

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

Juan Boubeta Puig
Dpto. de Lenguajes y Sistemas
Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
juan.boubeta@uca.es

Inmaculada Medina Bulo
Dpto. de Lenguajes y Sistemas
Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
inmaculada.medina@uca.es

Antonio García Domínguez
Dpto. de Lenguajes y Sistemas
Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
antonio.garciadominguez@uca.es

Resumen

La prueba de mutaciones es una técnica de prueba de software basada en fallos. Para aplicarla, es necesario disponer de un conjunto de operadores de mutación, específicos para cada lenguaje, que serán los que realicen los cambios en el programa a probar. En este artículo se realiza un estudio sobre cuáles de los operadores de mutación definidos para C, Fortran, Ada, SQL, C++, C#, ASP .NET y Java son equivalentes a los operadores definidos para el Lenguaje de Ejecución de Procesos de Negocio con Servicios Web, WS-BPEL 2.0. Esta comparativa nos permitirá identificar posibles mejoras para los operadores de mutación definidos para WS-BPEL.

1. Introducción

El lenguaje WS-BPEL [20] permite crear procesos de negocio mediante la composición de Servicios Web (WS) preexistentes y ofrecerlos a su vez como WS. La importancia económica que están alcanzando las composiciones WS-BPEL obliga a prestar especial atención a la prueba de este tipo de software.

Una técnica apropiada para medir la calidad de conjuntos de casos de prueba es el análisis de mutaciones [21, 27] (*mutation analysis*). Esta técnica utiliza una serie de reglas de transformación (operadores de mutación) que permiten realizar cambios sintácticos en el programa original que se desea probar, obteniéndose programas similares denominados *mutantes*.

Aunque se han publicado diversos trabajos sobre la definición de operadores de mutación para distintos lenguajes, no hemos encontrado

ninguno que defina cuáles son las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes de programación.

Para ello, estudiaremos cuáles son las similitudes existentes entre el lenguaje WS-BPEL y algunos lenguajes de programación no orientados a objetos y orientados a objetos. La comparación de dichos operadores de mutación nos permitirá identificar, a priori, oportunidades de mejora para la definición de nuevos posibles operadores de mutación para WS-BPEL.

El resto del artículo se estructura de la siguiente forma. En la sección 2 se especifican las características principales del lenguaje WS-BPEL 2.0. En la sección 3 se introducen conceptos fundamentales del análisis de mutaciones y se realiza un estudio exhaustivo sobre algunos trabajos que tratan sobre la definición de operadores de mutación definidos para C, Fortran, Ada, SQL, C++, C#, ASP .NET y Java. La sección 4 muestra cuáles de los operadores de mutación de estos lenguajes son comparables con los definidos para WS-BPEL 2.0 y los resultados obtenidos. Por último, se presentan las conclusiones y el trabajo futuro.

2. El lenguaje WS-BPEL

WS-BPEL es un lenguaje basado en XML que permite especificar el comportamiento de un proceso de negocio basado en interacciones con WS. La estructura de un 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.
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso. Pueden definirse manejadores de fallos, que indican las acciones a realizar en caso de producirse un fallo interno o en un WS al que se llama. También se definen los manejadores de eventos, que especifican las acciones a realizar en caso de que el proceso reciba una petición durante su flujo normal de ejecución.
4. Descripción del comportamiento del proceso de negocio; esto se logra a través de las actividades que proporciona el lenguaje.

Todos los elementos definidos anteriormente son globales si se declaran dentro del proceso. Sin embargo, también existe la posibilidad de declararlos de forma local mediante el contenedor *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, manipulación de datos, etc.). Las actividades estructuradas pueden contener otras actividades y definen la lógica de negocio. 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.

Además, 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.

3. Antecedentes

El análisis de mutaciones es el proceso de medir la calidad de conjuntos de casos de prueba [11]. Para ello genera un gran número de programas, denominados *mutantes*, que pueden contener una o más diferencias con respecto al programa original, denominados mutantes de primer orden o *First Order Mutation* (FOM) y mutantes de orden superior o *Higher Order Mutation* (HOM), respectivamente [13]. Los mutantes se generan aplicando al código fuente un conjunto de reglas definidas previamente, los *operadores de*

mutación, que introducen pequeños cambios sintácticos basados en los errores que suelen cometer habitualmente los programadores, o bien pretenden forzar ciertos criterios de cobertura del código. Estos operadores introducen cambios en el programa a probar manteniendo su validez sintáctica.

Una vez generados, los mutantes se ejecutan sobre los casos de prueba; si la salida que produce el mutante es diferente de la que produce el programa original sobre un determinado caso de prueba, se dice que el mutante está muerto. En ocasiones, aparecen mutantes que siempre provocan la misma salida que el programa original, por lo que no va a existir ningún caso de prueba que permita matarlos; éstos se denominan *mutantes equivalentes*. Para medir la calidad de un conjunto de casos de prueba debemos eliminar los mutantes equivalentes, ya que ésta se va a calcular mediante la *puntuación de mutación* (*mutation score*), el cociente entre el número de mutantes muertos y el número de mutantes no equivalentes.

Se han publicado diversos trabajos que tratan sobre la aplicación de la prueba de mutaciones [13]. En este artículo nos centraremos en los trabajos que definen los operadores de mutación para lenguajes de programación no orientados a objetos y lenguajes orientados a objetos.

En cuanto a los operadores de mutación definidos para lenguajes no orientados a objetos, Agrawal y col. [1] definen un conjunto de 77 operadores de mutación para el lenguaje C. King y Offutt [15] definen un conjunto de 22 operadores de mutación para Fortran y los integran en la herramienta de análisis de mutaciones Mothra, Offut y col. [22] definen un total de 65 operadores de mutación para Ada.

Chan y col. [4] definen 7 operadores de mutación para el lenguaje SQL. Los que han definido Tuya y col. [26] para este lenguaje han sido integrados en una herramienta para la generación de mutantes. Shahriar y Zulkernine [23, 24] definen 9 operadores y Derezińska [8] define 22 operadores de mutación para SQL.

Existen otros trabajos sobre operadores de mutación definidos para los lenguajes de programación orientados a objetos: C++, C#, ASP.NET y Java.

Zhang [28] define 24 operadores de mutación para el lenguaje C++ y Derezińska [6, 7] define 40 operadores de mutación para el lenguaje C#.

Mansour y Hourí [19] han propuesto un conjunto de operadores de mutación para evaluar la calidad de las pruebas realizadas a aplicaciones Web desarrolladas en el entorno .NET.

Kim y col. [14] han aplicado la técnica HAZOP (Hazard and Operability Studies) a la definición de la sintaxis de Java para identificar desviaciones de las construcciones del lenguaje y, así, definir un conjunto de operadores de mutación para este lenguaje. Alexander y col. [2] definen un conjunto de operadores para introducir mutaciones en los objetos de Java; estos operadores son utilizados por la herramienta OME (Object Mutation Engine) para generar mutantes. Ma y col. [16] definen 26 operadores de mutación para el lenguaje Java, estos operadores han sido integrados en la herramienta MuJava [17]. Bradbury y col. [3] definen un total de 24 operadores específicos para el comportamiento concurrente de Java (J2SE 5.0). Smith y Williams [25] definen unos operadores clasificados en operadores a nivel de método y operadores de clase. Ji y col. [12] han definido 5 operadores para la captura de excepciones. Madeyski y Radyk [18] han definido un conjunto de operadores clasificados en cuatro categorías: tradicionales, propios del lenguaje, orientados a objetos y operadores de clase.

4. Equivalencias entre operadores

En esta sección se realiza un estudio sobre las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL 2.0 en [9, 10] con los definidos para los lenguajes de programación no orientados a objetos: C definidos en [1, 5], Fortran definidos en [15], Ada definidos en [22] y SQL definidos en [4, 8, 23, 24, 26], y para los lenguajes orientados a objetos: C++ definidos en [28], C# definidos en [6, 7], ASP .NET definidos en [19] y Java definidos en [3, 12, 14, 18, 25].

Los operadores definidos para WS-BPEL 2.0 se clasifican en cuatro categorías, de acuerdo con el tipo de elemento sintáctico de WS-BPEL con el que se relacionan. Las categorías se identifican con una letra mayúscula y son las siguientes: mutación de identificadores (I), mutación de expresiones (E), mutación de actividades (A), y mutación de condiciones excepcionales y eventos (X). Dentro de cada categoría se definen varios

operadores de mutación que se identifican mediante tres letras mayúsculas: la primera de ellas coincide con la que identifica la categoría a la que pertenece el operador, mientras que las dos últimas identifican al operador dentro de la categoría. La Tabla 1 muestra el nombre y una descripción de estos operadores de mutación para WS-BPEL 2.0 definidos en [9, 10], clasificados por categorías.

A continuación, presentamos los operadores equivalentes entre dichos lenguajes de programación. La Tabla 2 muestra para cada operador de mutación definido para el lenguaje WS-BPEL sus operadores equivalentes definidos para otros lenguajes de programación, agrupados en las categorías en las que se clasifican los operadores WS-BPEL. Nos encontramos, por tanto, ante un resumen de la comparativa entre dichas categorías que explicamos y desarrollamos a lo largo de esta sección.

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

En el lenguaje C tenemos dos posibles operadores equivalentes: VGSR y VLSR. VGSR cambia variables escalares globales que afecten a una función f mientras que VLSR cambia variables escalares pasadas como parámetros a dicha función. Para que podamos comparar estos operadores con ISV consideramos que mutan variables globales de un proceso WS-BPEL y pertenecientes al mismo ámbito (*scope*) de un proceso, respectivamente. La razón es que en el lenguaje WS-BPEL no existen funciones como sí ocurre en la mayoría de los lenguajes de programación.

En el caso del lenguaje SQL un posible operador equivalente a ISV es IRP. Este operador sustituye cada parámetro por otra referencia de columna, constante o parámetro (de tipo compatible) en una cláusula *SELECT*. En WS-BPEL no encontramos la sintaxis propia de un lenguaje de consulta estructurado, por tanto, carece de cláusulas como la de *SELECT*. En este caso, podemos compararlo con ISV si lo redefinimos como un operador que sustituye cada "variable", en lugar de un "parámetro", por otra "variable" de tipo compatible.

Tabla 1. Operadores de mutación para WS-BPEL 2.0

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 (+, -, *, <i>div</i> , <i>mod</i>) 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 (<i>and</i> , <i>or</i>) 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	
<i>Relacionados con la concurrencia</i>	
ACI	Cambia el atributo <i>createInstance</i> de las actividades de recepción de mensajes a no
AFP	Cambia una actividad <i>forEach</i> secuencial a paralela
ASF	Cambia una actividad <i>sequence</i> por una actividad <i>flow</i>
AIS	Cambia el atributo <i>isolated</i> de un <i>scope</i> a no
<i>No concurrentes</i>	
AIE	Elimina un elemento <i>elseif</i> o el elemento <i>else</i> de una actividad <i>if</i>
AWR	Cambia una actividad <i>while</i> por una actividad <i>repeatUntil</i> y viceversa
AJC	Elimina el atributo <i>joinCondition</i> de cualquier actividad en la que aparezca
ASI	Intercambia el orden de dos actividades hijas de una actividad <i>sequence</i>
APM	Elimina un elemento <i>onMessage</i> de una actividad <i>pick</i>
APA	Elimina el elemento <i>onAlarm</i> de una actividad <i>pick</i> o de un manejador de eventos
Mutación de Condiciones Excepcionales y Eventos	
XMF	Elimina un elemento <i>catch</i> o el elemento <i>catchAll</i> 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 <i>throw</i>
XER	Elimina una actividad <i>rethrow</i>
XEE	Elimina un elemento <i>onEvent</i> de un manejador de eventos

Los operadores OVV para C++, OVV para Ada y SVR para Fortran son equivalentes a ISV sin realizar ninguna modificación sobre sus definiciones. Puede observarse que dichos operadores para C++ y Ada comparten el mismo nombre.

Los *operadores de mutación de expresiones* para WS-BPEL sólo sustituyen un operador por otro del mismo tipo, puesto que Estero-Botaro y col. [9, 10] consideran que éste es el fallo que puede cometerse con mayor frecuencia; mientras que los operadores definidos para otros lenguajes

además de realizar sustituciones, insertan o eliminan nuevas expresiones.

El operador OAAN para C es equivalente al operador de mutación de expresiones aritméticas para WS-BPEL (EAA).

Existen dos posibles operadores de mutación de expresión aritmética para C++ equivalentes a EAA: IBO1 y AOR. IBO1 cambia cada operador binario (*, /, %) y de adición (+, -) por otro operador binario y de adición, y AOR sustituye cada uno de los operadores +, -, *, y / por otro operador.

Tabla 2. Operadores de mutación de otros lenguajes comparables con los definidos para WS-BPEL 2.0

Categoría	WS-BPEL	Lenguajes No Orientados a Objetos				Lenguajes Orientados a Objetos			
		C	Fortran	Ada	SQL	C++	C#	ASP .NET	Java
Mutación de Identificadores	ISV	VGSR VLSR	SVR	OVV	IRP	OVV			
Mutación de Expresiones	EAA	OAAN	AOR	EOR	AOR	IBO1 AOR		ORO	AORB AOR
	EEU								UOD
	ERR	ORRN	ROR	ERR	ROR	IBO2		ORO	ROR
	ELL	OLLN	LCR	ELR	LCR	IBO3		ORO	LCR
	ECN	CRCR	CRP	EDT				EMO	
Mutación de Actividades	EMD								MXT
	AIE	SSDL	ELSE ELSEIF					SMO	
Mutación de Condiciones Excepcionales y Eventos	AWR	SDWD SWDD		SWR SRW					
	XMF						EHR		CBD
	XTF			SER			ENC		

En el caso de Java los posibles operadores equivalentes a EAA son: AOR y AORB. El primero de ellos sustituye un operador aritmético por otro, mientras que el segundo establece la condición de que dichos operadores deben ser equivalentes, por tanto, AORB es una especialización del operador AOR.

Al igual que en Java, también tenemos el operador AOR para SQL y Fortran. Además de realizar una sustitución de un operador aritmético por otro, también puede ser sustituido por *leftop* y *rightop*, propios de estos lenguajes. Además, en Fortran, AOR también permite mutar por el operador ******.

EOR es el operador para Ada encargado de mutar las expresiones aritméticas que, además de los operadores aritméticos básicos, también consideran ****** y **REM**.

ORO es el operador de mutación de expresiones para ASP .NET. Este operador es más general que los operadores tratados anteriormente para otros lenguajes, puesto que sustituye un operador por otro operador o constante. Por tanto, este operador es útil para la mutación de expresiones de distinto tipo, y no sólo para la mutación de expresiones aritméticas.

Tan solo encontramos un operador equivalente para el operador EEU en el lenguaje Java: UOD. Los operadores del resto de lenguajes realizan la inserción de este operador o la sustitución de éste por otro. Sin embargo, en WS-BPEL no se considera el añadirlo porque modelaría un fallo poco frecuente.

Los operadores equivalentes para el operador ERR son: ORRN para C, IBO2 para C++, ROR para Java, SQL y Fortran, ERR para Ada y ORO para ASP .NET. En el caso de SQL y Fortran también se sustituye la expresión relacional por *falseop* y *trueop*.

Los operadores equivalentes para el operador ELL son: OLLN para C, IBO3 para C++, LCR para Java, SQL y Fortran, ELR para Ada y ORO para ASP .NET. En el caso de SQL y Fortran también se sustituye la expresión lógica por *falseop*, *trueop*, *leftop* y *rightop*. ELR cambia cada operador lógico, además de por *and* y *or*, por *xor*, *and then* y *or else*.

El operador de mutación para C equivalente a ECN es CRCR. Éste, además de aumentar o disminuir el valor de la constante en una unidad, lo hace por un número real. Sin embargo, este operador para C, al igual que para el resto de

lenguajes, no permite añadir o eliminar un dígito a la constante como sí hace ECN.

Los demás operadores comparables con ECN son los siguientes: EDT para Ada si consideramos únicamente mutaciones de una expresión constante, EMO para ASP .NET y CRP para Fortran. Este último operador, a diferencia del definido para WS-BPEL, incrementa o decrementa el valor de la constante en un 10% si es de precisión doble, y si es 0 se reemplaza por ,01 y -0,1.

El único operador existente para modificar una expresión de duración, además del EMD, es definido para Java: MXT. Éste modifica el parámetro opcional de tiempo de las llamadas de métodos *wait()*, *sleep()*, *join()* y *await()*. La mutación consiste en incrementar o decrementar el tiempo en un factor de 2 ($t/2$ o $t*2$). Mientras que el operador para Java modifica el parámetro tiempo de las llamadas de unos métodos determinados, el operador EMD definido para WS-BPEL modifica una expresión de tiempo. Cada uno en su contexto realiza una tarea similar, aunque en Java las mutaciones consisten en incrementar o decrementar el tiempo en un factor de 2 y en WS-BPEL consisten en cambiar el valor inicial por 0 o por la mitad.

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*.

Los operadores ELSE y ELSEIF definidos para Fortran son equivalentes al operador AIE.

Sin embargo, los operadores propuestos SSDL para C y SMO para ASP .NET también pueden considerarse equivalentes a AIE teniendo en cuenta que mientras que los dos primeros eliminan una sentencia contenida en el *else*, el definido para WS-BPEL elimina toda la rama especificada por el elemento *else*.

Existen dos operadores para C que podrían ser equivalentes a AWR: SDWD y SWDD, si consideramos que los cambios se van a realizar entre elementos *repeatUntil* y *while*, en lugar de *do-while*. SDWD cambia la sentencia *while* por un *do-while* y SWDD realiza la operación inversa. En este caso, mientras que Estero-Botaro y col. [9,

10] han definido un único operador, AWR, para realizar ambas mutaciones, Agrawal y col. [1] definen dos operadores distintos: SDWD y SWDD.

Los lenguajes de programación ofrecen una u otra estructura repetitiva, que como ya sabemos, no tienen el mismo comportamiento. Cabe destacar que mientras que, en WS-BPEL, *while* y *repeatUntil* son actividades, en los lenguajes tradicionales nos referimos a ellos como sentencias.

Al igual que ocurre con los operadores para C propuestos como equivalentes a AWR tenemos dos operadores para Ada: SWR que cambia la sentencia *while* por un *loop*, en lugar de la actividad *repeatUntil* considerada en AWR, y SRW que realiza la operación inversa.

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: de fallos, de eventos, de compensación y de terminación.

Los manejadores de fallos permiten especificar mediante los elementos *catch* las actividades a llevar a cabo en caso de que se produzca un fallo determinado, y mediante el elemento *catchAll* las relacionadas con cualquier otro fallo no especificado anteriormente.

Las excepciones son el mecanismo utilizado para propagar los fallos que se produzcan durante la ejecución de un programa. Por tanto, parece bastante lógico pensar que existan operadores de mutación de eliminación de los bloques *catch* y *catchAll* para la mayoría de los lenguajes de programación estudiados, que permitan modelar el olvido por parte del programador de incluir un *catch* para un fallo determinado o un *catchAll* para cualquier fallo. Sin embargo, sólo hemos encontrado dos operadores equivalentes a XMF: EHR para C# y CBD para Java.

En los lenguajes tradicionales, la instrucción *catch* o *catchAll* ya es en sí un manejador de excepciones, por tanto, no existe la posibilidad de que un único manejador de excepciones contenga varios bloques *catch* como ocurre con el manejador de fallos en WS-BPEL.

El operador 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 instrucción *throw* también tiene el mismo comportamiento.

El operador ENC para C# cambia un tipo de excepción utilizada en una sentencia *throw* o *catch* y el operador SER para Ada cambia el nombre de la excepción por otro. Ambos operadores son equivalentes a XTF.

Como podemos comprobar en la Tabla 2 existen 47 operadores de mutación definidos para otros lenguajes de programación que son equivalentes con los de WS-BPEL. El 60% de estos operadores, 28 de los 47 operadores, corresponde a los lenguajes de programación no orientados a objetos, y el resto a los orientados a objetos.

En cuanto a las categorías en las que hemos realizado nuestro estudio, el 62% de los operadores definidos para otros lenguajes pertenecen a la categoría de operadores de mutación de expresiones. Además, existe un equilibrio entre el número de operadores de mutación definidos para lenguajes de programación no orientados a objetos con los orientados a objetos, comparables con los de WS-BPEL para dicha categoría.

5. Conclusiones y trabajo futuro

Hemos realizado un estudio exhaustivo sobre los diferentes trabajos que tratan sobre la definición de operadores de mutación para los lenguajes de programación no orientados a objetos: C, Fortran, Ada y SQL, y los lenguajes orientados a objetos: C++, C#, ASP.NET y Java.

Además, hemos investigado sobre cuáles son las equivalencias existentes entre estos operadores de mutación y los definidos para WS-BPEL 2.0. No hemos encontrado ningún otro trabajo que realice este mismo estudio.

Podemos observar en la Tabla 2 que sólo 11 de los 25 operadores definidos para WS-BPEL, es decir, un 44%, son equivalentes a operadores definidos para otros lenguajes. De aquí se deduce que una gran parte de los fallos que podrían comerse al generar una composición WS-BPEL (teniéndose en cuenta que normalmente estas composiciones no se suelen escribir de forma directa, sino que se utilizan herramientas gráficas) no aparecerán al escribir código en otros lenguajes de programación. No hemos encontrado ningún operador de mutación para otros lenguajes que pueda mejorar o ampliar los definidos para WS-BPEL, teniendo en cuenta únicamente los operadores que modelan errores al escribir código, no los de cobertura. Por tanto, afirmamos que los

operadores de mutación que modelan dichos fallos para WS-BPEL [9, 10] son completos, y que los autores no han olvidado adaptar otros operadores de mutación de los lenguajes de programación estudiados para dicho propósito.

Nuestra intención en un futuro próximo es completar esta comparativa con los operadores de mutación definidos para WS-BPEL que no son aplicables a los definidos para otros lenguajes, y con los operadores definidos para otros lenguajes que no son aplicables a los definidos para WS-BPEL. En el primer caso, necesitaremos estudiar qué características tiene el lenguaje WS-BPEL que lo hacen diferente a otros lenguajes y, por tanto, podremos explicar por qué no existen operadores de mutación definidos para otros lenguajes equivalentes a, por ejemplo, XMT. En el segundo caso, tendremos que buscar las diferencias existentes entre los lenguajes de programación estudiados y WS-BPEL para deducir el porqué algunos de estos operadores no podrían ser aplicados a código escrito en WS-BPEL. Este estudio nos permitirá completar el conjunto de operadores WS-BPEL con otros que tengan en cuenta algunos criterios de cobertura.

Referencias

- [1] Agrawal, H., DeMillo, R., Hathaway, B., Hsu, W., Hsu, W., Krauser, E., Martin, R.J., Mathur, A., Spafford, E. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, Indiana, 1989.
- [2] Alexander, R., Bieman, J., Ghosh, S., Ji, B. Mutation of Java Objects. Proceedings of ISSRE '02, pp. 341-351, IEEE CS, 2002.
- [3] Bradbury, J.S., Cordy, J.R., Dintel, J. Mutation Operators for Concurrent Java (J2SE 5.0). Proceedings of MUTATION '06, IEEE CS, 2006.
- [4] Chan, W.K., Cheung, S.C., Tse, T.H. Fault-based testing of database application programs with conceptual data model. Fifth International Conference on Quality Software (QSIC '05), pp. 187-196, 2005.
- [5] Delamaro, M., Maldonado, J. Proteum – A Tool for the Assessment of Test Adequacy for C Programs. Proceedings of the Conference on Performability in Computing System (PCS '96), pp. 79-95, 1996.

- [6] Derezińska, A. Quality Assessment of Mutation Operators Dedicated for C# Programs. Sixth International Conference on Quality Software (QSIC '06), pp. 227-234. IEEE CS, Beijing, China, 2006.
- [7] Derezińska, A. Advanced mutation operators applicable in C# programs. Software Engineering Techniques: Design for Quality, pp. 283-288, 2007.
- [8] Derezińska, A. An experimental case study to applying mutation analysis for SQL queries. International Multiconference on Computer Science and Information Technology (IMCSIT '09), pp. 559-566, 2009.
- [9] Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I. Mutation operators for WS-BPEL 2.0. Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications (ICSSEA '08), Paris, France, 2008.
- [10] Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I. Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions. Proceedings of Mutation 2010, Paris, France, 2010.
- [11] Guglielmo, G.D. Mutation Testing Online <http://www.mutationtest.net>.
- [12] Ji, C., Chen, Z., Xu, B., Wang, Z. A New Mutation Analysis Method for Testing Java Exception Handling. Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09), pp. 556-561. Seattle, Washington, USA, 2009.
- [13] Jia, Y., Harman, M. An Analysis and Survey of the Development of Mutation Testing. Technical Report TR-09-06, CREST Centre, King's College London, 2009.
- [14] Kim, S., Clark, J., McDermid, J. The Rigorous Generation of Java Mutation Operators Using HAZOP. Technical Report, The University of York, 1999.
- [15] King, K.N., Offutt, A.J. A Fortran Language System for Mutation-Based Software Testing. Software Reliability Engineering, vol. 21(7), pp. 685-718, 1991.
- [16] Ma, Y.S., Kwon, Y.R., Offutt, J. Interclass Mutation Operators for Java. Proceedings of ISSRE '02, pp. 352-363, IEEE CS, 2002.
- [17] Ma, Y.S., Offutt, J., Kwon, Y.R. MuJava: An Automated Class Mutation System. Software Testing, Verification and Reliability, vol. 15(2), pp. 97-133, 2005.
- [18] Madeyski, L., Radyk, N. Judy – A mutation testing tool for Java. IET Software, vol. 4(1), pp. 32-42, 2010.
- [19] Mansour, N., Houri, M. Testing web applications. Information and Software Technology, vol. 48(1), pp. 31-42, 2006.
- [20] OASIS. Web Services Business Process Execution Language 2.0. Organization for the Advancement of Structured Information Standards, 2007. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-S.html>.
- [21] Offutt, A.J., Untch, R.H. Mutation 2000: Uniting The Orthogonal, Mutation Testing for the New Century, pp. 34-44, Kluwer Academic Publishers, Norwell, MA, USA, 2001.
- [22] Offutt, A.J., Voas, J., Payne, J. Mutation Operators for Ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.
- [23] Shahriar, H. Mutation-based testing of buffer overflows, SQL injections, and format string bugs. Master Thesis, University of Queen, 2008. <http://hdl.handle.net/1974/1359>.
- [24] Shahriar, H., Zulkernine, M. MUSIC: Mutation-based SQL Injection Vulnerability Checking. Eighth International Conference on Quality Software (QSIC '08), pp. 77-86, 2008.
- [25] Smith, B.H., Williams, L. Should software testers use mutation analysis to augment a test set? Systems and Software, vol. 82(11), pp. 1819-1832, 2009.
- [26] Tuya, J., Suárez-Cabal, M.J., de la Riva, C. Mutating Database Queries. Information and Software Technology, vol. 49(4), pp. 398-417, 2007.
- [27] Woodward, M.R. Mutation Testing –its Origin and Evolution. Information and Software Technology, vol. 35(3), pp. 163-169, 1993.
- [28] Zhang, H. Mutation Operators for C++ http://people.cis.ksu.edu/~hzh8888/mse_project.htm.