar de una actividad *repeatUntil*, o estableciendo el valor de un atributo a *yes* en vez de *no*.

5. Operadores equivalentes a los de WS-BPEL

En esta sección se estudian las similitudes existentes entre los operadores definidos para WS-BPEL (ver sección 4) y los definidos para otros lenguajes (ver sección 3).

La tabla 3 resume los resultados presentados en esta sección. Los operadores de mutación definidos para otros lenguajes que son equivalentes a los de WS-BPEL se han

clasificado en las cuatro categorías mencionadas en la sección 4 y se han distinguido dos tipos de operadores: similares y completamente equivalentes.

Operador	*Descripción*
Mutación de Identificadores	
ISV	Sustituye el identificador de una variable por el de otra del mismo tipo
Mutación de Expresiones	
EAA	Sustituye un operador aritmético (+, -, *, *div*, *mod*) por otro del mismo tipo
EEU	Elimina el operador menos unario de cualquier expresión
ERR	Sustituye un operador relacional (<, >, >=, <=, =, !=) por otro del mismo tipo
ELL	Sustituye un operador lógico (*and*, *or*) por otro del mismo tipo
ECC	Sustituye un operador de camino (/, //) por otro del mismo tipo
ECN	Modifica una constante numérica incrementando o decrementando su valor en una unidad, añadiendo o eliminando un dígito
EMD	Modifica una expresión de duración cambiando por 0 o por la mitad el valor inicial
EMF	Modifica una expresión de fecha límite cambiando por 0 o por la mitad el valor inicial
Mutación de Actividades	
Relacionados con la concurrencia	
ACI	Cambia el atributo *createInstance* de las actividades de recepción de mensajes a no
AFP	Cambia una actividad *forEach* secuencial a paralela
ASF	Cambia una actividad *sequence* por una actividad *flow*
AIS	Cambia el atributo *isolated* de un *scope* a no
No concurrentes	
AEL	Elimina una actividad
AIE	Elimina un elemento *elseif* o el elemento *else* de una actividad *if*
AWR	Cambia una actividad *while* por una actividad *repeatUntil* y viceversa
AJC	Elimina el atributo *joinCondition* de cualquier actividad en la que aparezca
ASI	Intercambia el orden de dos actividades hijas de una actividad *sequence*
APM	Elimina un elemento *onMessage* de una actividad *pick*
APA	Elimina el elemento *onAlarm* de una actividad *pick* o de un manejador de eventos
Mutación de Condiciones Excepcionales y Eventos	
XMF	Elimina un elemento *catch* o el elemento *catchAll* de un manejador de fallos
XMC	Elimina la definición de un manejador de compensación
XMT	Elimina la definición de un manejador de terminación
XTF	Cambia el fallo lanzado por una actividad *throw*
XER	Elimina una actividad *rethrow*
XEE	Elimina un elemento *onEvent* de un manejador de eventos

Tabla 2. Operadores de mutación para WS-BPEL 2.0

Los operadores similares son aquellos operadores definidos para otros lenguajes que deben redefinirse con pequeños cambios para que sean equivalentes a otro operador de mutación para WS-BPEL (aparecen marcados con ⋆ en la tabla 3), mientras que los operadores completamente equivalentes, como su nombre indica, son aquellos que no necesitan ser redefinidos para ser comparables con los de WS-BPEL.

		Lenguajes de programación estructurada			*Lenguajes de programación orientada a objetos*				*Lenguajes específicos del dominio*	
Categoría	*WS-BPEL*	*C*	*Fortran*	*Ada*	*C++*	*C#*	*ASP .NET*	*Java*	*SQL*	*XSLT*
Mutación de identificador	ISV	⋆VGSR, ⋆VLSR	SVR	OVV	OVV				⋆IRP	⋆VR
Mutación de expresión	EAA	OAAN	AOR	EOR	AOR, IBO1		⋆ORO	AOR	AOR	AOR
	EEU							UOD		
	ERR	ORRN	ROR	ERR	IBO2		⋆ORO	ROR	ROR	ROR
	ELL	OLLN	LCR	ELR	IBO3		⋆ORO	LCR	LCR	LCR
	ECC									⋆XER
	ECN	⋆CRCR	⋆CRP	⋆EDT			⋆EMO			
	EMD							⋆MXT		
Mutación de actividad	AEL	⋆SSDL	⋆SDL	⋆SRN			⋆SMO			
	AIE	⋆STRP	⋆SAN	⋆SEE			⋆SMO			
	AWR	⋆SWDD, ⋆SDWD		⋆SWR, ⋆SRW						
Mutación de condición excepcional y evento	XMF					⋆EHR		⋆EHR, ⋆CBD		
	XTF			⋆SER						

Tabla 3. Equivalencias entre los operadores de mutación para WS-BPEL y otros lenguajes.

5.1. Mutación de identificadores

Existen operadores de sustitución de identificadores para la mayoría de los lenguajes de programación estudiados, excepto para C#, ASP.NET y Java.

Para el lenguaje C existen dos operadores similares a ISV: VGSR y VLSR. VGSR cambia los identificadores de las variables escalares que son globales a una función, mientras que VLSR cambia variables escalares que son locales a una función. Las funciones no existen en WS-BPEL, pero el ámbito de una declaración de variable puede limitarse introduciendo su declaración dentro de un elemento *scope*. VGSR podría ser equivalente si se limita ISV a que sustituya un identificador por otro que se encuentre en los ámbitos padres, excepto en el más próximo. VLSR podría ser equivalente si ISV únicamente sustituye un identificador por otro del ámbito padre más próximo.

IRP, para consultas SELECT de SQL, tiene cierto parecido con ISV. Este operador sustituye cada parámetro de consulta por otro parámetro, columna o constante de tipo compatible. Es más general que ISV, puesto que sustituye identificadores por constantes y

no solo por otros identificadores. Al igual que hace ISV, comprueba el tipo de la sustitución para evitar la generación de mutantes inválidos.

El operador VR para XSLT es similar a ISV. VR sustituye el valor del atributo *select* del elemento *variable* de XSLT por otro valor de los atributos *select* contenidos en otros elementos *variable*, mientras que ISV sustituye el valor del atributo *variable* de las actividades WS-BPEL.

Los operadores SVR para Fortran, OVV para Ada y OVV para C++ son equivalentes a ISV sin realizar ninguna modificación sobre sus definiciones.

5.2. Mutación de expresiones

A continuación, se tratarán las mutaciones de operadores aritméticos, relacionales y lógicos, así como las mutaciones de constantes y de duración.

OAAN para C es equivalente a EAA. Para Ada, EOR muta, además de los operadores aritméticos básicos, el operador de exponenciación ** y el de resto *rem*.

El operador AOR, originalmente para Fortran y más tarde adaptado para SQL, Java, C++ y XSLT, sustituye cada uno de los operadores (+, –, *, /) por otro operador del mismo conjunto. Este operador es equivalente a EAA. Las versiones para SQL y Fortran también pueden sustituir la expresión por *leftop* (devuelve el operando izquierdo, ignorando el derecho) y *rightop* (devuelve el operando derecho, ignorando el izquierdo). Además, la versión para Fortran puede reemplazar el operador por la función módulo MOD y el operador de exponenciación **.

Hay otro operador para C++ que es equivalente a EAA: IBO1. Este sustituye cada operador multiplicativo (*, /, %) o aditivo (+, –) por otro de los operadores multiplicativos o aditivos, respectivamente.

ORO es el operador de mutación de expresiones para ASP.NET. Este sustituye un operador por otro; puesto que no define cuál debe ser el tipo de operador, se considera similar a EAA, ERR y ELL.

Tan solo se ha encontrado un operador equivalente a EEU: UOD para Java. La mayoría de los autores prefieren insertar el operador de negación unario o reemplazarlo.

Los operadores equivalentes a ERR son: ORRN para C, ROR para Fortran, SQL, Java y XSLT, ERR para Ada y IBO2 para C++. En el caso de SQL y Fortran, la expresión

relacional también se sustituye por *trueop* (siempre devuelve un valor booleano verdadero) y *falseop* (siempre devuelve un valor booleano falso).

ELL tiene cuatro operadores de mutación equivalentes: OLLN para C, LCR para Fortran, SQL, Java y XSLT, ELR para Ada y IBO3 para C++. En el caso de SQL y Fortran, los operadores lógicos también pueden ser sustituidos por *falseop*, *trueop*, *leftop* y *rightop*. Además de los operadores *and* y *or*, ELR también cambia el operador lógico por *xor* y las formas en corto circuito *or else* and *and then*.

ECC sustituye un operador de camino (/, //) por otro del mismo tipo. De entre los lenguajes estudiados, los operadores de camino que permiten recorrer árboles aparecen únicamente en XPath. XER, para XSLT, sustituye una expresión XPath del atributo *select* del elemento *value-of* por una expresión XPath contenida en otro atributo *select* del elemento *value-of*. Este es similar a ECC si solo reemplaza los de camino.

El operador de mutación para C equivalente a ECN es CRCR. Este asegura que se manejan apropiadamente las referencias escalares de enteros y flotantes, como no hace ECN, que considera todos los números iguales, conforme al sistema de tipos XPath. CRCR puede incrementar, decrementar y sustituir una referencia de puntero a *null*. Sin embargo, CRCR no añade o elimina dígitos a una constante como sí hace ECN.

Otros operadores comparables con ECN son: EDT para Ada (si se considera únicamente mutaciones de expresiones constantes), EMO para ASP.NET y CRP para Fortran. CRP también permite incrementar o decrementar constantes de precisión doble en un 10%, y sustituir 0 por 0,01 y -0,01.

MXT para Java es el único operador que se ha encontrado para modificar expresiones de duración, al igual que hace EMD. Este operador para Java modifica el argumento opcional de las llamadas a los métodos estándar de Java *wait()*, *sleep()*, *join()* y *await()*, que representa el intervalo en milisegundos que el hilo actual debería esperar. Mientras que MXT cambia el tiempo especificado por el doble o la mitad, EMD lo cambia por 0 o la mitad. La duración es un valor entero integrado en Java, mientras que en WS-BPEL se utiliza una constante de duración de XML Schema.

5.3. Mutación de actividades

Algunos de los operadores de mutación de actividades para WS-BPEL modelan el fallo que puede cometerse al elegir una actividad que no es la más adecuada para las acciones que se

deben realizar, sustituyendo una actividad por otra. Los demás modelan la elección de un valor incorrecto para los atributos de las actividades, sustituyendo para ello su valor actual por otro valor válido. Estos operadores se clasifican en dos tipos, los relacionados con la concurrencia y los no concurrentes.

SSDL para C, SDL para Fortran, SRN para Ada y SMO para ASP.NET son similares a AEL. Mientras AEL elimina una actividad, los otros operadores eliminan una sentencia. No obstante, aunque SRN sustituye una sentencia por NULL, puede considerarse similar a AEL. STRP para C, SAN para Fortran, SEE para Ada y SMO para ASP.NET son similares a AIE, pero SMO difiere en que solo elimina una única sentencia de la rama *else*, en lugar de la rama entera.

Existen dos operadores para C que son similares a AWR: SWDD y SDWD. Estos operadores cambian un bucle *while* por un bucle *do-while* y viceversa, respectivamente. Asimismo, AWR intercambia las actividades *while* y *repeatUntil*, que son semánticamente equivalentes a las sentencias anteriores.

SWR y AWR, para Ada, también intercambian dos sentencias pero, en este caso, cambia el bucle *while* condicional por el bucle *loop* incondicional y viceversa.

5.4. Mutación de excepciones y eventos

Los operadores de mutación relacionados con las condiciones excepcionales y eventos para WS-BPEL están relacionados con los distintos tipos de manejadores que proporciona WS-BPEL: manejadores de fallos, eventos, compensación y terminación.

Los manejadores de fallos de WS-BPEL utilizan elementos *catch* para manejar fallos específicos y el elemento *catchAll* para manejar el resto. Estos son bastante similares a las excepciones, que son frecuentemente utilizadas para manejar los errores en los lenguajes modernos de programación orientada a objetos. Solo se han encontrado dos operadores similares a XMF: EHR para Java y C#, y CBD para Java. EHR elimina un único bloque *catch* junto con el bloque *finally* si no queda ningún bloque *catch*. CBD simplemente elimina un bloque *catch*.

XTF cambia el nombre del fallo lanzado por una actividad *throw* por otro del mismo ámbito. Esta actividad permite lanzar un fallo determinado, cuyo nombre se especifica mediante el atributo *faultName*. En el caso de los lenguajes tradicionales, la sentencia *throw*

tiene un comportamiento similar. SER, para Ada, cambia el nombre de la excepción utilizada en una sentencia *raise*.

6. Operadores no aplicables a WS-BPEL

En esta sección, se enumeran los operadores definidos para WS-BPEL que no son aplicables a ninguno de los descritos en la sección 3 y se clasifican en dos grupos, atendiendo a las características que mutan: características específicas del lenguaje WS-BPEL y características que comparten WS-BPEL y otros lenguajes.

6.1. Características específicas del lenguaje WS-BPEL

WS-BPEL tiene algunas características específicas que no presentan los otros lenguajes. Por tanto, se necesitan operadores específicos para WS-BPEL que muten dichas características.

ACI cambia el atributo *createInstance* de una actividad *receive* o *pick* de *yes* a *no* solo si el proceso tiene más de una actividad con el atributo a *yes*. Este operador no puede ser aplicado a otros lenguajes porque ninguno de los lenguajes estudiados tiene una sentencia especializada que reciba mensajes provenientes de una red.

AFP sustituye una actividad secuencial *forEach* por una paralela, cambiando el atributo *parallel* de *no* a *yes*. No hay ningún operador para otros lenguajes equivalente a AFP, debido a que no incluyen soporte para ejecutar bucles en paralelo. Para implementar los bucles en paralelo es necesario utilizar bibliotecas de terceras partes o extensiones de lenguajes específicos.

AIS cambia el atributo *isolated* de un elemento *scope* de *yes* a *no*. Si *isolated* tiene el valor *yes*, las lecturas y escrituras de variables concurrentes dentro del *scope* son serializadas para preservar la consistencia de los datos. Si se establece el valor *no* a *isolated* podría obtenerse un comportamiento inesperado que debería ser detectado con los casos de prueba existentes. La mayoría de los lenguajes estudiados permiten limitar el ámbito de una declaración introduciendo un constructor específico (como un *block* en C y sus descendientes). Sin embargo, no disponen de constructores específicos para serializar el acceso concurrente de un conjunto de variables.

AJC elimina el atributo *joinCondition* de una actividad dentro de un *flow*. Si no se especifica lo contrario, todas las actividades en un *flow* se ejecutan concurrentemente y

comienzan a la vez. Si una actividad *A* necesita comenzar antes que otra *B*, el programador debe crear un enlace o *link* de *A* a *B*. Todas las actividades esperan hasta que sus enlaces de entrada sean actualizados con la información del éxito o fracaso de sus actividades fuentes. Una vez que esta información esté disponible, cada actividad evaluará su condición de unión para decidir si ejecutar o lanzar un fallo. Por defecto, la ejecución continuará si al menos uno de los enlaces de entrada informa de un estado exitoso. El usuario puede personalizar este comportamiento proporcionando la expresión booleana XPath apropiada en el atributo *joinCondition* de una actividad. La sincronización basada en enlaces y el manejo de fallos no están en ninguno de los otros lenguajes estudiados.

APM elimina un elemento *onMessage* de una actividad *pick* si hay más de uno. *pick* se utiliza cuando se espera uno de los mensajes de las actividades previas. En cierta manera, es similar a una sentencia *switch* de C, pero los casos a probar son más complejos. Cada elemento hijo *onMessage* maneja un tipo de mensaje de entrada específico, y cada *onAlarm* gestiona la situación en la que ningún mensaje ha llegado durante el tiempo especificado. Puesto que no existe ninguna estructura de control en los otros lenguajes, se concluye que no hay ningún operador equivalente a APM.

APA elimina el elemento *onAlarm* de una actividad *pick* o de un manejador de eventos. Por la misma razón que se ha comentado anteriormente, tampoco existen operadores equivalentes a APA.

XMC elimina la definición de un manejador de compensación. WS-BPEL lo sustituirá por el manejador de compensación por defecto. Estos son específicos de WS-BPEL: deshacen las invocaciones previas de servicios cuando se lanza un fallo.

XMT elimina la definición de manejador de terminación. WS-BEL lo sustituirá por el manejador de terminación por defecto. Estos manejadores también son específicos de WS-BPEL: contienen las actividades que decidirán si es necesario terminar la ejecución de un *scope*.

XEE elimina un elemento *onEvent* de un manejador de eventos, modelando la situación en la que el programador olvida tratar un evento en particular. Estos manejadores de eventos se implementan manualmente mediante código en los otros lenguajes, ya que no están integrados.

6.2. Características comunes en WS-BPEL y otros lenguajes

A continuación, se enumeran los operadores de mutación definidos para WS-BPEL que mutan características que comparten tanto el lenguaje WS-BPEL como los demás.

EMF modifica una expresión de fecha. Puede aplicarse a la actividad *wait* y al elemento *onAlarm* de la actividad *pick*. Los cambios introducidos son equivalentes a EMD. Sin embargo, no se ha encontrado ningún operador para los otros lenguajes equivalente a EMF.

ASF sustituye una actividad *sequence* por una actividad *flow*. La actividad *sequence* ejecuta secuencialmente las actividades que contiene, mientras la actividad *flow* las ejecuta en paralelo. No existe ningún operador para los otros lenguajes estudiados que cambie la ejecución secuencial de sentencias a paralela.

ASI intercambia el orden de dos actividades hijas de una actividad *sequence*. Tampoco existe ningún operador para los otros lenguajes estudiados que intercambie el orden de dos sentencias que se ejecuten secuencialmente.

XER elimina una actividad *rethrow*. Esta actividad lanza un fallo capturado previamente, por lo que se activarán los manejadores de fallo padres. Ninguno de los operadores para los otros lenguajes elimina el relanzamiento de una excepción.

7. Operadores no aplicables a WS-BPEL

En esta sección se comentan algunos operadores de mutación definidos para los otros lenguajes que no son aplicables a WS-BPEL, con algunos ejemplos.

Algunos operadores para C y los conceptos con los que se relacionan no están disponibles en WS-BPEL 2.0 como, por ejemplo: los operadores de desplazamiento (OASN), los operadores de bits (OLBN), los vectores y registros (VGAR, muta referencias de vectores utilizando vectores globales, y VGTR, muta referencias de estructuras utilizando referencias de estructuras globales), los punteros (OIPM que muta el operador de indirección) y las funciones (SRSR, sustituye una sentencia *return*).

Por otro lado, las sentencias de control de selección múltiple en WS-BPEL 2.0 se encuentran limitadas a la actividad basada en alarmas y mensajes, y a los manejadores de eventos: la construcción *switch* que comprueba valores enteros no está disponible. Por ello, SSWM (muta una sentencia *switch*) tampoco podrá aplicarse a WS-BPEL.

Puesto que los procesos de negocio WS-BPEL se escriben en XML, en lugar de utilizar las llaves ({, }) para estructurar el código se emplean elementos XML. Los operadores de mutación como SMVB para C, que desplaza una llave arriba o abajo por una sentencia, no tiene sentido en WS-BPEL.

Otra diferencia importante es que mientras la mayoría de los operadores mutan sentencias e instrucciones, los de WS-BPEL mutan actividades, elementos y atributos.

Ninguno de los operadores de mutación de clase para Java definidos en [21] puede aplicarse a código WS-BPEL porque mutan características específicas a la orientación a objetos como la herencia y el polimorfismo, así como características específicas de Java no presentes en el lenguaje WS-BPEL.

Tanto WS-BPEL 2.0 como Java permiten manejar la concurrencia, por lo que se podría pensar que los operadores de mutación para la concurrencia de *Java 2 Standard Edition* (J2SE) 5.0 definidos en [22] podrían ser aplicables a WS-BPEL 2.0. Sin embargo, esto no es posible, excepto para el operador XMT, debido a que sus enfoques para soportar concurrencia son bien distintos.

La mayoría de los operadores de mutación definidos para SQL tampoco son aplicables a código WS-BPEL, excepto los de expresiones aritméticas, lógicas, relacionales y unarias. La razón fundamental es que el propósito de estos dos lenguajes es distinto: SQL es un lenguaje de consulta y WS-BPEL es un lenguaje de ejecución de procesos de negocio.

Los operadores definidos para XSLT que mutan plantillas no son equivalentes a ninguno de los operadores para WS-BPEL porque este no dispone de plantillas. Algunos han integrado XSLT como un lenguaje de expresión adicional para WS-BPEL, pero no es una extensión estándar.

8. Operadores para otros lenguajes que podrían aplicarse a WS-BPEL

En esta sección se pretende encontrar otros operadores de mutación definidos para los otros lenguajes estudiados que podrían aplicarse a WS-BPEL y que, sin embargo, Estero et al. no hayan definido. Para comprobar si el conjunto de dichos operadores puede mejorarse, se ha prestado especial atención a los operadores identificados por la técnica de mutación selectiva que modelan los fallos cometidos por los programadores (consúltese dichos operadores en la sección 3).

Los 5 operadores suficientes para Fortran [10] podrían aplicarse a programas WS-BPEL. Sin embargo, no existe ningún operador definido para WS-BPEL que sea equivalente a ABS o UOI. Por tanto, se proponen dos nuevos operadores de mutación de expresiones para WS-BPEL: EAI (inserta el operador de valor absoluto) y EUI (inserta el operador menos unario). Como se muestra en la tabla 4, estos operadores son completamente equivalentes a los de Ada, mientras que los definidos para el resto son similares, ya que realizan otras mutaciones además de las descritas.

Puesto que no se tiene conocimiento de que exista algún trabajo que defina operadores suficientes para XSLT, se han escogido los operadores de mutación para XSLT propuestos en [27]. Como se comentó en la sección 4, los procesos WS-BPEL son normalmente codificados con herramientas gráficas. Sin embargo, las expresiones XPath sí se escriben a mano, por lo que el programador podría incluir código extra por error. Dos de estos operadores que mutan expresiones XPath podrían aplicarse a programas WS-BPEL: DSI, que inserta una doble barra (//) al principio de una expresión XPath, y DNR, que sustituye . (indica el nodo actual) por .. (indica el nodo padre) y viceversa. Por ello, se proponen otros dos operadores para WS-BPEL completamente equivalentes a los dos anteriores: ECI y ENC, respectivamente.

		Lenguajes de programación estructurada			*Lenguajes de programación orientada a objetos*				*Lenguajes específicos del dominio*	
Categoría	*WS-BPEL*	*C*	*Fortran*	*Ada*	*C++*	*C#*	*ASP .NET*	*Java*	*SQL*	*XSLT*
Mutación de expresión	EAI	⋆VDTR	⋆ABS	EAI			⋆EMO	⋆ABS	⋆ABS	
	EUI	⋆Uuor	⋆UOI	EUI	⋆UOI		⋆EMO	⋆UOI	⋆UOI	
	ENC									DNR
	ECI									DSI

Tabla 4. Nuevos operadores propuestos para WS-BPEL junto con sus equivalentes para otros lenguajes.

9. Resultados

En este estudio se ha realizado una comparativa cualitativa entre los operadores de mutación definidos para WS-BPEL y los de otros lenguajes. Mientras que Estero et al. [3] estudiaron las características específicas de este lenguaje para definir los operadores que las mutaran, en este trabajo se ha llevado a cabo el proceso contrario: se han revisado todos los operadores de mutación definidos para los lenguajes de programación estructurada y

orientada a objetos así como específicos del dominio, para comprobar si estos autores han olvidado definir alguno de los operadores para WS-BPEL.

Como puede observarse en la tabla 3, se ha determinado que el operador de mutación XER para XSLT es similar a ECC, el operador XMT para Java es similar a EMD y el operador SER para Ada es similar a XTF; en cambio, Estero et al. los han definido como operadores específicos para WS-BPEL.

En la sección 8 se ha determinado que el conjunto de operadores de mutación para WS-BPEL no es completo y se han añadido 4 operadores de mutación de expresiones: EAI, EUI, ENC y ECI. Por tanto, existe un conjunto de 30 operadores definidos para WS-BPEL. A partir de las tablas 3 y 4 se deduce que 17 (56,7%) de los 30 operadores para WS-BPEL son equivalentes a algún operador definido para los otros lenguajes.

Al clasificar los operadores en las categorías de los operadores WS-BPEL (ver sección 4) se observa que el 91,7% de los operadores de mutación de expresiones para WS-BPEL es equivalente a los de otros lenguajes (ver tabla 5). Este alto porcentaje de similitud puede atribuirse al uso de XPath, que ha heredado algunas características de los lenguajes tradicionales.

Tipo de lenguaje	*Lenguaje*	*Completamente equivalente (%)*	*Similar (%)*	*Total (%)*
Programación estructurada	C	17,6	41,2	58,8
	Fortran	23,5	29,4	52,9
	Ada	35,3	29,4	64,7
Programación orientada a objetos	C++	23,5	5,9	29,4
	C#	0	5,9	5,9
	ASP.NET	0	47	47
	Java	29,4	17,6	47
Específico del dominio	SQL	17,6	17,6	35,2
	XSLT	29,4	11,8	41,2
Categoría				*Total (%)*
Mutación de identificador				100
Mutación de expresión				91,7
Mutación de actividad				27,3
Mutación de condición excepcional y evento				33,3
Todos los lenguajes y categorías				56,7

Tabla 5. Proporciones de los operadores de mutación para WS-BPEL equivalentes a los de otros lenguajes.

El 27,3% de los operadores de mutación de actividades para WS-BPEL es equivalente a los de otros lenguajes. Sin embargo, no se incluye ningún operador

relacionado con la concurrencia (ver tabla 3). Esto se debe a que el enfoque para soportar concurrencia es bastante diferente.

Además, solo el 33,3% de los operadores de mutación de condiciones excepcionales y eventos para WS-BPEL es equivalente a uno de los operadores para otros lenguajes. Esta situación es lógica porque mutan manejadores que son específicos de WS-BPEL, por lo que no están disponibles en otros lenguajes.

Si atendemos a las proporciones de los operadores de mutación para WS-BPEL que son equivalentes a los de otros lenguajes, clasificados según el tipo de lenguaje del que se trata, la tabla 5 muestra que los mayores porcentajes se asocian a los lenguajes de programación estructurada, especialmente a Ada. Por el contrario, el porcentaje de operadores para C# equivalente a los de WS-BPEL es el más bajo. Por consiguiente, se puede concluir que WS-BPEL se asemeja más a los lenguajes de programación estructurada. Además, puede comprobarse que ninguno de los operadores de mutación para ASP.NET y C# es completamente equivalente a un operador para WS-BPEL y deben ser redefinidos con pequeños cambios para que sean equivalentes.

10. Conclusiones y trabajo futuro

En este trabajo se ha evaluado cualitativamente el conjunto de los operadores de mutación definidos por Estero et al. [3] [4] para WS-BPEL 2.0, y se ha comparado con los operadores de mutación definidos para los lenguajes tradicionales más conocidos. En concreto, se han incluido en el estudio los lenguajes de programación estructurada C, Fortran y Ada, los lenguajes de programación orientada a objetos C++, C#, ASP.NET y Java, y los lenguajes específicos del dominio SQL y XSLT.

Se han clasificado los operadores de mutación en cuatro categorías: operadores disponibles tanto para los lenguajes tradicionales como para WS-BPEL, operadores solo aplicables a WS-BPEL, operadores solo aplicables a los lenguajes tradicionales, así como los operadores para otros lenguajes que podrían aplicarse a programas WS-BPEL y que no están ya incluidos en el conjunto de operadores para dicho lenguaje.

A partir de las tablas 3 y 4 se concluye que solo 17 (56,7%) de los 30 operadores para WS-BPEL están disponibles en los lenguajes tradicionales. Esto confirma que WS-BPEL 2.0 tiene algunas características (y tipos de errores al utilizarlas) que son específicas, como

por ejemplo: soporte por defecto de bucles paralelos y serialización de accesos concurrentes de un conjunto de variables, sincronización declarativa basada en enlaces en lugar de hilos, manejadores especiales para deshacer invocaciones previas de servicios cuando se lanza un fallo y decidir si es necesario finalizar la ejecución de una parte del código, y elementos especializados en recibir mensajes de la red.

Por otro lado, WS-BPEL carece de algunas características comunes en otros lenguajes, como funciones, sentencias *switch* de valores enteros, vectores u orientación a objetos, entre otras. Esto se debe a que el propósito de los lenguajes estudiados es muy diferente: WS-BPEL se utiliza como un lenguaje de ejecución de procesos de negocio; C, Fortran y Ada como lenguajes de programación estructurada; C++, C#, ASP.NET y Java como lenguajes de programación orientada a objetos; SQL como un lenguaje de consulta, y XSLT como un lenguaje de transformación de documentos XML.

Este estudio concluye que puede mejorarse el conjunto de operadores de mutación para WS-BPEL. Se han incluido cuatro operadores de mutación de expresiones: dos que insertan código (EAI y EUI) y dos que modelan errores que pueden cometer los programadores al escribir a mano las expresiones XPath (ENC y ECI).

En el futuro, se pretende ampliar el conjunto de operadores para WS-BPEL con operadores que muten otras características de las expresiones XPath y con los operadores necesarios para medir algunos criterios de cobertura de código, como la cobertura de sentencia o de rama.

Referencias

[1] Dereziňska, A., "An experimental case study to applying mutation analysis for SQL queries". En: Ganzha, M. y Paprzycki, M. (Eds.), *Proceedings of the International Multiconference on Computer Science and Information Technology*. *Mrągowo (Polonia), 12-14 de octubre de 2009*, pp. 559-566, 2009.

[2] Alves, A. et al., Eds., "Web Service Business Process Execution Language (WS-BPEL) 2.0". Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[3] Estero-Botaro, A., Palomo-Lozano, F. y Medina-Bulo, I., "Mutation Operators for WS-BPEL 2.0". En: *Proceedings of XXI International Conference on Software & Systems Engineering and their Applications*. *Paris (Francia), 9-11 de diciembre de 2008*, 2008.

[4] Estero-Botaro, A., Palomo-Lozano, F. y Medina-Bulo, I., "Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions". En: Bilof, R. (Ed.), *Proceedings of III International Conference on Software Testing, Verification, and Validation Workshops*. *Paris (Francia), 6-10 de abril de 2010*, pp. 142-150, 2010.

[5] Jia, Y. y Harman, M., *An Analysis and Survey of the Development of Mutation Testing*. Informe técnico, King's College, London, Reino Unido, 2009.

[6] Agrawal, H. et al., *Design of Mutant Operators for the C Programming Language*. Informe técnico, Purdue University, West Lafayette, Indiana, 1989.

[7] Delamaro, M. E. y Maldonado, J. C., "Proteum - A Tool for the Assessment of Test Adequacy for C Programs". En: *Proceedings of the Conference on Performability in Computing System*. *East Brunswick (Nueva Yersey)*, *julio de 1996*, pp. 79-95, 1996.

[8] Barbosa, E. F., Maldonado, J. C. y Vincenzi, A. M. R., "Toward the determination of sufficient mutant operators for C", *Software Testing, Verification and Reliability*, vol. 11, nº 2, pp. 113-136, 2001.

[9] King, K. N. y Offutt, A. J., "A FORTRAN Language System for Mutation-based Software Testing", *Software - Practice and Experience*, vol. 21, nº 7, pp. 685-718, 1991.

[10] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H. y Zapf, C., "An experimental determination of sufficient mutant operators", *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 99-118, 1996.

[11] Offutt, A. J., Voas, J. y Payne, J., *Mutation Operators for Ada*. Informe técnico, George Mason University, Fairfax, Virginia, 1996.

[12] Zhang, H., "Mutation Operators for C++", 2003. (Web). En: http://people.cis.ksu.edu/~hzh8888/mse_project.htm. Último acceso: 28 de febrero de 2011.

[13] Dereziński, A., "Quality Assessment of Mutation Operators Dedicated for C# Programs". En: Kellenberger, P. (Ed.), *Proceedings of VI International Conference on Quality Software*. *Beijing (China), 27-28 de octubre de 2006*, pp. 227-234, 2006.

[14] Dereziński, A., "Advanced mutation operators applicable in C# programs". En: Krzystof, S. (Ed.), *Software Engineering Techniques: Design for Quality*, Springer, 2007.

[15] Derezinska, A. y Szustek, A., "Tool-Supported Advanced Mutation Approach for Verification of C# Programs". En: Werner, B. (Ed.), *Proceedings of III International Conference on Dependability of Computer Systems*. *Szklarska Poreba (Polonia), 26-28 de junio de 2008*, pp. 261-268, 2008.

[16] Mansour, N. y Houri, M., "Testing web applications", *Information and Software Technology*, vol. 48, nº 1, pp. 31-42, 2006.

[17] Ma, Y. S., Kwon, Y. R. y Offutt, J., "Inter-Class Mutation Operators for Java". En: Kawada, S. (Ed.), *Proceedings of XIII International Symposium on Software Reliability Engineering*. *Annapolis (Maryland), 12-15 de noviembre de 2002*, pp. 352-363, 2002.

[18] Offutt, J., Ma, Y. S. y Kwon, Y. R., "The class-level mutants of MuJava". En: Anderson, K. (Ed.), *Proceedings of the 2006 International Workshop on Automation of Software Test*. *Shanghai (China), 20-28 de mayo de 2006*, pp. 78-84, 2006.

[19] Ma, Y. S., Offutt, J. y Kwon, Y. R., "MuJava: a mutation system for Java". En: Anderson, K. (Ed.), *Proceedings of XXVIII International Conference on Software Engineering*. *Shanghai (China), 20-28 de mayo de 2006*, pp. 827-830, 2006.

[20] Ma, Y. S., Offutt, J. y Kwon, Y., "MuJava: An Automated Class Mutation System", *Software Testing, Verification and Reliability*, vol. 15, nº 2, pp. 97-133, 2005.

[21] Ma, Y. S., Kwon, Y. R. y Kim, S. W., "Statistical Investigation on Class Mutation Operators", *ETRI Journal*, vol. 31, nº 2, pp. 140-150, 2009.

[22] Bradbury, J. S., Cordy, J. R. y Dingel, J., "Mutation Operators for Concurrent Java (J2SE 5.0)". En: *Proceedings of II Workshop on Mutation Analysis*. *Raleigh (Carolina del Norte), 7-10 de noviembre de 2006*, pp. 83-92, 2006.

[23] Ji, C., Chen, Z., Xu, B. y Wang, Z., "A New Mutation Analysis Method for Testing Java Exception Handling". En: Kellenberger, P. (Ed.), *Proceedings of XXXIII Annual IEEE International Computer Software and Applications Conference*. *Seattle (Washington), 20-24 de julio de 2009*, pp. 556-561, 2009.

[24] Madeyski, L. y Radyk, N., "Judy - a mutation testing tool for Java", *IET Software*, vol. 4, nº 1, pp. 32-42, 2010.

[25] Tuya, J., Suárez-Cabal, M. J. y de la Riva, C., "Mutating database queries", *Information and Software Technology*, vol. 49, nº 4, pp. 398-417, 2007.

[26] Tuya, J., Suárez-Cabal, M. J. y de la Riva, C., "SQLMutation: A tool to generate mutants of SQL database queries". En: *Proceedings of II Workshop on Mutation Analysis*. *Raleigh (Carolina del Norte), 7-10 de noviembre de 2006*, 2006.

[27] Lonetti, F. y Marchetti, E., "X-MuT: A Tool for the Generation of XSLT Mutants". En: Werner, B. (Ed.), *Proceedings of VII International Conference on the Quality of Information and Communications Technology*. *Porto (Portugal), 29 de septiembre - 2 de octubre de 2010*, pp. 280-285, 2010.

[28] Kim, S., Clark, J. A. y McDermid, J. A., "Class Mutation: Mutation Testing for Object-Oriented Programs". En: *Proceedings of Conference on Object-Oriented Software Systems*. *Erfurt (Alemania), octubre de 2000*, pp. 9-12, 2000.