Actas de los Talleres de Ingeniería del Software y Bases de Datos

Volumen 4, Número 5, Septiembre de 2010

ISSN 1988-3455

PRIS 2010: Taller sobre Pruebas en Ingeniería del Software

Editor: Claudio de la Riva

©2010, Los Autores. Los derechos de copia están permitidos para propósitos académicos y privados. Es necesario el permiso expreso de los propietarios del copyright para su re-publicación

Presentación

En el presente número de las Actas de los Talleres de Ingeniería del Software y Bases de Datos se publican los trabajos presentados en la cuarta edición del Taller sobre Pruebas en Ingeniería del Software (PRIS 2010: http://in2test.lsi.uniovi.es/pris2010/) celebrado en Valencia, el 7 de Septiembre de 2010 en el marco de las Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2010).

El objetivo general del taller fue establecer un foro de encuentro para la discusión de las actividades relacionadas con la prueba del software, tanto a nivel industrial como académico y de investigación. En esta quinta edición se publican 8 contribuciones diferentes tras haber sido seleccionadas mediante un proceso de revisión por pares. En el marco del taller se impartieron también 2 charlas invitadas impartidas en el taller, de las que se recogen resúmenes en las presentes actas.

El contenido de las actas se agrupa en tres bloques. En el primero se presentan trabajos relativos a las pruebas en sistemas basados en Internet y Arquitecturas basadas en Servicios (SOAs) con estudios sobre la literatura en el campo de pruebas sobre SOAs y operadores de mutación sobre BPEL. El segundo bloque recoge trabajos con diferentes propuestas de investigación en el campo de las pruebas del software y con diferentes temáticas: pruebas combinatorias, oráculos de prueba, pruebas evolutivas, pruebas basadas en mutaciones y pruebas sobre interfaces gráficos. En el último bloque se presentan los resúmenes de dos charlas invitadas realizadas en taller. La primera de ella, impartida por Tanja Vos y Oscar Pastor, presenta el proyecto europeo FITTEST, cuyo objetivo es abordar los desafíos de la prueba del software en Internet desde una perspectiva basada en técnicas de búsqueda. La segunda, impartida por Ewout van Driel, presenta diferentes estrategias de pruebas basadas en prioridades del negocio que contribuyen a la mejora de los procesos globales de gestión de las pruebas.

Claudio de la Riva

Comité Organizador y de Programa de PRIS 2010

Claudio de la Riva - Universidad de Oviedo
Peter Hodgson – Procedimientos-Uno, S.L.
Ewout van Driel – Sogeti Spain
Fergus Flaherty – CSC
Juan Garbajosa - Universidad Politécnica de Madrid
Luis Fernández Sanz - Universidad de Alcalá de Henares
Macario Polo - Universidad de Castilla la Mancha
Javier Tuya - Universidad de Oviedo

Revisores Adicionales

María José Suárez-Cabal - Universidad de Oviedo
José García-Fanjul - Universidad de Oviedo

Contenidos

Internet y SOA

Equivalencias entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes 1
Juan Boubeta Puig, Inmaculada Medina, Antonio García Domínguez
Revisión Sistemática: Pruebas de Software para SOA con UML 9
Mª del Pilar Romay, Luis Fernández, Roberto Tejedor
Protocolo para la revisión sistemática de estudios sobre pruebas en SOAs con enlace dinámico 17
Marcos Palacios, José García-Fanjul, Javier Tuya

Propuestas de Investigación y Herramientas

Propuesta de optimización en la prueba de mutaciones en Java 25
Inmaculada Medina, Lorena Gutiérrez Madroñal, Juan José Domínguez Jiménez
Un portal web para investigación y docencia en testing combinatorio 33
Beatriz Pérez, Macario Polo
Oráculo de prueba automatizado para programas de procesamiento XML 43
Dae S. Kim-Park, Claudio de la Riva, Javier Tuya
Verificación de Datos en la GUI como un Aspecto Separado de las Aplicaciones 51
Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, Diego Sevilla Ruiz
Pruebas Funcionales Evolutivas en la Industria 59
Tanja Vos, Arthur Vaas, Beatriz Marín

Charlas Invitadas

FITTEST – Testeo del Internet del Futuro 67
Tanja. Vos, Oscar Pastor
Business Driven Test Management - Una buena práctica en la gestión de las pruebas 69
Ewout van Driel

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 Dereziňska [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 Dereziňska [6, 7] define 40 operadores de mutación para el lenguaje C#.

Mansour y Houri [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 (+, -, *, *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
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

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

		Lenguajes No Orientados a Objetos				**Lenguajes Orientados a Objetos**			
Categoría	**WS-BPEL**	**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	
	EMD								MXT
Mutación de Actividades	AIE	SSDL	ELSE ELSEIF					SMO	
	AWR	SDWD SWDD		SWR SRW					
Mutación de Condiciones Excepcionales y Eventos	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 cometerse 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] Dereziňska, 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] Dereziňska, A. Advanced mutation operators applicable in C# programs. Software Engineering Techniques: Design for Quality, pp. 283-288, 2007.

[8] Dereziňska, 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.

Revisión Sistemática: Pruebas de Software para SOA con UML

Mª del Pilar Romay
Dep.Ingeniería Sistemas Información y Tel.
Universidad San Pablo CEU
pilar.romay@gmail.com

Luis Fernández-Sanz
Dep. Ciencias de la Computación.
Universidad de Alcalá
luis.fernandezs@uah.es

Roberto Tejedor
Meta 4
robertota@meta4.com

Resumen

El objetivo de este trabajo es realizar un estudio del estado actual de la investigación en pruebas de software para arquitecturas orientadas a servicio, considerando de manera específica el uso de UML como lenguaje de modelado. En primer lugar, se revisa el contexto general del estudio, detallando las cuestiones más cruciales referidas tanto a SOA y el concepto de servicio, como al uso de UML y sus extensiones como lenguajes de modelado, aplicados a las pruebas de software y a los propios servicios. A continuación se plantea una revisión sistemática siguiendo el método de Kitchenham, en la que se realizan búsquedas sobre UML, SOA y las pruebas de software, indicando tanto las preguntas como los criterios de exclusión usados. Se muestra que la mayoría de los resultados se agrupan en los campos del *Model-Based Testing* y el más específico *Model-Driven Testing*, a los que se dedican la mayoría de los artículos existentes; éstos son muy recientes, datándose los más antiguos de 2007. Se concluye que el campo de investigación es aún joven, y en él hay lugar para aportaciones significativas. También se concluye que UML no es específicamente muy utilizado en este contexto por sus necesidades particulares, aunque sí lo es de manera indirecta mediante la definición de *profiles* específicos.

1. Introducción

La aparición hace unos años de los sistemas software orientados a servicios, así como la gran expansión que han sufrido en la actualidad, hace muy importante cualquier avance en materia de definición de metodologías o de automatización de procesos de desarrollo de estos sistemas, entre los que se encuentran, con una importancia vital, sus procesos de aseguramiento de la calidad.

Este proceso de aseguramiento de la calidad incluye un complejo proceso de elaboración (y modelado) de las pruebas a realizar. Dado que los sistemas orientados a servicios son distintos, o tienen ciertas particularidades respecto a los sistemas más habituales (orientados a objetos), se considera necesario hacer un estudio de su estado, a día de hoy, con el objeto de comprobar a qué nivel están definidos los procesos, estándares y metodologías para este tipo de sistemas.

La complejidad a la que se está llegando dentro de los desarrollos software hace necesaria la utilización de modelos que representen y reduzcan esa complejidad: de ahí que se considere el uso de lenguajes de modelado específicos. El lenguaje más representativo en el proceso de desarrollo de software actual es el Lenguaje Unificado de Modelado (UML), actualmente en su versión 2.1 [1]. Sus capacidades de extensión le permiten crear lenguajes específicos de dominio (DSLs) a través de diversos *profiles* y extensiones del metamodelo. Esto hace posible usarlo como herramienta dentro del campo de pruebas.

Partiendo de las consideraciones anteriores, este artículo presenta una revisión sistemática del área de Pruebas de Software para Arquitecturas Orientadas a Servicios (SOA), utilizando UML como lenguaje de modelado. Esta revisión se centra principalmente en artículos de revista obtenidos a través de las llamadas Bases de Datos de Conocimiento, cuyo contenido son artículos científicos y

publicaciones de revistas oficiales del sector de la ingeniería.

Para la revisión sistemática se ha tomado la metodología propuesta por Barbara Kitchenham en 2004 [2]. En esta metodología se considera una revisión sistemática como un método de investigación desarrollado para identificar, obtener, analizar, evaluar, e interpretar toda la investigación existente y relevante para un tema, interrogante, fenómeno o área de interés particular. Un aspecto importante que tiene que cumplir esta revisión es que debe realizarse de forma completa, apoyándose en estudios individuales previos que se van a denominar *estudios primarios*, generando luego la revisión o *estudio secundario*.

Este trabajo se organiza en cinco apartados. El segundo apartado, *Contexto de estudio*, establece las tres áreas de estudio de este artículo: la Computación Orientada a Servicios, a través de la Arquitectura Orientada a Servicios (SOA). Luego, el área relativa al modelado y los lenguajes de modelado, que centraremos en UML y en su aplicación a servicios. La última área de estudio es la que abarca específicamente a las pruebas dentro del campo de servicios.

El tercer apartado, *Proceso de la Revisión*, determina los pasos llevados a cabo para la realización de esta revisión. Se presenta la obtención de información; ésta se ha llevado a cabo en dos fases: la primera se centra en la obtención de información guiada por términos básicos y la segunda usa términos de búsqueda ajustados por los resultados obtenidos de la fase anterior. Se indican las fuentes de conocimiento de las cuales se ha extraído la información, la cláusula de búsqueda y criterios de inclusión y exclusión de los estudios primarios.

El cuarto apartado presenta los resultados, esto es, *los estudios secundarios*. Se validará la calidad de búsqueda de los estudios primarios y se presentarán los trabajos más relevantes en cuanto a su continuidad temporal dentro del campo de pruebas en SOA con UML.

2. Contexto de Estudio

El objetivo de este apartado es presentar una introducción a las principales áreas de estudio sobre las que gira esta revisión sistemática.

2.1. Computación Orientada a Servicios.

Una de las máximas autoridades en el campo, Michael Papazoglou, define SOA como "un estilo meta-arquitectónico basado en servicios débilmente acoplados que proporciona flexibilidad a los procesos de negocio de manera interoperable e independiente de la tecnología" [3]. El objetivo esencial es por tanto la *interoperabilidad*, y es consecuencia directa del *acoplamiento débil*. Ésta es a su vez consecuencia de los principios esenciales de los servicios SOA, esto es: (1) toda función (simple o compuesta) es descrita como un servicio; (2) todo servicio es independiente de los demás servicios, y sólo se comunica a través de su interfaz, y (3) toda interfaz de servicio puede ser invocada con un identificador único, e independientemente de si es local o remoto. Cada uno tiene más consecuencias: de los dos primeros nace el debate sobre la *composición de servicios* (orquestación vs. coreografía; vertical u horizontal); y del tercero las implicaciones sobre su *descubrimiento* (identificación, contratos, SLA, servicios semánticos), así como el dinamismo y las necesidades de adaptación que éste causa.

La definición del término SOA (*Service-Oriented Architecture*) resulta todavía confusa, a pesar de su popularidad; posiblemente porque se ha utilizado con distintos sentidos en distintos contextos. De manera habitual se utiliza al menos con tres significados distintos, aunque mutuamente relacionados. En primer lugar, SOA como paradigma de computación, esto es, como equivalente de SOC (*Service-Oriented Computing*) [4]. En segundo lugar, SOA como estilo arquitectónico, esto es, como el conjunto de características que debe reunir un diseño para poder calificarlo de este modo. Y en tercer lugar, SOA como referido exclusivamente a la tecnología específica de los llamados *web services*, esto es, los basados en protocolos y estándares como SOAP y WSDL, que sin duda hicieron popular al término; ésta recibe más apropiadamente el nombre de WSA (*Web Services Architecture*). En nuestro caso, y de manera general, se usará el término en la segunda acepción, aunque esto no lo disocia completamente de los otros dos.

SOA tiene un papel especialmente relevante en el contexto de la integración de aplicaciones, el clásico problema de la EAI (*Enterprise Application Integration*). En realidad, puede decirse que nació, como tal, con el objetivo de solucionar este problema. Se plantea SOA como la arquitectura de integración definitiva: todas las capacidades y necesidades de un Sistema de Información heterogéneo se describen como servicios, que se integran en una infraestructura común, el denominado bus de servicios o ESB (*Enterprise Service Bus*). De hecho, en este contexto, SOA y ESB son prácticamente sinónimos. Este enfoque tiene sus consecuencias a otros niveles: la concepción de todo el Sistema de Información a nivel de servicios relaciona a éstos de manera directa con el modelado y gestión de procesos de negocio (BPM, en ambos casos: *Business Process Modeling* o *Management*). Esto se desarrolla tanto a nivel técnico (en la estructura de una orquestación en BPEL, o un servicio semántico en OWL-S) como desde un punto de vista organizativo: en este contexto, el tradicional Gobierno de las TIC se transforma, de manera efectiva, en *SOA Governance*.

Si en lugar de usar SOA, con cualquiera de estas connotaciones, usamos de manera genérica el término servicio –sólo en su acepción moderna en Informática–, la perspectiva es aún más complicada. En su contexto más general, el que usa la SSME (*Service Science, Management and Engineering*), el concepto de servicio describe todo un nuevo paradigma de concepción de los Sistemas de Información, y no sólo de la Informática. En su enfoque más restrictivo, el término se reduce a la tecnología de los web services originales, que más arriba se han denominado como WSA. Por ejemplo, las populares referencias a la "muerte de SOA" se han de entender en este contexto: de hecho, a pesar de enmarcarse en un debate arquitectónico más amplio (SOA vs. REST).

Igualmente, se debe tener en cuenta el papel de los servicios y su composición, a nivel de usuario avanzado, tanto desde la perspectiva de la definición de *mashups* personalizados, como de la elaboración de las RIA (*Rich Internet Applications*), tanto en el contexto de la web como en el de las plataformas móviles (teléfonos, PDAs, etc.)

En resumen, los términos a considerar en una revisión sistemática distan de ser simplemente "SOA" o "servicio", ya que hay que considerar toda una multitud de matices; pero entendidos en su contexto, y teniendo en cuenta su interrelación, se adquiere una perspectiva mucho más amplia de todos los aspectos implicados en este estudio.

2.2. Sobre Modelado y el Lenguaje de Modelado Unificado (UML)

UML, concebido como un lenguaje unificado, se ha utilizado también como lenguaje "universal" de modelado, gracias a sus mecanismos de extensión pesada y ligera. Con estos últimos ha tomado la forma de *profiles* UML concretos, que le permiten definir Lenguajes de Modelado Específicos de Dominio (DSMLs).

Una alternativa a los *profiles* de UML es la de crear lenguajes a partir del metamodelo de alto nivel definido por el MOF, como es el caso del *Eclipse Modeling Framework* (EMF) de Eclipse. Esta herramienta ha sido la elegida en muchos desarrollos de software, lo que ha hecho que muchas aplicaciones no trabajen realmente con UML, aunque sí haya un grado de compatibilidad debido a que UML también utiliza el MOF.

En la literatura la utilización de UML en el proceso de pruebas es indiscutible. La utilización más frecuente es la generación de casos de prueba a través de los diagramas de secuencia y los diagramas de estado. Pero en muchos casos, la utilización de algunos diagramas de UML en el tema de pruebas va aún más allá: permiten modelar todo el proceso de pruebas, haciendo uso de un *profile* de pruebas definido desde UML. De hecho, el *profile* UTP [5] se está consolidado como un estándar de la OMG.

La utilización del modelado en el proceso de desarrollo de software permite la generación de forma automática de aplicaciones mediante las transformaciones de modelos, descritas utilizando lenguajes tales como ATL o QVT; éste es el enfoque definido por MDA (*Model-Driven Architecture*). Cabe preguntarse la existencia de alguna técnica o aproximación que utilice los principios de MDA en el campo de pruebas. En efecto así es; durante la búsqueda de

resultados sobre pruebas y SOA, se encuentran dos líneas relacionadas con el tema de modelado: Las pruebas basadas en modelos, *model-based testing*, y dentro de ésta, ligada específicamente a MDA, estaría el *model-driven testing*. Estas dos técnicas de pruebas también se han localizado en el proceso de búsqueda de esta revisión sistemática.

El tema de modelado aplicado al área de los servicios esta representando a través de una metodología propia de SOA, IBM SOMA (*Service-Oriented Modeling and Architecture*). Esta metodología utiliza una técnica de análisis y diseño denominada SOAD [6], adaptada de la orientación a objetos y ubicada dentro del marco de modelado SOMF (*Service-Oriented Modeling Framework*). La utilización de SOAD no rompe con el lenguaje de modelado unificado, a pesar de tener un lenguaje característico propio. Esto es, a nivel de aplicación, el uso de UML se mantiene; es en el nivel de arquitectura del sistema en el que se aplican consideraciones particulares propias de la orientación a servicios, utilizando el lenguaje de modelado SoaML, que está siendo a su vez considerado como estándar de la OMG. A su vez, SoaML no supone una ruptura con UML, sino una extensión a través del mecanismo de *profiles*.

2.3. Pruebas sobre Sistemas Orientados a Servicios

Las propiedades intrínsecas a la orientación a servicios hacen que el tratamiento de pruebas tenga matices diferentes a la orientación a objetos. Así, la dificultad en conocer el código y su estructura complica el poder establecer pruebas estructurales basadas en la cobertura de ejecución. Por ello resulta más compleja la realización de pruebas de caja blanca, si bien hoy existen propuestas interesantes como la de Bartolini [7].

Además de la dificultad anterior, si se consideran las pruebas no sólo a través del código, hay más aspectos que hacen difícil probar los sistemas orientados a servicios, como indica Canfora [8]. Su dinamismo y la adaptabilidad intrínseca complican las pruebas. El control de los servicios resulta complicado, debido a que como tal, no se integran en el sistema, sino que se ejecutan en una infraestructura independiente que controla el proveedor del servicio. La veracidad de las características de calidad y el coste que suponen las pruebas son otros de los aspectos que complican todo este proceso.

Algunas referencias de interés que cubren el tema de pruebas en el campo de servicios son las que resumen las propuestas de Canfora [8] y Bucchiarone [9], este último a nivel de composición de servicios.

Como se ha mencionado con anterioridad, dentro del campo se encuentran también los sistemas de pruebas basados en modelos, *model-based testing* (MBT) [10]. Tal como se indica a continuación, estas referencias fueron el punto de partida de la siguiente fase de búsqueda.

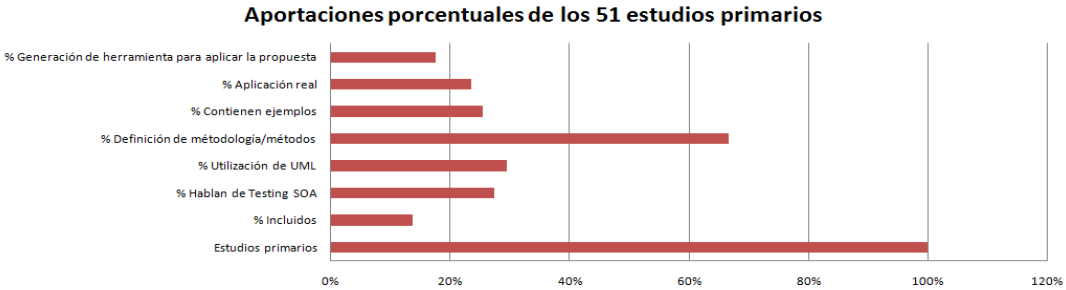
Los niveles a los que se aplican las pruebas en un contexto de servicios son principalmente a nivel de unidad, es decir a nivel del servicio, y a nivel de integración o composición de servicios, esto es, pruebas de integración.

3. Proceso de Revisión

3.1. Motivo y Fuentes de Conocimiento

El motivo que ha llevado a plantearse esta revisión es comprobar las posibilidades que existen, o pudieran existir, de utilizar modelos UML como base para diseñar pruebas de software en el ámbito de la arquitectura software orientada a servicios. Este interés está apoyado en una de las primeras propuestas para el ciclo de desarrollo de sistemas SOA donde se integraban modelos UML como soporte de análisis y diseño [11].

Para ello, la principal base de información sobre la que se ha trabajado, se ha obtenido de la Universidad de Alcalá de Henares (UAH), mediante el acceso a las Bases de Datos de artículos científicos. Ésta, a día de hoy, dispone de un total de más de 400.000 volúmenes, unos 5.300 títulos de publicaciones seriadas en diferente tipo de soportes y recursos electrónicos: acceso a bases de datos y un total de 13.000 títulos de revistas electrónicas. Es este último recurso el que se ha tomado como punto de partida para este estudio.



	Estudios primarios	% Incluidos	% Hablan de Testing SOA	% Utilización de UML	% Definición de métodología/métodos	% Contienen ejemplos	% Aplicación real	% Generación de herramienta para aplicar la propuesta
Series1	100%	14%	27%	29,41%	67%	25%	24%	18%

Figura 1. Aportaciones porcentuales de estudios primarios obtenidos en la primera fase

La búsqueda de la información se facilitó gracias a un Buscador, gestor de enlaces SFX, incorporado en el portal "MetAL" -que integra todos los recursos electrónicos de la UHA-, junto con los servicios suministrados por los editores y proveedores de las suscripciones.

La búsqueda de información se guió por las siguientes preguntas:

5. ¿Existen metodologías, procesos y herramientas definidas para el aseguramiento de la calidad mediante el estudio de las pruebas de software orientado a servicios?
6. ¿Cuáles son los métodos más conocidos que siguen los estándares de UML?
7. ¿Hay casos de estudio o casos de prueba realizados en estos sistemas?

El proceso de obtención de información se realizó en dos fases: En una primera fase se optó por buscar información con términos básicos implicados con la búsqueda. En una segunda fase, con información refinada obtenida de la primera fase, se optó por localizar información más específica, para mejorar las conclusiones y hacer un mejor *estudio secundario*. Así, la principal consulta para obtener los estudios primarios estuvo formada por las siguientes cláusulas lógicas, enlazadas por el operador OR:

- [("Test" OR "Testing") AND ("SOA" OR "Service oriented arquitecture" OR "Web service" OR "UML" OR "unified modeling language" OR "web" OR "methodology")].
- [("SOA" OR "Service oriented arquitecture") AND ("UML" OR "unified modeling language")].
- [("Web service") AND ("UML" OR "unified modeling language")].

En un primer momento con estas cláusulas pretendíamos conseguir la mayor información posible dentro del campo. A posteriori, y dado que, en otras áreas lejanas al campo de la ingeniería software se hace uso de este tipo de términos, se optó por restringir el campo de estudio al ámbito de tecnología de la información y considerar sólo los estudios posteriores al 1995, fecha en la que aparece la primera versión de UML.

En la segunda fase la consulta se realizó con las cláusulas siguientes, tras el análisis de los resultados obtenidos en la primera Fase:

- ("Model based testing" AND "Service Oriented") y sus variantes.
- ("Model driven testing" AND "Service Oriented") y sus variantes.

En esta segunda fase los resultados se focalizaron en búsquedas de los editores que se presentan a continuación, ya que estos habían sido los que habían producido resultados en la fase anterior: ACM (Association for Computing Machinery), IEL/IEEE (Institute of Electrical and Electronics Engineers), ScienceDirect (Elsevier), SpringerLink (Kluwer).

3.2. Criterios de Inclusión y Exclusión

La revisión de los estudios primarios se hizo de forma exhaustiva haciendo en un primer momento una lectura rápida, que permitió decidir el grado de profundidad de la lectura según se ajustara o no a los criterios de inclusión o exclusión. Independientemente de esto, para la realización de la revisión se procedió a la realización de una ficha de cada uno de los estudios primarios obtenidos.

Los criterios de inclusión se centraron en artículos que trataban aspectos de "test" o aseguramiento de la calidad para aplicaciones orientadas a servicios, haciendo mención directa de aquellos artículos que utilizan UML en el ámbito de pruebas. La lista de tópicos relevantes para la inclusión se recoge en la siguiente lista:

- Test para SOA.
- Métodos para el proceso de "testing" en SOA.
- Estándares para el proceso de "testing" en SOA.
- UML en la fase de pruebas.
- Referencia y ejemplifica modelos UML para SOA y su desarrollo.
- Propone herramientas para el uso de UML en pruebas para SOA.

4. Resultados y Análisis de la Revisión.

En este apartado se presentan los resultados obtenidos de la primera y segunda Fase, finalizando con un resumen del *estudio secundario* obtenido.

4.1. Fase 1: Resultados y Análisis

Los resultados de la revisión se presentan en la Figura 1. Los campos que se tienen en cuenta tanto para esta figura como los de la tabla 1 son:

- **Nº**: Hace referencia a la clave de revisión de estudio primario.
- **Título**: Es el título completo del artículo.
- **Inclusión**: Hace referencia a si cumple los requisitos de inclusión
- **SOA Testing**: Indica si se utilizan pruebas para aplicaciones SOA.
- **UML**: Si utilizan el lenguaje de modelado unificado.
- **Metodología**: Propone una metodología o un método de actuación.
- **Aplicación práctica**: Documenta ejemplos o casos de aplicación
- **Herramientas**: Hace uso o construye alguna herramienta para el método

Se excluyeron del estudio detallado, en la primera fase, aquellos artículos que sólo hacían referencia a procesos, pero no dan información adicional de cómo se lleva a cabo las pruebas. No se excluyeron los artículos que, tratando el tema de pruebas en orientación a servicios, no utilizaban UML. El objeto de no haberlos descartado en esta primera fase fue poder comprobar si UML se había descartado por la utilización de otros lenguajes o otros métodos. En la segunda fase se excluyeron de la búsqueda aquellos que no estaban directa o indirectamente ligados con UML

Así para el 100% de los estudios primarios obtenidos sólo se consideraron incluidos el 14%. El 27% habla de "testing" con SOA, el 29,41% utiliza UML, el 67% hace una metodología o aplica un método, el 25% contiene ejemplos, el 24 % aplica los resultados sobre un entorno real.

Para el estudio se utilizaron un total de 21 editores de los 44 disponibles y de las 1577 revistas publicadas por dichos 44 editores, al final fueron útiles un total de 1545. Así, un total de un 47,72 % de los editores fue útil para nuestro estudio y del total de revistas que nos proporcionaron dichos editores, se utilizaron para el estudio un total del 97,97%.

De los 1545 artículos útiles y descartando los que trataba los términos SOA, test, UML de forma lateral, se obtuvieron un total de 51 artículos. De estos, y tras un análisis para considerar aquellos que incluyeran métodos, o metodologías estándares o aplicaciones prácticas y herramientas, se obtuvieron un total de 7 estudios relevantes. Sólo dos [12], [13] incluían todos los aspectos propuestos, sin considerar la utilización de herramientas.

4.2. Fase 2 : Resultados y Análisis

Una característica común a ambos artículos es que se centran en las técnicas de pruebas basadas en modelos, *model-based testing*, al que se ha hecho referencia en el apartado 2.3. En el artículo [12] la primera parte está dedicada a la definición de la propuesta a grandes rasgos y la

Título	Autores	Descripción
Hybrid test of web applications with webtest (2008)	Harald Raffelt et al	-Sistemas de pruebas estático y Sistema de pruebas dinámico con técnicas de aprendizaje basadas en autómatas. -Considera pruebas de caja negra. Futuro considera posibilidad de incorporar TTCN-3 a la herramienta WebTest, - Aplicación en RIAS sin servicios
Engineering rich internet applications with a model-driven approach (2010)	Piero Fraternali et al	Webtest herramienta utiliza dos framework: -Jabc cubre aproximacion model-drive y orientada a servicios - Proceso de prueba dínámico, facilidad de almacenar, reproducir, integrar y recoger información.
Towards Adaptive Test Code Generation for Service Oriented Systems (2009)	Felderer, M., et al	- *Profile* propio orientado al negocio concreto compatible con UTP por transformación sin pasar por TTCN-3 y adaptable a SOAML (base de aplicación en UML). Comprobaciones de consistencia/recubrimiento, generación, ejecución y análisis pruebas. Se ejecutan automáticamente por un *framework* caso concreto FIT. Transformación de modelos y Aserciones
Testing Service-Oriented Architecture Applications (2007)	Paul Baker et al	- Profile UML para Pruebas UTP. -Frameworks TTCN-3 y el JUnit test framework para Java. -Demuestra como usar UTP con el framework TTCN-3 y el framework de prueba JUnit para Java
A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services (2007)	Marc Lohmann, Leonardo Mariani y Reiko Heckel	UML+Transformaciones de grafos y Restricciones OCL para Java modeladas con JML. 1. Extiende y aplica *domain-based testing* y técnica de flujo de datos para transformación de grafos. 2.Genera pruebas ejecutables predecibles desde la transformación de grafos. 3. Prueba y valida automáticamente servicios web, antes de incorporarlos al sistema
Model-based functional conformance testing of web services operating on persistent data (2006)	Avik Sinha, Amit Paradkar	Propone un sistema de *model based test case* para la generación de casos de prueba que pemite generate functional conformance casos de prueba para servicios web basado en Extended Finite State Machine (EFSM). Primero se especifican con el estándar WSDL-S, y luego se traslada a EFSM.

Tabla 1. Selección de artículos de la fase dos, complementarios a los MBT y MDT del texto

definición de todos los conceptos que rodean a la generación del Framework de testeo.

En primer lugar se hace una introducción al lenguaje *Testing and Test Control Notation* (TTCN-3). El Framework propuesto consta de una serie de casos de test para la evaluación de la carga y funcionamiento de servicios Web.

En [13] la aplicación de la propuesta, en torno a la que gira todo el artículo, es el uso de las técnicas y el framework propuesto sobre la aplicación *The eHealth Alarm Dispatcher*, encargada de comunicar el dispositivo móvil de una persona con los servicios médicos mediante llamadas a Web Services.

El artículo describe y define el *framework* de testeo que proponen desde este proyecto, así como el modelado de sistemas a través de la representación en diagramas denominados *Symbolic Transition System* (STS).

La información para generar estos diagramas se obtiene de la definición de los servicios Web, normalmente como un fichero WSDL; para ello, se ha desarrollado la herramienta MINERVA.

Como punto final se encuentra la descripción de una herramienta desarrollada en Java, Jambition, que va a generar los casos de test a partir de los diagramas STS. Todo ello se realiza dentro del Proyecto PLASTIC.

Los resultados de esta segunda fase están ligados a las dos técnicas de pruebas: *model-based* , y *model-driven testing* (MBT, MDT).

En esta segunda fase, que se reorienta la búsqueda hacia las dos técnicas anteriores con servicios y UML, se obtuvieron un total de 55 artículos. En este caso la búsqueda se centra en los 4 editores más representativos encontrados en la primera fase. Se comprueba que todos ellos están en un margen temporal entre 2006 y el 2010. En esta búsqueda se obtuvieron un total de 9 artículos directamente implicados con nuestro estudio, de los 55 obtenidos en la búsqueda de esta segunda fase. El mayor número de artículos se concentra en torno al 22% entre el año 2008 al 2010. Dentro de los artículos se puede considerar una línea de trabajo continuada en el grupo de artículos firmados por Antonia Bertolino y los miembros de los proyectos

TAROT y PLASTIC. Este grupo trabaja en la línea de pruebas a nivel de integración y de sistema con descripciones arquitectónicas y modelos UML. Dentro de este grupo una referencia interesante es la de SOA Governance [15]. Lo más interesante es su metodología de trabajo sustentada por herramientas.

Otra línea de interés, dentro del *model-driven testing*, es en la que trabaja el grupo de Ina Schieferdecker [16]. Las líneas en las que trabajan es en la especificación utilizando UML, en *Message Sequence Charts* (MSC) y en *Testing and Test Control Notation (*TTCN-3). Otras aproximaciones de MBT y MDT se presentan en la Tabla 1[1].

5. Conclusiones

El campo de pruebas en la orientación a servicios es un área aún por explorar. Las características peculiares de estos sistemas no permiten la aplicación directa de técnicas tradicionales, sin más cambios. Se requiere hacer adaptaciones en el campo de pruebas a través de modelos, donde se aprovechan las capacidades de los diferentes lenguajes y sus ampliaciones como es el caso de UML a través de los *profiles* o los metamodelos. Se han de considerar los lenguajes implicados a diferentes niveles; para el caso de modelado de aplicación podríamos aplicar UML y adaptaciones, pero para el nivel de arquitectura del sistema se requeriría el uso de lenguajes específicos, donde el más significativo es BPEL.

Agradecimientos

Los autores agradecen el apoyo del proyecto TIN2007-67843-C06-01 para este trabajo.

Referencias

[1] Kitchenmam, B., *Procedures for performing systematic review*. 2004, Departament of Computer Science, Keele University.

[2] Papazoglou, M. P. *Web Services: Principles and Technology*. Pearson, 2008

[3] Papazoglou, M. P. *Web Services: Principles and Technology*. Pearson Pearson, 2008.

[4] Bell, M. Service-Oriented Modeling: Service Analysis, Design, and Architecture. John Wiley & Sons, 2008.

[5] Object Management Group. RFP for a UML Testing Profile http://www.omg.org/

[6] Papazoglou, M. P.; Georgakapoulos, G. *Service-Oriented Computing*, Comnications of the ACM, 46(10), pp.24-28, Oct. 2003.

[7] Bartolini, C,; Bertolino, ; Elbaum, S and Marchetti, E. Whitening SOA testing. In V. Issarny, editor, ESEC-FSE '09. ACM, 2009.

[8] Canfora,G and Di Penta, M. "Service-Oriented architectures testing: a survey," LNCS, 2009, vol. 5413

[9] Bucchiarone, A.; Melgratti, H.; Severoni, F. Testing service composition. In: (ASSE'07), Mar del Plata, Argentina, 2007.

[10] Dias Neto, A.C; Subramanyan, R; Vieira, M and Travassos, GH. A survey on model-based testing approaches: a systematic review. In WEASELTech '07, pages 31, 36, Atlanta, Georgia, 2007. ACM

[11] Erl, T.: Service-Oriented Architecture: Concepts, Technology & De, Pearson, 2005.

[12] Schieferdecker, I et al. Distributed Functional and Load tests for Web services, (STTT), Springer 2004

[13] Frantzen, L. et al. On-The-Fly Model-Based Testing of Web Services with Jambition, Workshop, WS-FM 2008, Milan, Italy, September 4-5, 2008, Springer-Verlag, Berlin, Heidelberg, 2009

[14] Bartolini, C.: Bertolino, A.; Elbaum, S. and Marchetti, E. Whitening SOA testing. In V. Issarny, editor, ESEC-FSE '09. ACM, 2009.

[15] Bertolino, A and Polini, A. SOA test governance: enabling service integration testing across organization and tech. borders. In Wor. on Web Testing, pp. 277-286, 2009.

[16] Zander, J., Dai, Z.R., Schieferdecker, I., Din, G.: From U2TP models to executable tests with TTCN-3: an approach to model driven testing. In: Testing of Communicating Systems. LNCS, vol. 3502/2005, page. 289–303. Springer (2008).

[1] Las referencias de la tabla no están listadas en el apartado de referencias por problemas de espacio.

Protocolo para la revisión sistemática de estudios sobre pruebas en SOAs con enlace dinámico

Marcos Palacios[1], José García-Fanjul[2], Javier Tuya[2]
Departamento de Informática
Universidad de Oviedo
Campus Universitario de Viesques
33204 Gijón
[1] palacios@lsi.uniovi.es
[2] {jgfanjul, tuya}@uniovi.es

Resumen

Las características inherentes a las Arquitecturas Orientadas a Servicios (SOAs), como el enlazado dinámico de los servicios en tiempo de ejecución, suponen un reto para los investigadores ya que dificultan el proceso de pruebas del software basado en este paradigma de desarrollo. Con el objetivo de identificar y clasificar los trabajos existentes en el campo de las pruebas en SOAs con enlace dinámico, se ha elaborado un protocolo para guiar el desarrollo de revisiones literarias sistemáticas. En el protocolo se especifican tanto las estrategias de búsqueda y selección de los artículos como la posterior extracción y síntesis de la información recopilada. En este artículo se describen todos los elementos del protocolo desarrollado y un resumen de los resultados obtenidos tras la revisión sistemática.

1. Introducción

La notable importancia que ha adquirido el desarrollo de software basado en servicios se refleja en las grandes cantidades de dinero invertidas por las empresas para adaptarse a este modelo arquitectónico [19]. En este proceso de implementaciones basadas en SOA, se estima que el esfuerzo dedicado a la prueba de este tipo de software supone el 25% del total del proyecto [10]. Las particulares características de las aplicaciones SOA presentan nuevos retos a los investigadores cuyo objetivo es mejorar el proceso de pruebas del software basado en servicios [6]. La posibilidad de descubrir y enlazar los servicios en tiempo de ejecución dificulta el proceso de pruebas y, más concretamente, las pruebas de integración de los potenciales servicios a invocar en composiciones [1][5][7].

El conocimiento del estado del arte de las pruebas en SOAs con enlace dinámico permite la identificación de potenciales líneas de investigación que no han sido aún suficientemente abordadas. Las revisiones sistemáticas de literatura (SLRs) son un método utilizado para conocer, evaluar e interpretar la información existente en un determinado campo de investigación. Para llevar a cabo una SLR es necesario el desarrollo de un protocolo donde se definan, claramente y de forma inequívoca, los objetivos de la revisión y las estrategias de búsqueda y selección de artículos así como de extracción y síntesis de la información contenida en dichos trabajos.

En este artículo se describe el protocolo diseñado para llevar a cabo el proceso de revisión de los trabajos existentes en el ámbito de las pruebas del software basado en SOA donde el enlazado de los servicios se hace dinámicamente. También se discute la experiencia obtenida en la aplicación del protocolo y se presenta un resumen de los resultados alcanzados tras el desarrollo de la revisión.

El artículo presenta la siguiente estructura: en la sección 2 se describen los conceptos básicos tanto de las SLRs como de las pruebas del software, identificando SLRs existentes en este ámbito. En la sección 3 se describen los elementos del protocolo desarrollado. En la sección 4 se discute la experiencia obtenida aplicando el protocolo en el proceso de revisión. En la sección 5 se resumen los resultados de la revisión mientras que, finalmente, en la sección 6 se presentan las conclusiones obtenidas.

2. Fundamentos

En este apartado se describe la aplicación de las SLRs en el ámbito de la Ingeniería del Software así como la definición de pruebas del software adoptada para la elaboración del protocolo.

2.1. SLRs

Las SLRs son uno de los procedimientos más usados en la Ingeniería del Software basado en Evidencias (EBSE) y proporcionan un método sistemático y objetivo para buscar y analizar la literatura existente en un campo de investigación determinado [4]. En el ámbito de las pruebas del software pueden consultarse resultados de SLRs en [1][9][12].
Dependiendo de los objetivos de la revisión hay distintos tipos de SLRs, siendo los más comunes la SLR convencional y el mapping study. Una SLR convencional trata de responder a un conjunto de preguntas muy específicas sobre los resultados alcanzados en un tema de investigación concreto mientras que un mapping study trata de identificar y clasificar toda la literatura existente en un campo de investigación. Las principales diferencias entre ambos tipos de SLR pueden consultarse en [15]. Dado que el objetivo de esta investigación es conocer el estado del arte en las pruebas del software basado en SOA con enlace dinámico, el protocolo presentado en este artículo está diseñado para llevar a cabo un mapping study.

En el ámbito de la Ingeniería del Software se ha extendido el uso de un conjunto de pautas, publicadas por Kitchenham et al. en [13] y posteriormente refinadas en [14], para llevar a cabo SLRs. Según estas pautas, el proceso de revisión se divide en tres fases diferenciadas:

1. Planificación de la revisión.
2. Desarrollo de la revisión.
3. Publicación de resultados.

En la primera de las fases es donde se lleva a cabo el diseño del protocolo que guiará el resto de fases. Este protocolo debe incluir los siguientes elementos:

- Conjunto de preguntas de investigación a responder mediante el mapping study.
- Estrategia de búsqueda de los artículos.
- Estrategia de selección de los artículos.
- Criterios para el estudio de la calidad de los artículos.
- Estrategia de extracción de datos.

El protocolo no es un artefacto estático que se diseña antes de llevar a cabo la revisión y permanece inalterable durante el desarrollo de la misma. A medida que se avanza en la segunda fase puede ser necesario modificar o refinar alguno de los apartados del protocolo en función de la información que se va obteniendo en la revisión.

2.2. Pruebas del software

El concepto de pruebas (testing) ha sido usado con diferentes acepciones en la literatura en el ámbito de la Ingeniería del Software. En [11] se define testing como una actividad en la cual el sistema es ejecutado bajo unas condiciones especificadas, los resultados son observados y un veredicto es proporcionado en función de dichos resultados. La información observada puede permitir la detección de fallos en el sistema bajo prueba o la toma de decisiones basadas en los resultados de las pruebas. Uno de los enfoques de pruebas más utilizado en la actualidad en SOAs es la observación pasiva de las invocaciones de los usuarios en un entorno de producción y ha sido definido como on-line testing [2] o monitorización. Alineando estos conceptos con el ámbito de las pruebas en SOAs, se consideraron para revisión todos aquellos trabajos que impliquen la ejecución de los servicios, tanto con el objetivo de detectar fallos como para recopilar información y usarla para tomar una decisión.

3. Protocolo

En este apartado se describe el contenido de cada uno de los elementos del protocolo diseñado, argumentando las decisiones tomadas durante el desarrollo de dicho protocolo.

3.1. Preguntas de investigación

La especificación de las preguntas de investigación es la actividad más importante en

cualquier revisión sistemática ya que su formulación guía el diseño del resto de elementos del protocolo y, por tanto, el proceso completo de revisión. Con el objetivo de identificar el estado actual de la investigación en pruebas del software basado en SOA, donde el enlazado de los servicios se lleva a cabo dinámicamente, las preguntas de investigación formuladas son las especificadas como RQ1-RQ6 en la Tabla 1. Además de responder a este conjunto de preguntas de carácter técnico, también se ha considerado relevante identificar la actividad reciente que ha habido en la temática de investigación abordada. Por tanto, se han definido dos preguntas adicionales (RQ7-RQ8).

La formulación de estas preguntas de investigación está alineada con los distintos niveles de pruebas identificados por Bertolino en [2]. La pregunta RQ1 aborda el *por qué* de la investigación realizada mientras que la pregunta RQ2 identifica *cómo* se están ejecutando las pruebas. Las preguntas RQ3 y RQ4 responden a *dónde* se ejecutan las pruebas y *cuándo* se está llevando a cabo este proceso. La identificación de las tecnologías y estándares en RQ5 permite saben *qué* se está investigando. Finalmente, mediante la respuesta de RQ7 es posible determinar *cuánto* esfuerzo está siendo dedicado a las pruebas en el ámbito de las aplicaciones SOA con enlace dinámico.

RQ1	¿Cuál es el objetivo principal de la investigación?
RQ2	¿Qué métodos o técnicas de pruebas están siendo aplicados?
RQ3	¿Quién está participando en el proceso de pruebas?
RQ4	¿En qué momento se están ejecutando las pruebas?
RQ5	¿Qué estándares o tecnologías están siendo usados?
RQ6	¿Qué método se está aplicando para validar la investigación?
RQ7	¿Cuánta actividad ha habido durante los últimos años respecto a la prueba de SOAs con enlace dinámico?
RQ8	¿Dónde han sido publicados los trabajos?

Tabla 1. Preguntas de investigación

3.2. Estrategia de búsqueda

El proceso de búsqueda es el conjunto de tareas llevadas a cabo con la finalidad de identificar los trabajos relevantes existentes en la temática de investigación fijada. La estrategia de búsqueda define las fuentes de información que van a ser usadas y la selección de los términos adecuados para la búsqueda.

Para llevar a cabo una búsqueda lo más exhaustiva posible, se ha optado por una estrategia consistente en tres fases diferenciadas, tal como se refleja en la Figura 1. En la primera fase se hará una búsqueda manual en las revistas, conferencias y workshops más representativos, alineados con la temática de investigación abordada. La selección de estas fuentes se ha realizado teniendo en cuenta el índice de impacto de las revistas indexadas en el JCR y los rankings CORE de conferencias y workshops. También se han incluido otras fuentes donde se han publicado previamente trabajos relacionados con las pruebas del software y las SOAs. El conjunto final de fuentes de información está incluido en la Tabla 2. Para llevar a cabo esta búsqueda manual se utilizará la biblioteca digital DBLP, donde se encuentran listados casi todos los trabajos publicados en las citadas fuentes. Así mismo se ha fijado como fecha de inicio de la búsqueda el año 2000, fecha en la que el consorcio W3C publicó la primera especificación tanto del estándar WSDL para la descripción de la interfaz de los servicios como UDDI para la implementación de las funciones del registro.

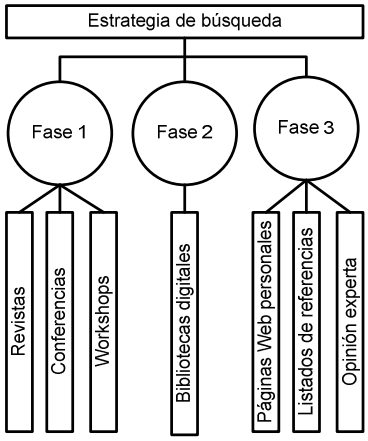


Figura 1. Estrategia de búsqueda

Revistas	ACM Computing Surveys, ACM TOSEM, ESE, IEEE Computer, IEEE Internet Computing, IEEE Software, IEEE TSE, IJSEKE, JWSR, IST, JSS, Sigplan Notices, SIGSOFT, SQJ, STVR
Conferencias	ASE, CAV, COMPSAC, ECOWS, EDOC, ESEM, ESEC, ETAPS, FORTE, FSE, ICSE, ICSOC, ICST, ICWE, ICWS, IEEE Services, ISSTA, QSIC, SAC, SCC, SEKE, TAIC PART, TAP, TESTCOM, WWW
Workshops	A-MOST, AST, FATES, Mutation, SBST, SOSE, SPIN, TAV-WEB, WESOA, WS-Testing

Tabla 2. Revistas, conferencias y workshops

Durante la segunda fase, se realizará una búsqueda automática usando las bibliotecas digitales especificadas en la Tabla 3. Para llevar a cabo este proceso, se han seleccionado los términos adecuados atendiendo a tres puntos de vista, tal como se recomienda en [14]: población, intervención y resultados. La población representa el área de aplicación de la investigación a revisar. En este caso, se identifican los términos de búsqueda correspondientes a la población desde dos perspectivas: un primer conjunto de términos relacionados con las arquitecturas SOA (Población1) y un segundo conjunto con términos más específicos que presentan relación con el proceso de enlazado dinámico (Población2).

La intervención se corresponde con la metodología o procedimientos utilizados en un tema de investigación concreto. En este caso se han seleccionado los términos de búsqueda más usados habitualmente en trabajos relacionados con las pruebas del software.

Los términos relativos a los resultados deberían ser propiedades específicas a probar en las aplicaciones SOA. Esta revisión trata de identificar los trabajos relacionados con las pruebas en aplicaciones SOA con enlace dinámico, independientemente de las características funcionales o no funcionales que están siendo probadas. Por tanto, no se ha seleccionado un conjunto de términos asociados al punto de vista de los resultados. En la Tabla 4 se listan los términos de búsqueda seleccionados atendiendo a los distintos puntos de vista.

IEEExplore
ACM Digital Library
Scopus
EI Compendex
ISI Web of Science

Tabla 3. Bibliotecas digitales

Población1	web services, service oriented, composition, composite web services, compose, SOA, SOAP, WSDL, UDDI, SLA, WSLA, WS-Agreement, OWL-S, BPEL, WS-BPEL, BPEL4WS, BPMN
Población2	discovery, selection, binding, dynamic, linking, self-healing, self-adapting, adaptive, adaptation, interoperability, compatibility, capability, broker, matching, matchmaking, runtime
Intervención	testing, test cases, monitoring, monitor, checking, validation, verification, quality

Tabla 4. Términos de búsqueda de población

Una vez seleccionados los términos, éstos deben ser combinados para obtener una cadena final de búsqueda que será introducida como entrada en los buscadores de las diferentes bibliotecas digitales. En este caso, la combinación de términos se realiza utilizando la fórmula lógica:

Población1 AND Población2 AND Intervención

Dentro de cada punto de vista, los términos se combinan entre sí usando la conjunción lógica OR. Aunque algunos de los términos elegidos son muy genéricos (por ejemplo, selection, runtime, capability, etc.) y podrían generar un número muy elevado de resultados, su aplicación en conjunción con términos mucho más específicos en el ámbito de las pruebas y las aplicaciones SOA provoca que el conjunto de resultados esté muy alineado con la temática de investigación seleccionada.

3.3. Estrategia de selección

Durante la fase de búsqueda es muy posible que muchos de los artículos encontrados no sean relevantes en el ámbito concreto de la investigación. Por tanto, es necesario definir un

conjunto de criterios que han de cumplir todos los artículos para proceder a su posterior revisión. En este caso, se ha decidido excluir de la revisión aquellos artículos que:

C1: No tengan relación con el paradigma de desarrollo basado en SOA.
C2: No ejecuten pruebas del software.
C3: No prueben el enlace dinámico.

La estrategia de selección de artículos está representada en la Figura 2. En primer lugar, se recopilarán los artículos encontrados durante las dos primeras fases de búsqueda. Dado que muchos artículos pueden haber sido encontrados mediante más de una fuente, se eliminarán los duplicados y se almacenará una única copia de cada artículo. Así mismo, se ha decidido seleccionar un solo artículo que siga la misma línea de investigación. En el caso que se hayan ido publicando artículos que complementen o mejoren el trabajo anterior, se consultará a sus autores para seleccionar su trabajo más representativo.

Tras haber eliminado los artículos duplicados, se aplicarán los criterios C1 y C2 a los artículos procedentes de la búsqueda manual. Este filtro se realizará teniendo en cuenta únicamente el título y el abstract de cada artículo. De esta forma se garantiza que todos los artículos que superen estos filtros están abordando enfoques de pruebas en SOA.

La búsqueda se completará con la tercera fase donde se consultarán manualmente las páginas web de los autores de los artículos encontrados previamente, sus listados de referencias y la opinión de los propios autores. Al igual que antes, se eliminarán los artículos que ya hayan sido encontrados en alguna de las fases anteriores.

Al conjunto resultante de artículos procedentes de las tres fases, se le aplicará el criterio C3 teniendo en cuenta el contenido completo del artículo, dando lugar a un conjunto final de estudios primarios. Mediante la aplicación de los sucesivos criterios se garantiza que los estudios primarios están ejecutando pruebas para mejorar el enlace dinámico en aplicaciones SOA.

3.4. Estudio de calidad

La estrategia de selección descrita en el apartado anterior se complementa con la definición de un estudio que asegura la calidad de los estudios primarios. Cada uno de estos artículos será valorado de un modo dicotómico (Sí o No) de acuerdo al siguiente conjunto de criterios, alguno de ellos obtenido de [1][8][14][18].

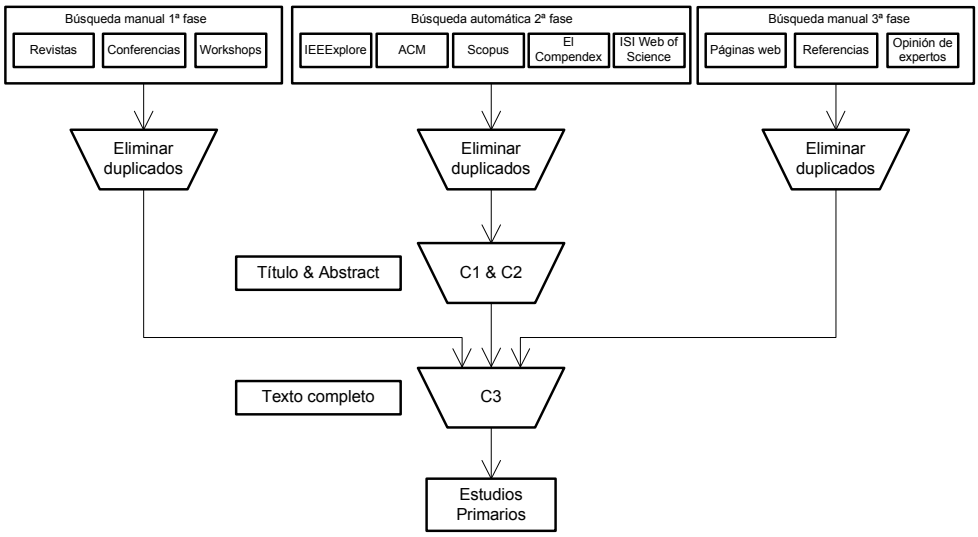


Figura 2. Estrategia de selección de estudios primarios

CA1: ¿Puede el lector entender el objetivo de la investigación?
CA2: ¿Hay una descripción adecuada del contexto en el que se realiza la investigación?
CA3: ¿Hay un apartado dedicado al trabajo relacionado?
CA4: ¿Se describe el método o técnica de pruebas aplicada?
CA5: ¿Ha sido validada la propuesta?
CA6: ¿Las conclusiones están alineadas con el objetivo descrito?
CA7: ¿Se recomienda trabajo futuro?

3.5. Estrategia de extracción de datos

El objetivo de esta fase es diseñar formularios de extracción de datos que permitan obtener la información necesaria para resolver las preguntas de investigación a partir de los artículos seleccionados. El formulario de extracción de datos diseñado contiene un conjunto de campos que proporcionan información estándar del proceso de extracción:

ID identificador único del estudio primario.
Revisor nombre del investigador que ha revisado el estudio.
Fecha fecha de la extracción.

En el formulario también se representa información referente a la publicación revisada:

Título título del estudio.
Autores nombres de los autores del estudio.
Año año de publicación del artículo.
Tipo de publicación fuente de la publicación (revista, conferencia, workshop, capítulo de libro, etc.).
Nombre nombre de la fuente de la publicación.
Referencia información adicional de la publicación (páginas, volumen, número, etc.).

Finalmente, el formulario contiene campos para extraer la información técnica específica de cada artículo:

Objetivo objetivos del estudio.
SUT sistema bajo prueba (servicios web atómicos, composiciones de servicios, etc.).
Tecnologías tecnologías y estándares investigados.
Técnicas técnicas de pruebas aplicadas.
Stakeholder participantes en el proceso de pruebas.
Momento momento en que se ejecutan las pruebas.
Validación tipos de validación aplicados.
Ejemplo ejemplos utilizados para la validación o presentación de la propuesta.
Notas información adicional relevante.

4. Experiencia en la aplicación

El contenido del protocolo ha sufrido modificaciones continuas durante el proceso de revisión. La descripción presentada en este artículo se corresponde con la versión definitiva tras haber desarrollado y finalizado la revisión sistemática.

El proceso completo de revisión se lleva a cabo durante un período temporal más o menos extenso, dependiendo de la amplitud de dicho proceso. Desde que se empezó a planificar la revisión hasta la publicación final de los resultados obtenidos pasó un tiempo en el cual se siguieron publicando artículos relevantes para la revisión. Este hecho hizo que el trabajo estuviese continuamente realimentándose y fuese necesario fijar una fecha límite para la inclusión de nuevos artículos a revisar. Algo parecido sucedió con la identificación de nuevas fuentes de información que no habían sido incluidas en las versiones iniciales del protocolo. Por tanto, uno de los factores que dificultaron esta revisión es que las tareas de búsqueda y selección de artículos (y, en consecuencia, la extracción y síntesis de la información) se extendieron hasta casi el final del proceso de revisión.

Además de la amplitud, tanto temporal como de fuentes de información a revisar, uno de los elementos que más variación sufrió en el protocolo fue la selección de los términos de búsqueda. Al conjunto inicial de términos que se consideraban relevantes en el ámbito de la investigación, se fueron añadiendo nuevos términos como consecuencia del conocimiento

adquirido durante el proceso de revisión. Estas modificaciones obligaron a completar la selección de artículos ampliando el campo de búsqueda inicial.

Tras la tarea de búsqueda y selección de artículos, el estudio de calidad debería haber filtrado aún más el conjunto final de estudios primarios, eliminando aquellos que no cumpliesen los criterios diseñados. Sin embargo, debido al carácter tan específico y reciente del tema de investigación abordado, se decidió no excluir ningún artículo a consecuencia de este estudio y así mitigar el riesgo de obviar algún trabajo relevante para la revisión.

5. Resultados

En esta sección se presenta un resumen de los resultados del proceso de revisión. El contenido completo de dichos resultados y las conclusiones obtenidas tras el estudio puede consultarse en [16].

Tras haber encontrado un conjunto inicial de aproximadamente 400 artículos, los diferentes filtros redujeron el conjunto final de estudios primarios a 33. Aunque no es un número elevado, sí puede ser considerado relevante debido al tema de investigación tan específico abordado. Resulta destacable que ninguno de los artículos seleccionados fue publicado en revistas o congresos especializados en pruebas del software, como por ejemplo, la revista STVR o las conferencias ICST o ISSTA.

La revisión permitió clasificar los artículos atendiendo a un conjunto de taxonomías definidas. Se identificaron dos objetivos a conseguir mediante la ejecución de pruebas: la detección de fallos en el software y la obtención de información para tomar una decisión sobre los servicios web a enlazar. La monitorización de las invocaciones de los servicios es la técnica de pruebas más utilizada, duplicando al número de enfoques que proponían la generación y ejecución de casos de prueba.

En cada uno de los artículos revisados se identificó quién participaba en las pruebas y en qué momento exacto se hacía. Los participantes de las pruebas desempeñan el rol de proveedor de los servicios, cliente, registro o una tercera parte independiente (broker) mientras que las pruebas se ejecutan antes de la publicación de los servicios (t1), mientras éstos están publicados en el registro (t2), justo antes de llevar a cabo el enlazado (t3) y durante la ejecución de los servicios (t4).

Finalmente, se clasificaron los diferentes métodos de validación de las propuestas atendiendo a la taxonomía definida en [17]. La mayoría de los estudios utilizan ejemplos para evaluar sus aportaciones mientras que, por otra parte, un número relevante de artículos no presenta ningún tipo de validación.

6. Conclusiones

En este artículo se ha presentado un protocolo a utilizar en revisiones sistemáticas del estado del arte de la investigación de pruebas del software basado en SOAs con enlace dinámico. El protocolo ha sido diseñado siguiendo las pautas publicadas por Kitchenham en [13].

Se ha formulado un conjunto de preguntas cuyas respuestas permiten identificar aspectos técnicos de los estudios a revisar: objetivos abordados y técnicas de pruebas aplicadas, participantes en el proceso de pruebas en diferentes momentos del ciclo de vida de las aplicaciones, tecnologías y estándares investigados y métodos utilizados para la validación de las propuestas.

El protocolo define la estrategia de búsqueda de los artículos seleccionando tanto las diferentes fuentes de información como los términos de búsqueda adecuados. Dado que la búsqueda proporcionará con seguridad trabajos duplicados o que no aborden exactamente la temática de investigación fijada, se ha elaborado una estrategia de selección de los artículos restringiendo la revisión únicamente a aquellos artículos que satisfacen un conjunto de filtros diseñado. Además, se realiza un estudio de calidad de los artículos valorando cada uno de ellos en función de un conjunto de criterios establecidos. Finalmente, se ha elaborado un formulario de extracción de datos que permite obtener de cada artículo revisado la información necesaria para dar respuesta a las preguntas de investigación planteadas.

La revisión sistemática realizada siguiendo el protocolo diseñado permitió, por una parte, conocer el estado actual de la investigación de las pruebas en el ámbito de las aplicaciones SOA con enlace dinámico, resumiendo en este artículo las

conclusiones obtenidas y, por otra, identificar potenciales líneas de investigación que no han sido aún suficientemente abordadas.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación con el proyecto Test4SOA (TIN2007-67843-C06-01) y por la beca FICYT (Gobierno del Principado de Asturias) BP09-075.

Referencias

[1] Afzal, W., Torkar, R., Feldt, R. A systematic review of search-based testing for non functional system properties, Information and Software Technology 51 (2009), pp. 957-976.

[2] Bertolino, A. Software Testing Research: Achievement, Challenges and Dreams, Workshop on the Future of Software Engineering (FOSE'07), 2007, pp. 85-103.

[3] Bucchiarone, A., Melgratti, H., Severoni, F. Testing service composition, Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07), 2007.

[4] Budgen, D., Turner, M., Brereton, P., Kitchenham, B.A. Using Mapping Studies in Software Engineering, Proceedings of PPIG, Lancaster University, 2008, pp. 195-204.

[5] Canfora, G., Di Penta, M. Testing services and service-centric systems: challenges and opportunities, IT Professional 8, 2, Mar. 2006, pp. 10-17.

[6] Canfora, G., Di Penta, M. Testing and Self Checking, Proceedings of International Workshop on Web Services – Modelling and Testing (WS-MATE'06), 2006, pp. 3-12.

[7] Canfora, G., Di Penta, M. Service-oriented Architectures Testing: A survey, A. De Lucia and F. Ferrucci (Eds.): ISSSE 2006-2008, LNCS 5413, 2009, pp. 18-105.

[8] Diba, T., Dingsoyr, T. Empirical Studies of Agile Software Development: A Systematic Review, Information and Software Technology 50 (2008), pp. 833-859.

[9] Engström, E., Skoglund, M., Runeson, P. Empirical evaluations of regression test selection techniques: a systematic review, Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08). ACM, New York, NY, 2008, pp. 22-31.

[10] Goasduff, L., Forsling, C. Bad Technical Implementations and Lack of Governance Increase Risks of Failure in SOA Projects. Online Press Release (June 2007), http://gartner.com/it/page.jsp?id=508397 (accessible en Mayo de 2010).

[11] ISO/IEC 24765, Software and Systems Engineering Vocabulary, 2006.

[12] Juristo, N., Moreno, A., Vegas, S., Solari, M. In search of what we experimentally know about unit testing, IEEE Software 23 (6), 2006, pp. 72–80.

[13] Kitchenham, B.A. Procedures for Performing Systematic Reviews, Keele University Technical Report TR/SE-0401 and NICTA Technical Report 0400011T.1, 2004.

[14] Kitchenham, B.A., Charters, S. Guidelines for performing systematic literature reviews in software engineering, Technical Report, EBSE-2007-001, UK, July 2007.

[15] Kitchenham, B.A., Budgen, D., Brereton, P. The value of mapping studies – A participant-observer case study, Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE'10), 2010.

[16] Palacios, M., García-Fanjul, J., Tuya, J. Testing in Service Oriented Architectures with Dynamic Binding, Information and Software Technology *(enviado y en proceso de revision)*.

[17] Shaw, M. Writing Good Software Engineering Research Papers, Proceedings of the 25th International Conference on Software Engineering (ICSE'03), 2003, pp. 726-736.

[18] Staples, M., Niazi, M. Experiences Using Systematic Review Guidelines, Journal of Systems and Software 80 (9) (2007), pp. 1425-1437.

[19] Worldwide Services Oriented Architecture (SOA) Infrastructure Market Shares, Strategies, and Forecasts, 2009 to 2015. http://www.the-infoshop.com/report/wg86437-world-soa-infra.html (accesible en Mayo 2010).

Propuesta de optimización en la prueba de mutaciones en Java

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

Lorena Gutiérrez Madroñal
Dpto. de Lenguaje y Sistemas Informaticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
lorena.gutierrez@uca.es

Juan José Domínguez Jiménez
Dpto. de Lenguajes y Sistemas Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
juanjose.dominguez@uca.es

Resumen

Actualmente existen diversas aplicaciones de prueba de mutaciones para programas escritos en Java, la mayoría de ellas ejecutan todos los mutantes generados o emplean técnicas de optimización básicas. En este trabajo se propone emplear la *mutación evolutiva*, una técnica de prueba novedosa que utiliza algoritmos evolutivos para reducir el número de mutantes generados sin pérdida significativa de efectividad, como técnica de optimización de la prueba de mutaciones en programas escritos en Java. En esta propuesta se pretende realizar una nueva herramienta, que empleando esta técnica de optimización basada en algoritmos evolutivos, disminuya su alto coste computacional, que es el principal inconveniente de estas aplicaciones.

1. Introducción

La prueba de mutaciones es una técnica basada en fallos, que mide la efectividad de los conjuntos de los casos de pruebas para la localización de errores, para detectar fallos introducidos de forma intencionada [1, 2]. La efectividad de detección de errores de los conjuntos de los casos de pruebas, está definida como el porcentaje de errores que pueden ser detectados por esos conjuntos de casos de pruebas [3]. La prueba de mutaciones ha sido validada como una poderosa herramienta para la evaluación de la calidad de los casos de prueba [4].

Esta técnica ha sido empleada con éxito en diversos lenguajes, encontrándose herramientas que automatizan la generación de mutantes para diversos lenguajes, tales como Mothra [5] para Fortran, MuJava [6, 7, 8] para Java, GAmera [4] para WS-BPEL, SQLMutation [9] para SQL, etc.

Sin embargo, uno de los problemas a los que se enfrenta la prueba de mutaciones es el elevado número de mutantes que se dispone para un programa original, con el consiguiente coste computacional que implica la ejecución de todos ellos. En este sentido, la mutación evolutiva [10] se presenta como una técnica que permite reducir el número de mutantes generados sin pérdida de efectividad. Esta técnica ha sido implementada en la herramienta GAmera para el lenguaje WS-BPEL.

El objetivo principal de este trabajo es sentar las bases para elaborar una arquitectura de código abierto para la realización de prueba de mutaciones en Java, utilizando *mutación evolutiva* [10] para reducir el número de mutantes generados.

El resto del artículo se organiza de la siguiente forma. En la sección 2 se especifican los conceptos fundamentales asociados con la prueba de mutaciones y el estado actual del tema. En la sección 3 se exponen las hipótesis planteadas en este trabajo. Por último, la sección 4 presenta las conclusiones y el trabajo futuro.

2. Antecedentes y estado actual

En esta sección se presenta, en primer lugar, los conceptos fundamentales relacionados con la prueba de mutaciones. A continuación, se profundiza en una de las herramientas que se ha empleado en el estudio realizado y se contrasta la optimización empleada en dicha herramienta con las utilizadas en el resto de aplicaciones.

Finalmente, se valora dicha optimización para emplearla en nuestro trabajo.

2.1. Prueba de mutaciones

La prueba de mutaciones es una técnica de prueba basada en fallos [1,11], que consiste en introducir fallos simples en el programa original mediante la aplicación de operadores de mutación. Los programas resultantes reciben el nombre de *mutantes*. Cada operador de mutación se corresponde con una categoría de error típico que puede cometer el programador. Si un caso de prueba es capaz de distinguir al programa original del mutante, es decir, la salida del mutante y la del programa original son diferentes, se dice que mata al mutante. Si por el contrario, ningún caso de prueba es capaz de diferenciar al mutante del programa original, es decir, la salida del mutante y del programa original es la misma, se habla de un mutante vivo para el conjunto de casos de prueba empleado.

Aplicar la prueba de mutaciones tiene diversas dificultades, la principal de ellas es la existencia de mutantes equivalentes. Un mutante es equivalente cuando provoca la misma salida que el programa original, por lo que no va a existir ningún caso de prueba que permita matarlo; es decir, ambos tienen el mismo comportamiento. Desafortunadamente, determinar cuál de los mutantes es equivalente al programa original es una actividad extremadamente tediosa y propensa a errores [3].

La calidad de los conjuntos de casos de prueba se mide mediante la puntuación de mutación (*mutation score*), que está definida como el cociente entre el número de mutantes muertos y el número total de mutantes no equivalentes [12].

El número total de mutantes no equivalentes viene de la diferencia entre el número total de mutantes y el número de mutantes equivalentes. La puntuación de mutación es una medida cuantitativa de la calidad del conjunto de casos de pruebas. El conjunto de casos de prueba de mejor calidad es el que obtiene la mayor puntuación de mutación posible, es decir, es el que consigue matar al mayor número de mutantes. Por otro lado, dados dos conjuntos de casos de prueba con la misma puntuación de mutación será preferible el que tenga un menor número de casos de prueba [13].

Gracias a las pruebas de mutaciones se pueden obtener nuevos casos de pruebas que matan a los mutantes que permanecen vivos, de tal modo que la calidad del conjunto inicial de casos de prueba mejora. En 1998, fue Offutt [14] el que propuso por primera vez, la utilización de prueba de mutaciones para la generación de casos de prueba, detallando las condiciones a cumplir por un caso de prueba para matar a un mutante:

1. La sentencia mutada debe ser alcanzada.
2. El estado de ejecución del mutante debe ser diferente al del programa original una vez ejecutada la sentencia mutada.
3. El estado de ejecución tras la sentencia mutada debe propagarse a la salida final del mutante.

En [15] van un paso más allá, ya que no sólo quieren generar casos de prueba para matar mutantes, sino que también intentan minimizar el número de casos de prueba generados para matar al mayor número posible de mutantes.

2.2. Herramientas de prueba de mutaciones en Java

Hoy en día, para el lenguaje de programación Java, hay disponibles diversas herramientas de pruebas de mutaciones. Entre estas aplicaciones se pueden destacar: MuJava [7], Jester [16], Jumble [17], RI [18], JavaMut [19] y Judy [3].

La diferencia entre ellas radica en los operadores de mutación que soportan así como los métodos empleados para la generación de mutantes y las técnicas de aceleración u optimización utilizadas.

La tabla 1 recoge las principales diferencias entre estas herramientas. En ella no aparece JavaMut al no poder disponer del código de la misma. De todas ellas, Judy es la más reciente, y según [3] la más completa de las que existen hasta el momento, según las características anteriormente comentadas.

Características	*MuJava*	*Jester*	*Jumble*	*RI*	*Judy*
Operadores de mutación de selección y tradicionales (ABS, AOR, LCR, ROR, UOI)	Sí	No	No[1]	No[2]	Sí
Operadores de mutación orientados a objetos	Sí	No	No	No	Sí[3]
Nivel de generación de mutantes	Bytecode[4]	Código fuente	Bytecode	--	Código fuente
Produce mutantes no ejecutables	Sí[5]	Sí	Sí[4]	-	Sí
Formato de los mutantes	Ficheros de clases separados	Ficheros fuentes separados	En memoria	--	Agrupados en ficheros fuentes
Soporta JUnit	No	Sí	Sí	Sí	Sí
Soporte para herramienta de construcción o de línea de comando	No	Sí	Sí	--	Sí
Disponible	Sí	Sí	Sí	No	Sí

Tabla 1: Comparativa de las herramientas de prueba de mutaciones en Java [3]

[1]Los operadores de selección no se proporcionan. Éstos deberían ser implementados, y no es algo fácil, éstos se obtienen modificando el código de la herramienta.

[2]El prototipo RI usa sólo unos operadores de mutación simples, por ejemplo, cambiar los tipos primitivos y los objetos de la clase Cadena.

[3]Solo en los niveles entre métodos.

[4]Según Offutt [6], se han implementado tres versiones de MuJava usando diferentes técnicas de implementación: modificación del código fuente, modificación bytecode y esquema de los mutantes. Hubo algunos problemas con las dos últimas técnicas, pero finalmente, se combinaron [7]. Esto significa que algunas de las mutaciones se pueden seguir haciendo con el análisis del código fuente usando el esquema de los mutantes.

[5]Aunque, los mecanismos de generación basados en la transformación de bytecode reducen la posibilidad de la creación de mutantes no ejecutables en comparación con aquéllos basados en la transformación de código fuente, ellos no pueden garantizar que todos los mutantes creados usando esta técnica serán ejecutados (por ejemplo, el cambio del modificador de visibilidad de `public` a `private` (operador de mutación AMC [20]) mediante un método que sea invocado externamente en otros paquetes de proyectos, crearán un mutante no ejecutable, pero esto puede realizarse usando una transformación bytecode).

Todos estos sistemas se caracterizan por estar todos los mutantes posibles de acuerdo a los operadores de mutación empleados en una posible selección previa.

3. Optimizando la prueba de mutaciones en Java

Uno de los principales inconvenientes que presenta la prueba de mutaciones, es el alto coste computacional que supone la ejecución de la gran cantidad de mutantes que se generan para el programa de prueba. En [4] se desarrolla la herramienta GAmera para la aplicación de prueba de mutaciones a composiciones WS-BPEL, que introduce un mecanismo de optimización que permite seleccionar sólo un subconjunto de los mutantes totales que puedan generarse. Para ello GAmera incorpora un algoritmo genético que selecciona sólo los mutantes de mayor calidad, reduciéndose el coste computacional que supondría la ejecución de todos los mutantes.

Actualmente no existe ninguna herramienta de prueba de mutaciones con lenguaje Java que utilice esta técnica de optimización. De ahí nace el estudio que se presenta en este trabajo, el cual quiere reutilizar esta técnica para la generación de mutaciones con un coste computacional menor gracias a la previa selección de aquellos mutantes de mayor calidad.

Esta arquitectura va a constar de tres partes: análisis del programa original, generación de mutantes y ejecución de los mutantes. La figura 1 muestra un esquema de la arquitectura del sistema de prueba de mutaciones en Java que se propone. En ella se recibe como entrada el lenguaje original y un conjunto de casos de prueba.

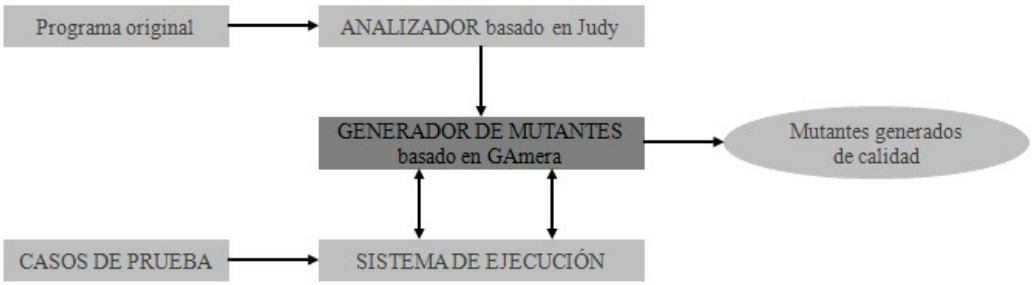


Figura 1: Esquema de la arquitectura del sistema de prueba de mutaciones en Java

El primer componente, el *analizador*, tomaría el programa original y determinaría los operadores de mutación que se pueden aplicar, así como el número de localidades donde puede usarse. Se propone la reutilización de alguna de las arquitecturas de prueba de mutaciones ya existentes para el lenguaje Java, y adaptarla para su integración en GAmera.

Esta información es tomada por el segundo componente, el *generador de mutantes*. Para ello se quiere utilizar el sistema GAmera [4] del grupo de investigación SPI&FM (Software Process Improvement and Formal Methods). En dicho grupo se emplea el lenguaje WS-BPEL y en este estudio, se quiere reutilizar este motor de *generador de mutantes* para el mismo propósito, pero con un lenguaje de programación diferente, Java. En este caso, tendríamos que establecer los mecanismos adecuados para la conexión de ambos componentes. En el siguiente punto se describirá el generador de mutantes del sistema GAmera.

Finalmente, se añadiría un componente de ejecución de mutantes, el sistema de *ejecución*, para que el sistema generador de mutantes pudiese determinar la calidad de los mutantes que va generando. Esto permitiría al sistema generador producir un subconjunto de mutantes adecuados.

El empleo de la mutación evolutiva y su herramienta GAmera a Java se sustenta bajo las siguientes hipótesis:

- Posibilidad de obtener un menor coste computacional: podemos afirmar que el número de mutantes a ejecutar es menor, por lo que, en este caso, se obtendrá un coste menor que si se ejecutasen todos los mutantes, tal y como hacen la mayoría de las herramientas citadas anteriormente.
- Ejecución de mutantes de mayor calidad: Se utilizan algoritmos genéticos para la selección de mutantes de mayor calidad. De este modo a la hora de realizar la ejecución, se ejecutarían los mutantes que hemos obtenido más representativos.

Una vez adaptada la técnica de mutación evolutiva al lenguaje Java podremos realizar comprobaciones con otras herramientas ya existentes. Esto permitiría también contrastar de manera más eficiente esta técnica de optimización. Por tanto, si la adaptación de esta técnica al lenguaje de programación Java es satisfactoria sería una técnica a considerar por el resto de herramientas que emplean otros lenguajes de programación para que el coste computacional no sea el principal inconveniente.

3.1. Generador de mutantes de GAmera

Los mutantes se generan a partir de la información que se recibe del analizador. La herramienta nos da la posibilidad de generar todos los mutantes posibles, o bien, un subconjunto de éstos que va a ser seleccionado por el algoritmo genético. En este último caso, se llamará al componente denominado Generador de mutantes, que está compuesto por dos elementos. El primero, denominado *Búsqueda Genética de Mutantes,* es un algoritmo genético en el que cada individuo representa a un mutante, capaz de generar y seleccionar de forma automática un conjunto de mutantes. Esta selección se realiza aplicando una función de aptitud que mide su calidad en función de si hay o no casos de prueba que lo matan. El segundo elemento es el *Conversor*, que transforma un individuo del

algoritmo genético en un mutante WS-BPEL. Para realizar esta conversión, se utilizan hojas de estilos XSLT, una por cada operador de mutación.

3.2. Operadores de mutación en Java

Son varias las clasificaciones para los diferentes operadores de mutación que se proponen en los distintos estudios realizados para el lenguaje de programación Java. Algunas de las encontradas no son realmente una clasificación, ya que muchos de los estudios se centran en un tipo concreto de operador: concurrentes [21], captura de excepciones [22], tradicionales [3, 23], propios del lenguaje [3, 23], orientados a objetos [3], clases [3, 23], ocultación de información [24], herencia [24], polimorfismo [24] y sobrecarga de métodos [24].

Uno de los primeros pasos que necesitaremos para la realizar es la creación de una clasificación única de operadores de mutación. Esto permitiría disponer de una herramienta que abarcase el conjunto completo de operadores de mutación disponibles para el lenguaje Java.

3.3. Análisis del programa original

Uno de los componentes de la arquitectura a desarrollar será el analizador. Éste se encargará de determinar el conjunto de operadores que se puede aplicar al programa original.

Para ello se quiere reutilizar la aplicación Judy, que consta de un analizador, pero limitado a un subconjunto de los operadores de mutación disponibles.

Nuestro trabajo será, por un lado, ampliar Judy para el reconocimiento de todos los operadores de mutación disponibles, y por otro, realizar la conexión con el motor de generación de GAmera.

Antes de desvelar los motivos que nos han llevado a la selección de Judy para elaborar nuestro trabajo, describimos las diferentes herramientas de pruebas de mutación para el lenguaje Java.

3.3.1. JavaMut

JavaMut [19] no está disponible de manera gratuita para descargar, no soporta la ejecución de mutantes y compila de forma separada cada clase de mutante (lo que es un proceso muy lento). Esto dificulta su elección para el analizador al no disponer del código y poder así reutilizarlo y adaptarlo a nuestra arquitectura.

3.3.2. MuJava

MuJava [6, 7, 8] puede verse como un sucesor de JavaMut que ofrece un gran conjunto de operadores de mutación tradicionales y orientados a objetos para el lenguaje Java. Sin embargo tiene algunas limitaciones:

- No proporciona una interfaz de línea de comando.
- No trabaja con JUnit. Sin embargo el reciente plugin MuClipse actúa como un puente entre el motor de mutación de MuJava y el IDE de Eclipse, parece que resuelve esta limitación. De hecho MuClipse muestra algunas limitaciones referentes a la automatización de la generación de mutantes.
- También hay limitaciones con el proceso de prueba (no es posible seleccionar más de una clase de forma simultánea, y es necesario seleccionar una prueba adicional por cada clase de forma explícita).

No obstante, es la herramienta de prueba de mutaciones más extendidas. El problema es que el proceso de generación de mutantes en bytecode dificultaría su integración con GAmera que emplea código fuente.

3.3.3. Jester

Jester [16] es una herramienta que inicialmente tuvo bastante aceptación en el ámbito de la prueba del software en Java. Sin embargo, Offutt [25] argumentó que Jester resultó ser una forma muy cara para aplicar las pruebas de ramificación, más que las pruebas de mutación, debido a un mecanismo simplificado de la generación de mutantes. Los problemas que conciernen al rendimiento y fiabilidad de la herramienta, así como un rango limitado de posibles operadores de mutación (basados solo en la sustitución de cadenas). Además, los operadores de mutación ofrecidos por Jester no son sensibles al contexto, y a menudo producen mutantes no válidos. Debido a esta gran desventaja, Jester se excluyó como herramienta a reutilizar.

3.3.4. Jumble

Jumble [17] opera directamente en nivel de bytecode para acelerar la prueba de mutaciones. Soporta JUnit y ha sido desarrollado en un entorno industrial, aunque después fue destinado a la investigación. Desafortunadamente, Jumble soporta un conjunto limitado de operadores de mutación, siendo versiones simplificadas de AOR, ASR, COR, LOR, ROR and SOR estudiados en [26]. Por tanto, Jumble tiene un analizador del lenguaje muy simplificado puesto que modela sus propios operadores de mutación, lo que unido a la generación de mutantes bytecode dificulta su integración en GAmera.

3.3.5. Judy

Judy [3] es una implementación del enfoque de FAMTA Light (rápido algoritmo de prueba de mutaciones orientado a aspectos) desarrollado en Java con extensiones AspectJ. Las características del núcleo de Judy es su alto rendimiento del proceso de la prueba de mutaciones así como su fácil integración con herramientas de desarrollo en el ámbito de la prueba del software en Java, lo que facilita la total automatización del proceso de la prueba de mutaciones y soporte para la última versión de Java. Sin embargo, Judy soporta únicamente 16 operadores de mutación predefinidos, por lo que necesitaría extenderse para el soporte del resto de operadores de mutación. No obstante, la disponibilidad de código de la herramienta y su documentación, la convierten en una opción válida.

3.3.6. Response Injection (RI)

RI [18] es un prototipo que se aprovecha de los mecanismos orientados a aspectos para generar mutantes. Se usa solamente para operadores de mutación simples, por ejemplo cambiar los tipos primitivos y los objetos de la clase Cadena. A diferencia de MuJava y Judy, la única mutación aplicada a los objetos es el valor nulo. Además, no está claro cómo RI puede soportar el ancho rango de operadores de mutación diseñados para Java. Otra limitación es el enfoque de RI que no permite cambios en las pruebas en objetos pasados como parámetros de un método. La documentación, así como el código del prototipo no están disponibles. Por ello, RI no nos parece adecuada para su reutilización.

Analizadas las distintas herramientas disponibles, nos decantamos por Judy, por sus características de tratamiento de operadores de mutación, su disponibilidad y documentación.

4. Conclusiones y trabajo futuro

Se ha propuesto emplear la técnica de mutación evolutiva para programas escritos en Java. En ella se pretende obtener un menor número de mutantes a ejecutar, pero siendo éstos los de mayor calidad, de tal forma que el coste computacional no sea el principal inconveniente para este tipo de aplicaciones. Esto se obtiene tras el empleo de algoritmos genéticos para la generación de los mutantes. Si se demuestra que el coste computacional de esta nueva propuesta es menor que la del resto de herramientas que ejecutan todos los mutantes, llegaremos a la conclusión de que la técnica de mutación evolutiva es una de las optimizaciones a tener en cuenta a la hora de realizar estudios de prueba de mutaciones.

Se propone una arquitectura para la optimización de las pruebas de mutaciones en Java. Esta arquitectura reutilizaría el sistema generador de mutantes de GAmera, y se ampliaría con un analizador basado en Judy, así como un sistema de ejecución.

Como trabajo futuro está la implementación de la arquitectura propuesta en este artículo. En primer lugar, habría que ampliar Judy para el reconocimiento de todos los operadores de mutación disponibles, y luego, realizar la conexión con el motor de generación de GAmera.

Referencias

[1] DeMillo, R.A., Lipton, R.J., and Sayward, F.G.: “Hints on Test Data Selection: Help for the Practicing Programmer”, IEEE Computer, 1978, 11, (4), pp. 34 – 41.

[2] Hamlet, R.G.: “Testing Programs with the Aid of a Compiler”, IEEE Trans. Softw. Eng. 1977, 3, (4), pp. 279 – 290.

[3] Madeyski, L. & Radyk, N. 2010, "Judy–A Mutation Testing Tool for Java", IET Softw, vol. 4, no. 1.

[4] Estero Botaro, A., Domínguez Jiménez, J. J., Gutiérrez Madroñal, L.: “GAmera: una herramienta para la generación y selección mediante algoritmos genéticos de mutantes WS-BPEL”, en XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), San Sebastián (2009), págs. 66 – 69.

[5] King, K. N., Jefferson Offutt, A., "A Fortran language system for mutation-based software testing, Software"—Practice & Experience, v.21 n.7, p.685-718, July 1991.

[6] Ma, Y., Offutt, J., Kwon, Y.R.: "MuJava: an automated class mutation system", Softw Test Verif Rel, 2005, 15, (2), pp. 97-133.

[7] Offutt, J., Ma, Y.S., and Kwon, Y.R.: "An Experimental Mutation System for Java", SIGSOFT Softw. Eng. Notes, 2004, 29, (5), pp. 1-4.

[8] Ma, Y.S., Offutt, J., and Kwon, Y.R.: "MuJava: A Mutation System for Java". Proc. 28th Int. Conf. on Software Engineering, Shanghai, China, May 2006, pp.827-830.

[9] Tuya, J., Suarez-Cabal, M. J., & de la Riva, C. (2006). SQLMutation: A tool to generate mutants of SQL database queries. Mutation Analysis, 2006. Second Workshop on, 1-1.

[10] Domínguez Jiménez, J. J., Estero Botaro, A., García Domínguez, A., Medina Bulo, I.: "Mutación evolutiva". En XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Valencia (2010). Aceptado para su publicación.

[11] Offutt, A.J., Untch, R.H.: "Mutation 2000: uniting the orthogonal. Mutation testing for the new century", pág. 34–44 Kluwer Academic Publishers, Norwell, MA, USA (2001).

[12] Zhu, H., Hall, Patrick.A.V., and May, J.H.R.: "Software Unit Test Coverage and Adequacy". ACM Comput. Surv. 1997, 29, (4), pp. 366 – 427.

[13] Ayari, K., Bouktif, S., Antoniol, G.: Automatic mutation test input data generation via ant colony. En: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pág. 1074-1081 (2007).

[14] Offutt, J.: Automatic test data generation. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA (1988).

[15] Estero-Botaro, A., Domínguez-Jiménez, J. J., Medina-Bulo, I., "Una arquitectura para la generación de casos de prueba de composiciones WS-BPEL basada en mutaciones".

[16] Moore, I.: "Jester - a JUnit test tester". Proc. 2nd Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering, Sardinia, Italy, May 2001, pp. 84-87.

[17] Irvine, S. A., Tin, P., Trigg L., Cleary, J. G., Inglis, S., and Utting, M.: "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests". Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques, Windsor, UK, September 2007, pp. 169-175.

[18] Bogacki, B., and Walter, B.: "Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing". Proc. IFIP Work. Conf. on Software Engineering Techniques, Warsaw, Poland, October 2006, pp. 273-282.

[19] Chevalley, P., Thévenod-Fosse, P.: "A mutation analysis tool for Java programs", Int. J. Softw. Tools Tech. Transfer, 2003, 5, (1), pp. 90-103.

[20] Ma, Y.S., Offutt, J.: "Description of Method-level Mutation Operators for Java", November 2005, http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[21] Bradbury, J.S., Cordy, J.R. & Dingel, J. 2006, "Mutation Operators for Concurrent Java (J2SE 5.0)", Mutation Analysis, 2006. Second Workshop on, pp. 11.

[22] Changbing, J., Zhenyu, C., Baowen, X. & Ziyuan, W. 2009, A New Mutation Analysis Method for Testing Java Exception Handling.

[23] Smith, B.H. & Williams, L. 2009, "Should software testers use mutation analysis to augment a test set?", The Journal of Systems & Software, vol. 82, no. 11, pp. 1819-1832.

[24] Yu-Seung, M., Yong-Rae, K. & Offutt, J. 2002, "Inter-class mutation operators for Java", Software Reliability Engineering, 2002. ISSRE 2002. Proceedings. 13th International Symposium on, pp. 352.

[25] Offutt, J.: "An analysis of Jester based on published papers". http://cs.gmu.edu/~offutt/jester-anal.html, accessed September 2006.

[26] Ammann, P., and O_utt, J.: `Introduction to Software Testing' (Cambridge University Press, 2008, 1st edn.).

Un portal web para investigación y docencia en testing combinatorio

Beatriz Pérez Lamancha, Macario Polo Usaola
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4
13071-Ciudad Real

Resumen

En este artículo se presenta un sistema web que se ha desarrollado ex profeso para facilitar la docencia y la investigación en testing combinatorio. El sistema implementa varias estrategias de combinación, todas las cuales se desarrollan a partir de una jerarquía de herencia que facilita la implementación de cualquier algoritmo, que no es sino una especialización de la superclase Algorithm.
La herramienta dispone de una opción Verbose que muestra cada paso en la ejecución de cada algoritmo, así como los resultados intermedios que se van produciendo. Esta característica resulta muy útil para que los alumnos entiendan bien el funcionamiento de cada algoritmo.
La herramienta web se ha utilizado este año en un curso de doctorado en la UCLM: varias parejas de alumnos la han utilizado para evaluar la diferencia de calidad entre test suites construidos con diferentes estrategias, mientras que otras dos han implementado dos nuevos algoritmos: uno genético y otro bacteriológico.

1. Introducción

El testing combinatorio se basa en la combinación de valores de prueba de los parámetros de los servicios del sistema que se va a probar (el *SUT,* el *System Under Test*). Existen muchas técnicas para combinar los valores de prueba, denominadas "Estrategias de combinación", de las cuales Grindal et al. han realizado una excelente recopilación [1].

Realmente, todas las estrategias toman, como entrada, un conjunto de valores de prueba y producen, como salida, un conjunto de casos de prueba (*test suite*). Algunas de estas estrategias tienen como objetivo obtener un test suite de tamaño reducido, si bien esto supone, en muchos casos, que la calidad de sus casos de prueba no sea demasiado alta.

En las siguiente sección se describen algunas estrategias de combinación que se han implementado. En la sección 3 se muestra el diseño de clases del sistema combinatorio. La sección 4 describe otras estrategias de combinación implementadas por algunos alumnos del curso de doctorado *Pruebas y seguridad de sistemas de información*, para lo que han reutilizado el diseño de clases mostrado en la sección previa. Finalmente, se presenta brevemente la aplicación web, invitándose a todos los interesados a utilizarla, y se comentan algunas conclusiones.

2. Algunas estrategias de combinación

Para un conjunto de datos de prueba, la estrategia que más casos de prueba produce es la denominada *All combinations*, de la que se extraen todas las posibles combinaciones de valores en forma de casos de prueba (corresponde, realmente, al producto cartesiano de los conjuntos representados por los parámetros). Muchos de estos casos, sin embargo, son redundantes, en el sentido de que recorren zonas del código que ya son recorridas por otros; no obstante, también es la técnica que obtiene la cobertura más alta en el SUT, pues cualquier caso de prueba obtenido con All combinations estará incluido en el test suite producido por cualquier otra estrategia.

A modo de ejemplo, supongamos que tres parámetros *A, B* y *C* tienen los valores que se muestran en la Tabla (1).

A={1, 2, 3, 4} B={5, 6, 7} C={8, 9}

(1) Tres parámetros con sus valores

El número máximo de combinaciones que puede obtenerse con estos valores es de 4*3*2=24, que pueden enumerarse del 0 al 23

como en la Tabla (2): la primera columna muestra el índice de la combinación; las columnas 2ª, 3ª y 4ª muestran el elemento de los conjuntos A, B y C que forma parte de cada combinación; la columna cuarta muestra las posiciones de los elementos en sus respectivos conjuntos que conforman la combinación.

Como se ha dicho, cualquier caso de prueba obtenido con otra estrategia estará contenido en los que genera el producto cartesiano (estrategia *All combinations*). A continuación, describimos un algoritmo iterativo (no recursivo) para obtener todas las combinaciones y, después, algunas otros algoritmos de combinación.

2.1. Implementación iterativa de *All combinations*

En la Tabla (2) se observa que cada elemento de A aparece tantas veces seguidas como valga el producto de los cardinales de B y C (6 veces, en este caso), que cada elemento de B aparece tantas veces seguidas como el cardinal de C, y que los elementos de C van cambiando en cada combinación. De forma general, si se tienen n conjuntos $\{S_1, S_2, \ldots S_n\}$, cada elemento de S_1 aparecerá $|S_2|\cdot|S_3|\cdot\ldots\cdot|S_n|$ veces seguidas; los elementos de S_2, $|S_3|\cdot|S_4|\ldots\cdot|S_n|$, etc. Los elementos de S_n aparecen una vez cada uno, alternándose en cada combinación.

Puesto que el número de combinaciones se conoce a priori ($\prod_{i=1}^{n}|S_i|$), dado un índice arbitrario i, $0 \le i < \prod_{i=1}^{n}|S_i|$, se puede conocer la composición de la combinación correspondiente a ese índice, por medio del algoritmo *cartesianProduct* que se muestra a continuación. Inicialmente (bucle 1) se calcula el número máximo de combinaciones; a continuación, se añaden al conjunto resultado la combinación correspondiente a ese i.

La función *getCombination*, a la que se llama en el Algoritmo 1, toma como argumentos el conjunto de parámetros (A, B, C, siguiendo con el ejemplo anterior) y el número de combinación que se desea recuperar. Su código, con alguna optimización para evitar la duplicación de cálculos, recibe tres parámetros y corresponde al Algoritmo 2.

i	A [i]	B [i]	C [i]	Posiciones
0	1	5	8	{0, 0, 0}
1	1	5	9	{0, 0, 1}
2	1	6	8	{0, 1, 0}
3	1	6	9	{0, 1, 1}
4	1	7	8	{0, 2, 0}
5	1	7	9	{0, 2, 1}
6	2	5	8	{1, 0, 0}
7	2	5	9	{1, 0, 1}
8	2	6	8	{1, 1, 0}
9	2	6	9	{1, 1, 1}
10	2	7	8	{1, 2, 0}
11	2	7	9	{1, 2, 1}
12	3	5	8	{2, 0, 0}
13	3	5	9	{2, 0, 1}
14	3	6	8	{2, 1, 0}
15	3	6	9	{2, 1, 1}
16	3	7	8	{2, 2, 0}
17	3	7	9	{2, 2, 1}
18	4	5	8	{3, 0, 0}
19	4	5	9	{3, 0, 1}
20	4	6	8	{3, 1, 0}
21	4	6	9	{3, 1, 1}
22	4	7	8	{3, 2, 0}
23	4	7	9	{3, 2, 1}

(2) Enumeración de las 24 combinaciones de los conjuntos A, B y C de la Tabla (1)

```
   function cartesianProduct({S}) :
                    Set(Combinations) {
     numberOfCombinations : int = 1;
     divisors : int[] = new int[|S|]
     for (i=0; i<|S|; i++) {
          numberOfCombinations=
             numberOfCobinations*|Si|
          divisors[i]=1
          for (j=0; j<|S|; j++)
             divisors[i]=divisors[i]*|Sj]
     }
     result : Set(Combinations)=∅
     for   (i=0;   i<numberOfCombinations;
i++)
          result= result ∪
             getCombination(S,          i,
divisors)
     return result
   }
```

Algoritmo 1. Pseudocódigo del algoritmo para obtener todas las combinaciones de varios conjuntos

```
   function getCombination(S, index,
             divisors) : Combination {
     result : Combination = new int[|S|]
     for (int i=0; i<|S|; i++)

  result[i]=(index/divisors[i])%|Si|
     return result
}
```

Algoritmo 2. Recuperación de una combinación abritraria

El tercer parámetro (*divisors*), es un array de enteros que contiene, para el elemento *i-ésimo*, el producto de los cardinales de los conjuntos *i+1* hasta *n*. En el ejemplo de los conjuntos *A, B* y *C* visto anteriormente:

$$divisors_A=|B|\cdot|C|=3\cdot 2=6 \quad divisors_B=|C|=2 \quad divisors_C=1$$

Así, para el caso de los conjuntos A, B y C, la aplicación de la función *getCombination* con el valor 18 en el parámetro *index* devuelve, en el array *result*, los valores {3, 0, 0}, correspondientes a las posiciones de los elementos 4, 5 y 8 de, respectivamente, los tres conjuntos.

El hecho de que el algoritmo sea iterativo y no recursivo (lo cual es relativamente sencillo de conseguir) supone una ventaja clara cuando el número de conjuntos y de elementos es muy grande. Además, la posibilidad de enumerar las combinaciones (casos de prueba) permite almacenar como resultado números enteros, en lugar de combinaciones propiamente dichas: así, por ejemplo, si una determinada estrategia de combinación devuelve los casos de prueba {1, 5, 8}, {2, 7, 9} y {4, 6, 8}, será suficiente con almacenar los valores 0, 11 y 20, lo que también permite ahorros importantes de costo de almacenamiento y de cómputo.

2.2. Implementación de un algoritmo aleatorio

La implementación de un algoritmo que genere casos de prueba aleatorios resulta muy sencilla, pues basta con generar tantos números aleatorios entre 0 y el número máximo de combinaciones menos uno y, a continuación, obtener las combinaciones correspondientes mediante la función *getCombination.*

2.3. Implementación de *Antirandom*

Antirandom es una estrategia de generación de casos propuesta por Malayia [2]: a partir de un caso de prueba inicial que se coloca en el test suite, se añade al conjunto de casos el que sea "más diferente" del primero; a continuación, se añade al test suite el siguiente caso más diferente. Esta "diferencia" se mide en términos de la distancia Hamming y de la distancia cartesiana.

Para el cálculo de la distancia Hamming, cada caso de prueba se codifica como un vector de bits; la distancia de uno a otro se calcula como el número de bits distintos. A igualdad en la distancia Hamming, se toma el caso de prueba cuya distancia cartesiana sea mayor. La distancia cartesiana de un vector binario *v* a un conjunto de vectores binarios es la suma de las raíces cuadradas de las distancias Hamming de *v* a cada uno de los vectores. En la implementación que le hemos dado a este algoritmo aprovechamos que cada caso de prueba se codifica mediante un número entero entre 0 y el número máximo de combinaciones menos uno; la función que calcula la distancia Hamming utiliza operadores de desplazamiento de bits para calcularla (Algoritmo 3).

```
   protected int hD(int a, int b) {
     int d=a^b;
     int count=0;
     while (d>0) {
        if (d%2==1)
           count+=1;
        d=d>>1;
     }
     return count;
}
```

Cálculo de la distancia Hamming de 25 y 40

a=25=011001_2 b=40=101000_2

Antes del bucle:

d=25^40 = 011001_2 ^ 101000_2 = 49 = 110001_2

count=0

Bucle while:

iter. 1: count=1; d = (49>>1) = 11000_2 = 24

iter. 2: count=1; d = (24>>1) = 1100_2 = 12

iter. 3: count=1; d = (12>>1) = 110_2 = 6

iter. 4: count=1; d = (6>>1) = 11_2 = 3

iter. 5: count=2; d = (3>>1) = 1_2 = 1

iter. 6: count=3; d = (1>>1) = 0_2 = 0

Algoritmo 3. Cálculo de la distancia Hamming

El algoritmo procede incluyendo siempre el caso de prueba 0; luego, incluye aquel cuya distancia Hamming sea mayor, y continúa de este modo hasta que se ha obtenido el número de casos deseado.

2.4. Implementación de *Comb*

Una de las principales desventajas de Antirandom es su elevado coste computacional pues, para cada caso de prueba que se va a añadir al test suite, es preciso calcular las distancias de cada candidato al test suite preconstruido.

Inspirados, sin embargo, en la idea de las distancias de Antirandom, hemos implementado *Comb,* un nuevo algoritmo que añade los casos de prueba en función de la distancia de sus correspondientes índices a los índices de los casos de prueba preseleccionados.

De este modo, suponiendo un máximo de 100 casos de prueba, se añaden los casos primero y último (0 y 99), luego el del centro (50), y se continúa con los situados en mitad de los intervalos (0, 50) y (50, 99). En la siguiente iteración, se añaden los elementos situados en el centro de los intervalos correspondientes a la iteración anterior. Así, el algoritmo va completando una especie de "peine" (Figura 1), que es la razón de haber dado este nombre (*comb* en inglés significa peine) a este algoritmo.

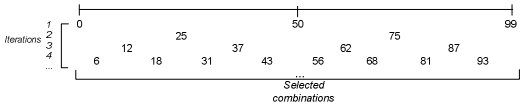


Figura 1. Progreso del algoritmo *Comb*

Puesto que la naturaleza de Antirandom y de Comb es muy parecida, hemos realizado algunos estudios para comprobar la cobertura que alcanzan, medida en términos de *pair wise* (Figura 2), asi como el tiempo de generación de casos de prueba (Figura 3): por lo general, la cobertura de pares es bastante más alta con *Comb*; respecto del tiempo de generación de casos, el tiempo es polinomial en *Comb,* mientras que progresa exponencialmente en *Antirandom*.

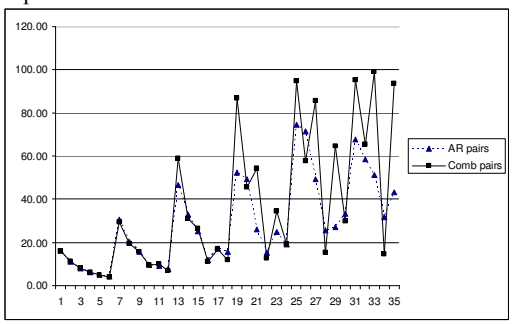


Figura 2. Pares cubiertos por Antirandom y Comb

2.5. Implementación de AETG

AETG es un algoritmo propuesto por Cohen et al. [3] que permite obtener cobertura *n-wise.* La implementación que le hemos dado obtiene cobertura *pair wise*. Los pasos seguidos por nuestro algoritmo se ilustran en el pseudocódigo del Algoritmo 4.

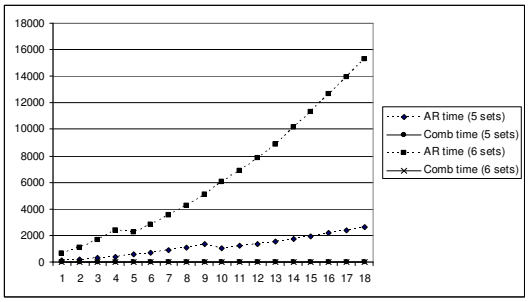


Figura 3. Tiempos de generación de casos en Antirandom y Comb

1. Build *pairTables* for *S*, the set of sets/parameters
2. let *c*=combination #0
3. Add *c* to the *selected* set
4. Update *pairTables* with the pairs visited by *c*
5. while there are unvisited pairs in *pairTables*
   1. initialize *c* putting the value which visits more unvisited pairs in *pairTables*
   2. complete *c* with the values of the remaining sets in such way most pairs are jointly visited
   3. Add *c* to the *selected* set
   4. Update *pairTables* with the pairs visited by *c*

Algoritmo 4. Pseudocódigo de AETG

2.6. Implementación de *Each choice*

Each choice es una estrategia de combinación muy básica [3], en la que cada valor de prueba debe ser incluido en, al menos, un caso de prueba. Esto se consigue generando casos que, en cada iteración, añadan valores de prueba no utilizados.

De este modo, mientras haya elementos (valores de prueba) no visitados, se construye una nueva combinación (caso de prueba): si algún parámetro tiene un valor no utilizado, se coloca en la posición que corresponda en la combinación. El algoritmo para cuando todos los valores han sido utilizados al menos una vez.

3. Diseño del sistema de clases

Como hemos comentado, cualquiera de las estrategias de combinación puede entenderse como un algoritmo que toma como entrada un conjunto de parámetros con valores y produce, como salida, un conjunto de casos de prueba. Así pues, el diseño de clases del sistema (Figura 4) consta de una superclase abstracta *Algorithm*, que dispone de una oepración abstracta *buildCombinations*. El desarrollador de una estrategia de combinación debe, en este contexto, dar la implementación adecuada a esta operación en la subclase correspondiente.

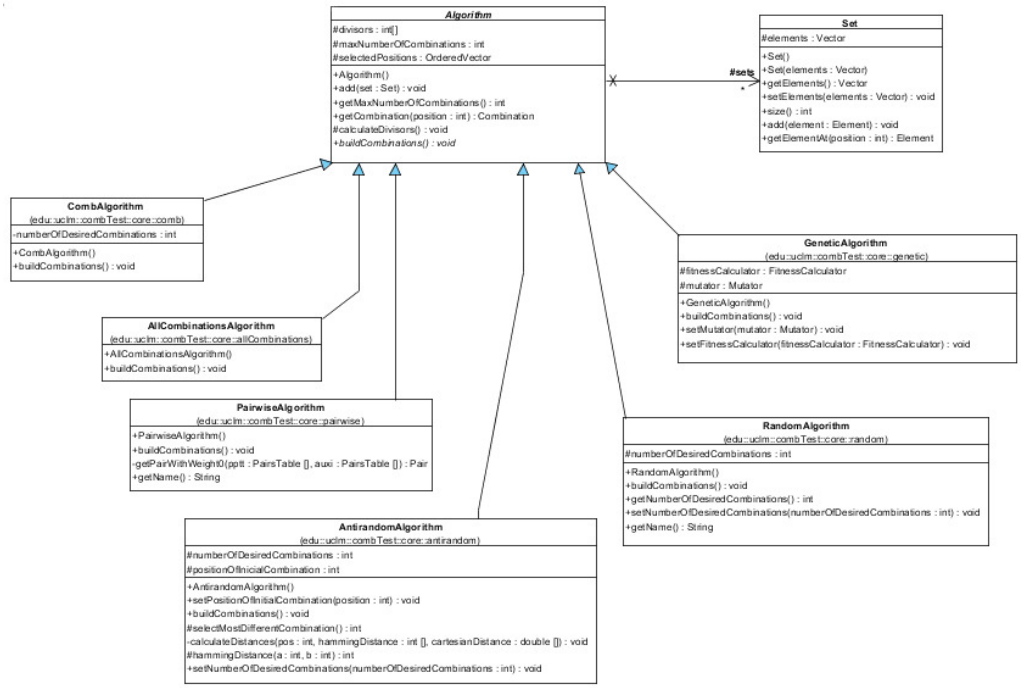


Figura 4. Estructura de clases de las estrategias de combinación

Como se observa en la figura, el algoritmo posee una colección de conjuntos (instancias de *Set*), cada uno de los cuales representa un parámetro. El campo *selectedPositions* es un vector ordenado de números enteros, que representa las posiciones (entre 0 y *n-1,* siendo *n* el número máximo de combinaciones) de los casos de prueba seleccionados.

En la figura se muestran algunas de las especializaciones que han sido implementadas en la aplicación web. Ésta se encuentra disponible en http://alarcosj.esi.uclm.es/CombTestWeb/. En esta URL, el lector interesado encontrará también un *white paper* más detallado, el código fuente en Java de los algoritmos implementados, así como diversas implementaciones que se han hecho también con Visual Basic for Applications en Microsoft Excel.

4. Implementación de otros algoritmos por terceros

En esta sección se describen dos algoritmos implementados por dos parejas de alumnos del curso *Pruebas y seguridad de sistemas de información*, que se imparte en el programa de doctorado de la Universidad de Castilla-La Mancha (y que cuenta con mención de calidad de la ANECA). Para su desarrollo, los alumnos han dispuesto del código fuente de todo el sistema, que se encuentra disponible en la web antes mencionada.

4.1. Implementación de un algoritmo genético

Este algoritmo ha sido implementado por Raúl Miguel Sabariego y Manuel David Jiménez Patiño. Procede como un algoritmo genético habitual: la población inicial y las subsiguientes generaciones están formadas por casos de prueba, que van siendo mejoradas sucesivamente mediante la determinación del *fitness* de los mejores individuos. En la implementación que se ha dado, el *fitness* se calcula en función de la cobertura *pairwise* obtenida por los casos de

prueba. Para ello, los alumnos disponían previamente de una clase abstracta *FitnessCalculator*, que posee la operación abstracta *getFitness(Combination):double*, que ha sido implementada por os estudiantes en la especialización *PairwiseFitnessCalculator.*

Respecto de los cruces de individuos para obtener la población de la siguiente generación, los alumnos han implementado una clase abstracta *Crosser* y una especialización concreta *CrosserImpl*. La mutación (en el sentido de los algoritmos genéticos, no del criterio de cobertura para testing), se consigue también mediante la especialización concreta de un *Mutator* abstracto. Los alumnos han añadido, además, un *PopulationSelector* abstracto que, en su especialización concreta, selecciona la siguiente población de una determinada manera.

El conjunto de clases abstractas incluido en este paquete *genetic* permite la sencilla implementación de nuevos algoritmos de cruce, mutación, cálculo de *fitness* y selección de los individuos que formarán la siguiente población.

En las ejecuciones que se han realizado, el algoritmo converge rápidamente y obtiene, en la inmensa mayoría de los casos, una cobertura *pairwise* del 100%.

4.2. Implementación de un algoritmo bacteriológico

Miguel Ángel Martínez Pinedo y Cleto Martín Angelina han interpretado la descripción del algoritmo genético de Baudry et al. [4, 5], que no es, ciertamente, demasiado detallada como para proceder a su implementación.

El algoritmo es relativamente similar a los genéticos, si bien posee algunas diferencias. Inicialmente, se determina el porcentaje de individuos que compondrán la población inicial, se determina el tamaño de la élite, el porcentaje de exceso de *fitness* requerido para pasar a la siguiente generación y el número máximo de iteraciones.

Este algoritmo reutiliza el calculador de *fitness* descrito en la sección anterior: es decir, evalúa la calidad de cada individuo en función del número de pares de valores de parámetros visitados por el caso de prueba.

Igual que ocurre con el algoritmo genético, este bacteriológico también converge rápidamente y obtiene así mismo una cobertura *pairwise* del 100% en la mayoría de los casos.

5. Descripción de la aplicación web

La Figura 6 muestra la ventana principal de la aplicación web, en donde se ha cargado (mediante el botón *Play example*) un ejemplo por defecto que se incluye con propósitos ilustrativos. Como se observa, se han añadido seis conjuntos de datos (que corresponden a los parámetros, y denotados por las letras *A* a *F*). El ejemplo corresponde a las *features* de una línea de productos de juegos de mesa: *A* representa el tipo de juego; *B* denota si el juego requiere o no dados; *C* indica si el jugador contrario es una persona o un sistema; *D* indica el número de jugadores; *E* representa el medio de pago; y *F* indica si el juego requiere preguntas (caso del Trivial) o no.

En la parte izquierda se listan los algoritmos implementados; el algoritmo que se desee aplicar se seleccionar mediante un *radio button.* Al pulsar el botón *Execute,* los datos se envían al servidor, el algoritmo deseado se pone en marcha y los resultados son retornados al usuario.

El sistema dispone de una opción *Verbose*, que muestra con bastante detalle el progreso del algoritmo en cuestión. Además, se puede generar código para los casos de prueba que se obtienen: obsérvese la caja de texto situada bajo los conjuntos *A* a *F*. Para generar código, el usuario puede utilizar el token *TCNUMBER* para representar el número del caso de prueba, y los tokens *#A, #B, #C,* etc. para hacer referencia al valor del conjunto correspondiente.

La Figura 7 muestra parte de los resultados obtenidos al ejecutar el algoritmo *AETG* con el conjunto de datos de ejemplo y la opción *Verbose* activada. Esta posibilidad resulta de gran utilidad para que los estudiantes aprendan correctamente el funcionamiento de los diferentes algoritmos.

Por último, la Figura 8 muestra los resultados finales (valores de los parámetros de los 13 casos de prueba generados), parte del código de los casos de prueba (obtenido a partir de la plantilla de la Figura 5) y los resultados en cuanto a cobertura *pairwise* obtenidos por estos casos.

```
public void testTCNUMBER {
   Game g=new Game(#A);
   if (#B!=null) g.setDice(true);
   g.setOpponent(#C);
   g.setNumberOfPlayers(#D);
```

```
    g.setPaymentMethod(#E);
    if (#F.equals("Quiz"))
        g.setQuiz(true);
}
```

Figura 5. Plantilla para generar código de los casos de prueba

Agradecimientos

Esta investigación ha sido financiada por los proyectos: PRALIN (PAC08-0121-1374) y MECCA (PII2I09-00758394), Cons. de Ciencia y Tecnol. de la Junta de Comunidades de Castilla-La Mancha. Beatriz Pérez cuenta con una beca FPI de la Junta de Comunidades de Castilla-La Mancha.

7. Referencias

1. Grindal M, Offutt AJ and Andler SF. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(167-199, 2005.
2. Malaiya Y. *Antirandom testing: Getting the most out of black-box testing*. International Symposium on Software Reliability Engineering (ISSRE 95). Toulouse, France, pp. 86-95. IEEE Computer Society Press, Los Alamitos, CA, 1995.
3. Cohen DM, Dalal SR, Parelius J and Patton GC. The combinatorial design approach to automatic test generation. *IEEE Transactions on Software Engineering*, 13(5), 83-89, 1996.
4. Baudry B, Fleurey F, Jézéquel J-M and Traon YL. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(2), 73-96, 2005.
5. Baudry B, Fleurey F, Jézéquel J-M and Traon YL. Automatic test case optimization: a bacteriologic algorithm. *IEEE Software*, 22(2), 76-82, 2005.

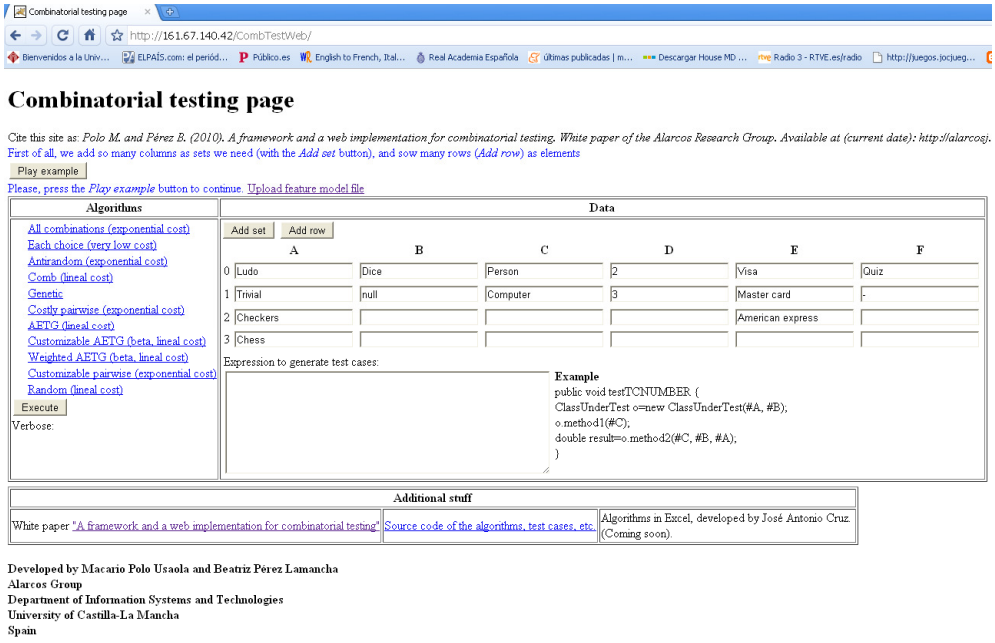


Figura 6. Página principal de la aplicación web

Bienvenidos a la Univ... ELPAIS.com: el period... Público.es English to French, Ital... Real Academia Española últimas publicadas | m... Descarg

Elements	# of visits
(0, 0)	1
(0, 1)	0
(0, 2)	0
(1, 0)	0
(1, 1)	0
(1, 2)	0

4 pairs in (3, 5)	
Elements	# of visits
(0, 0)	1
(0, 1)	0
(1, 0)	0
(1, 1)	0

Elements	# of visits
(0, 0)	1
(0, 1)	0
(1, 0)	0
(1, 1)	0
(2, 0)	0
(2, 1)	0

2. Building new combination
   1. Initial element positions: {-1, 1, -1, -1, -1, -1}
   2. Modified position #0: {0, 1, -1, -1, -1, -1}
   3. Modified position #1: {0, 1, 1, -1, -1, -1}
   4. Modified position #1: {0, 1, 1, 1, -1, -1}
   5. Modified position #1: {0, 1, 1, 1, 1, -1}
   6. Modified position #1: {0, 1, 1, 1, 1, 1}
3. {0, 1, 1, 1, 1, 1} corresponds to combination #45 ({Trivial, Dice, Person, 2, Visa, Quiz}). **{Trivial, Dice, Person, 2, Visa, Quiz} added.**

Visiting pairs:

8 pairs in (0, 1)	
Elements	# of visits
(0, 0)	1
(0, 1)	1
(1, 0)	0
(1, 1)	0

8 pairs in (0, 2)	
Elements	# of visits
(0, 0)	1
(0, 1)	1
(1, 0)	0
(1, 1)	0

8 pairs in (0, 3)	
Elements	# of visits
(0, 0)	1
(0, 1)	1
(1, 0)	0
(1, 1)	0

Figura 7. Resultados parciales con la opción *Verbose*

AETG (lineal cost)
Customizable AETG (beta, lineal cost)
Weighted AETG (beta, lineal cost)
Customizable pairwise (exponential cost)
Random (lineal cost)
Execute
Verbose:

2 Checkers
3 Chess

Expression to generate test cases:

```
public void testTCNUMBER {
Game g=new Game(#A);
if (#B!=null) g.setDice(true);
g.setOpponent(#C);
g.setNumberOfPlayers(#D);
g.setPaymentMethod(#E);
if (#F.equals("Quiz")) g.setQuiz(true);
}
```

Example

```
public void testTCNUMBER {
ClassUnderTest o=new ClassU
o.method1(#C);
double result=o.method2(#C, #
}
```

Algorithm "aetg"

#	Results for 192 combinat	
1	{Trivial,null,Computer,3,Master card,-}	public void test1 { Game g=new Game(Trivial); if (null!=null) g.setDice(true); g.setOpponent(
2	{Trivial,null,Person,3,American express,Quiz}	public void test2 { Game g=new Game(Trivial); if (null!=null) g.setDice(true); g.setOpponent(
3	{Trivial,Dice,Person,2,Visa,Quiz}	public void test3 { Game g=new Game(Trivial); if (Dice!=null) g.setDice(true); g.setOpponent
4	{Chess,null,Computer,2,Visa,-}	public void test4 { Game g=new Game(Chess); if (null!=null) g.setDice(true); g.setOpponent(
5	{Chess,null,Person,3,American express,-}	public void test5 { Game g=new Game(Chess); if (null!=null) g.setDice(true); g.setOpponent(
6	{Chess,Dice,Computer,3,Master card,Quiz}	public void test6 { Game g=new Game(Chess); if (Dice!=null) g.setDice(true); g.setOpponent
7	{Ludo,null,Computer,3,Visa,Quiz}	public void test7 { Game g=new Game(Ludo); if (null!=null) g.setDice(true); g.setOpponent(
8	{Ludo,null,Computer,3,American express,-}	public void test8 { Game g=new Game(Ludo); if (null!=null) g.setDice(true); g.setOpponent(
9	{Ludo,Dice,Person,2,Master card,-}	public void test9 { Game g=new Game(Ludo); if (Dice!=null) g.setDice(true); g.setOpponent(
10	{Checkers,null,Computer,3,Master card,-}	public void test10 { Game g=new Game(Checkers); if (null!=null) g.setDice(true); g.setOppo
11	{Checkers,null,Computer,3,Visa,-}	public void test11 { Game g=new Game(Checkers); if (null!=null) g.setDice(true); g.setOppo
12	{Checkers,null,Computer,3,American express,-}	public void test12 { Game g=new Game(Checkers); if (null!=null) g.setDice(true); g.setOppo }
13	{Checkers,Dice,Person,2,American express,Quiz}	public void test13 { Game g=new Game(Checkers); if (Dice!=null) g.setDice(true); g.setOpp g.setQuiz(true); }

Figura 8. Resultados finales y parte del código generado para los casos de prueba

Oráculo de prueba automatizado para programas de procesamiento XML

Dae S. Kim-Park, Claudio de la Riva, Javier Tuya
Departamento de Informática
Campus de Viesques
Universidad de Oviedo
33204 Gijón
kim_park@lsi.uniovi.es, claudio@uniovi.es, tuya@uniovi.es

Resumen

Se presenta un oráculo de prueba automatizado para dar soporte a la prueba de programas destinados al procesamiento de datos XML. La automatización del oráculo está determinada principalmente por el uso de un lenguaje de especificación ejecutable con el cual se representa la especificación de los programas bajo prueba combinando dos niveles de especificación: (1) una especificación de los requisitos de comportamiento particulares del programa bajo prueba, proporcionada de forma manual de acuerdo a los objetivos de la prueba, y (2) una especificación formando parte del mecanismo de evaluación del oráculo, compuesta por un conjunto de restricciones que describen el comportamiento común a todos los programas de procesamiento XML. Se ilustra la aplicación de oráculo a través de un caso de estudio que muestra resultados satisfactorios.

1. Introducción y motivación

El estándar XML [7] es actualmente el formato de datos universal que predomina en entornos como la Web, en donde dispositivos heterogéneos con diferente hardware y sistemas operativos interaccionan entre sí mediante el intercambio de información.

Habitualmente, los sistemas que forman parte de estos entornos disponen de componentes software especializados dedicados a efectuar las operaciones de acceso y manipulación de datos XML requeridas. Existen múltiples tecnologías con las que es posible implementar estas operaciones, como XPath, XQuery, XSLT, SAX o DOM. Algunas de ellas son adecuadas para operar sobre ficheros con datos XML, otras están adaptadas a la consulta de recursos XML almacenados en repositorios de datos, o son capaces de manipular datos XML en diferentes niveles de detalle. Sin embargo, a pesar de sus diferencias tecnológicas, cualquier conjunto de operaciones de acceso y manipulación de datos XML puede considerarse en términos generales como un *programa de procesamiento XML*, entendiendo como tal cualquier artefacto software de caja negra que toma como entrada un conjunto de datos XML, y produce nuevos datos XML.

Los programas de procesamiento XML pueden implementar operaciones de muy diversa complejidad para procesar entradas XML y calcular y construir sus salidas, por lo que resulta importante verificar y validar su funcionamiento sometiéndolos a pruebas. No obstante, la prueba de este tipo de programas plantea una serie de retos derivados de las características propias de los datos XML que manejan.

Los datos XML presentan las características del modelo de datos semi-estructurado [1]: permiten que un mismo tipo de dato tenga diversas representaciones válidas, pueden definir datos con diferentes tipos de estructuración (desde datos muy estructurados hasta datos completamente desestructurados), y sus datos no necesitan estar restringidos por un esquema. Todo ello dificulta la obtención de casos de prueba para programas de procesamiento XML, tanto en lo que respecta a la obtención de las entradas de las pruebas, como en lo relacionado con la obtención de un *oráculo de prueba* [6], conocido como cualquier principio o mecanismo encargado de comprobar las salidas de la prueba con el objetivo

de emitir un veredicto acerca de la corrección del programa.

En cuanto a la obtención de entradas de prueba, existen varias técnicas y herramientas que pueden emplearse para dar soporte a la generación automática de datos [2][3][5]. Sin embargo, en cuanto a la obtención oráculos de prueba se aprecia una menor actividad investigadora, y el cuerpo de conocimiento en torno a los oráculos de prueba es generalmente escaso. En la actualidad no se dispone de oráculos específicos para programas de procesamiento XML y la evaluación de las salidas de las pruebas se realiza manualmente, lo cual tiende a ser una labor tediosa, propensa a errores y de alto coste debido al volumen y complejidad que pueden alcanzar los datos XML de salida.

Con estas consideraciones, en este trabajo se presenta un oráculo de prueba para dar soporte a la prueba de programas de procesamiento XML. El oráculo presentado basa su procedimiento de evaluación de salidas en dos niveles de especificación, una particular, proporcionada manualmente por el tester, que describe el comportamiento esperado del programa bajo prueba, y otra general integrada en el mecanismo del oráculo de prueba que define el procedimiento para determinar los veredictos de las pruebas. El oráculo ha sido automatizado codificando estos niveles de especificación con un lenguaje de especificación ejecutable.

Este trabajo se estructura de la forma siguiente. En la Sección 2 se describe el oráculo de prueba y su modo de operación dentro del escenario de prueba de los programas de procesamiento XML. En la Sección 3 se detalla la especificación ejecutable aceptada por el oráculo. En la Sección 4 se presenta un caso de estudio con un programa bajo prueba de ejemplo. Finalmente, la Sección 5 presenta las conclusiones y plantea las líneas de trabajo futuro.

2. Descripción del oráculo de prueba

El oráculo opera sobre las salidas de la prueba con dos niveles de especificación: los requisitos de comportamiento y las restricciones del oráculo. Los *requisitos de comportamiento* son atributos que describen el comportamiento de un programa bajo prueba particular. Las *restricciones del oráculo* son condiciones que, dependientes de los

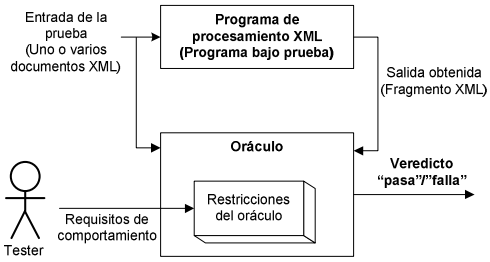


Figura 1. Escenario de prueba de programas de procesamiento XML

requisitos de comportamiento, describen comportamientos generales que todo programa de procesamiento XML correctamente implementado debe cumplir. Son por tanto condiciones necesarias que al ser violadas revelan la presencia de defectos en el programa bajo prueba.

En la **Figura 1** se muestra el escenario de prueba soportado por el oráculo. Durante la prueba, cada programa de procesamiento requiere al menos una especificación de requisitos de comportamiento suministrada manualmente por el *tester*. Por otro lado, las restricciones del oráculo son independientes de los detalles del programa bajo prueba, por lo que se encuentran integradas en el mecanismo de evaluación del oráculo y el tester no necesita especificarlas o conocerlas.

El procedimiento de evaluación de salidas se lleva a cabo a través de un proceso automático en el cual se parametriza cada restricción del oráculo con los requisitos de comportamiento del programa y los datos de ejecución de la prueba, constituidos por la entrada, formada por uno o varios documentos XML [7], y la salida obtenida en forma de fragmento XML—cualquier estructura XML sintácticamente correcta—. A continuación, el oráculo comprueba el cumplimiento de cada una de las restricciones. Si alguna de ellas no se satisface, el oráculo emite un veredicto de fallo (veredicto "falla") indicando el motivo de la violación. En caso contrario, si todas las restricciones se satisfacen, el oráculo indica que el programa ha pasado la prueba (veredicto "pasa"), en cuya situación el oráculo no revela la presencia de defectos en el programa.

3. Especificación ejecutable del oráculo

Para automatizar el procedimiento de evaluación de salidas, los requisitos de comportamiento y las restricciones del oráculo se han codificado manualmente mediante el lenguaje XQuery [10]. Este lenguaje está especialmente concebido para implementar operaciones de acceso y manipulación de datos XML, y presenta una sintaxis más simplificada y de más alto nivel que otras alternativas existentes orientadas a operar sobre datos XML, como podrían ser las interfaces de programación DOM o SAX. Con ello, el lenguaje XQuery se ha empleado como lenguaje de especificación ejecutable, cumpliendo la doble función de permitir la especificación de operaciones sobre datos XML con un alto nivel de abstracción, y de ser ejecutado sobre instancias de datos particulares para producir ciertos resultados útiles para el oráculo.

A continuación se describen los requisitos de comportamiento y las restricciones aceptadas por el oráculo. En un trabajo previo [4] se puede encontrar una definición formal de estos elementos de especificación.

3.1. Requisitos de comportamiento

Los requisitos de comportamiento pueden ser suministrados con el nivel de detalle que sea adecuado de acuerdo a los objetivos de la prueba, pudiendo incluso omitirse algunos de ellos si no se consideran necesarios. En cualquier caso el oráculo podrá emitir veredictos de fallo fiables, aunque la detección de los fallos será más imprecisa cuanto menor detalle tengan los requisitos de comportamiento. De aquí en adelante en esta sección, se definen los requisitos de comportamiento disponibles para especificar el comportamiento del programa bajo prueba.

Implementación relajada del programa (`gs`). Es un programa de procesamiento de XML implementado en el lenguaje XQuery que, recibiendo cualquier entrada de prueba, produce un superconjunto de la salida esperada. El programa así implementado contendrá menos operaciones de filtrado de datos que el programa bajo prueba, por lo que se espera que su implementación sea más sencilla y menos propensa a errores que la del programa bajo prueba. La salida producida por esta implementación se considera una aproximación de la salida esperada.

Conjunto de nodos esperados (`$Es`). Es una variable de XQuery que enumera una secuencia de nombres y tipos de nodos XML (elementos, atributos o nodos de texto [9]) que han de estar incluidos en la salida de todas las ejecuciones del programa bajo prueba.

Conjunto de nodos inesperados (`$Us`). Similar al conjunto de nodos esperados, permite enumerar los nodos que no han de estar presentes en ninguna salida del programa bajo prueba.

Conjunto de ordenación esperada (`$Rs`). Es una variable de XQuery que especifica la ordenación esperada de los nodos de la salida (ordenación lexicográfica del valor de los nodos, ascendente o descendente).

Conjunto de cardinalidad esperada (`$Cs`). Es una variable de XQuery que permite especificar el número de nodos hijo que debe haber bajo ciertos nodos de la salida.

3.2. Restricciones del oráculo

Las restricciones del oráculo son condiciones necesarias para la corrección del programa bajo prueba, destinadas a comprobar si los datos de ejecución de las pruebas satisfacen los requisitos de comportamiento. Estas restricciones están definidas formando parte del oráculo, y no necesitan ser suministradas ni modificadas por el tester para efectuar las pruebas.

Cada una de las restricciones ha sido implementada como una función XQuery que comprueba el cumplimiento de la condición establecida por la restricción. Cada función se encuentra parametrizada con los datos de la prueba y los requisitos de comportamiento y, en caso de que la restricción sea violada, produce un mensaje que describe el fallo desencadenante de la violación.

A continuación se describen las restricciones del oráculo indicando la condición que deben satisfacer para que el oráculo no emita el veredicto de fallo.

Restricción 1: Inclusión en la salida relajada: Los nodos producidos por el programa bajo prueba deben estar contenidos en la salida producida por la implementación relajada (`gs`) pero no en el conjunto de nodos inesperados (`$Us`). Como ejemplo, en la **Figura 2** se muestra la implementación de esta restricción. Como puede observarse, si la restricción es violada, se produce

un mensaje de error representado por un nodo

```
declare function constraints:constraint1(
      $gs_I as item()*,
      $Us as item()*,
      $p_I as item()*)
as item()*
{
  let $gsI_Us := nodes:delete-from-nodeset($gs_I, $Us)
  for $n in nodes:distinct-nodes($p_I)
  return
    if(exists(nodes:equivalent-nodes($gsI_Us, $n))
    then
      ()
    else
      <failure constraint="1"
       message="Unexpected node found in the test
                output: {nodes:print($n)}"/>
};
```

Figura 2. Implementación de la restricción 1.

`failure`. El resto de restricciones sigue un patrón de implementación análogo.

Restricción 2: Inclusión de nodos raíz en la salida relajada: Los nodos raíz de la salida de la prueba deben estar contenidos entre los nodos raíz de la salida producida por la implementación relajada (`gs`), pero no entre los nodos inesperados (`$Us`).

Restricción 3: Consistencia de la jerarquía: Para todo par de nodos que cumplan la relación padre-hijo en la salida esperada, debe existir en la salida de la implementación relajada otro par de nodos equivalente que cumpla la misma relación.

Restricción 4: Presencia de nodos esperados: La salida del programa bajo prueba contiene los nodos esperados (`$Es`).

Restricción 5: Ordenación correcta: Los nodos de la salida obtenida se encuentran correctamente ordenados de acuerdo al conjunto de ordenación esperada (`$Rs`).

Restricción 6: Rangos de cardinalidad esperados: Los nodos de la salida obtenida cumplen los requisitos de cardinalidad impuestos por el conjunto de cardinalidad esperada (`$Cs`).

4. Caso de estudio

En esta sección se ilustra la aplicabilidad del oráculo de prueba a través de un caso de estudio experimental que consta de los siguientes pasos:

1. Inicialmente, se selecciona un programa de procesamiento XML objetivo (Sección 4.1).
2. Se suministra al oráculo un conjunto de requisitos de comportamiento que especifican el comportamiento esperado del programa seleccionado en el paso 1 (Sección 4.2).
3. Se toma un caso de prueba de referencia para el programa seleccionado que incluye una entrada y una salida esperada. Aplicando la técnica de perturbación de datos [11] sobre la salida esperada, se obtienen múltiples instancias de salidas con fallos, simulando resultados incorrectos de la ejecución del programa (Sección 4.3).
4. Se ejecuta el oráculo con los requisitos de comportamiento suministrados en el punto 2 y los datos de prueba con fallos simulados obtenidos en el paso 3, y se recogen los resultados (Sección 4.4).

Tras completar estos pasos, se ha medido la efectividad del oráculo en función del número de salidas con fallos simulados que el oráculo ha juzgado con un veredicto de fallo. Este método es adecuado en tanto en cuanto el mecanismo de comprobación del oráculo se basa en restricciones que han sido definidas en forma de condiciones necesarias para la corrección del programa. Como consecuencia de ello, cada vez que el oráculo emite un veredicto de fallo—y por tanto se ha violado alguna restricción—se puede afirmar que dicho veredicto es fiable y está determinado por las capacidades del oráculo. En cambio, no sería factible evaluar la efectividad del oráculo según los veredictos positivos (veredicto "pasa") que emita ante ejecuciones correctas, ya que si no se produce la violación de ninguna restricción, el veredicto no es concluyente y no indica ninguna medida de efectividad fiable.

4.1. Selección del programa objetivo

Se ha seleccionado el programa mostrado en la **Figura 3**. Este programa ha sido extraído del Caso de Uso XMP del W3C [8] (identificado con el nombre Q12). Nótese que aunque el programa está implementado en el lenguaje XQuery, desde el punto de vista del oráculo podría estar implementado con cualquier otro lenguaje siempre y cuando con el enfoque de caja negra presente el comportamiento de un proceso con entradas y

salidas XML acordes al escenario de prueba de la Sección 2.

```
<bib>
{
    for $book1 in doc("bib.xml")//book,
        $book2 in doc("bib.xml")//book
    let $aut1 := for $a in $book1/author
                 order by $a/last, $a/first
                 return $a
    let $aut2 := for $a in $book2/author
                 order by $a/last, $a/first
                 return $a
    where $book1 << $book2
    and not($book1/title = $book2/title)
    and deep-equal($aut1, $aut2)
    return
        <book-pair>
            { $book1/title }
            { $book2/title }
        </book-pair>
}
</bib>
```

Figura 3. Programa bajo prueba de ejemplo

El programa seleccionado tiene como cometido acceder a un fichero (*bib.xml*) que almacena información bibliográfica en formato XML, y procesar su contenido para producir los *pares de títulos diferentes de los libros que tengan los mismos autores*.

4.2. Especificación de requisitos de comportamiento

Se ha especificando el comportamiento esperado del programa suministrando los requisitos de comportamiento mostrados en la **Figura 4**. Este conjunto de requisitos incluye:

- Una implementación relajada (función XQuery **`gs`**) obtenida relajando la condición sobre los autores de los libros; es decir, la implementación relajada devuelve todos los posibles pares de títulos diferentes. Esta implementación es más sencilla y menos propensa a errores que el programa bajo prueba de la **Figura 3**, aunque en lugar de producir la salida esperada, produce un superconjunto de la misma (contiene la salida esperada).
- Un conjunto de cardinalidad esperada (variable XQuery **`$Cs`**) indicando que cada par de libros devuelto está formado por dos elementos—cardinalidad en el rango [2, 2]—correspondientes al título de cada libro.

El resto de requisitos de comportamiento no ha sido definido en este caso particular. La

```
declare function gs($input as item()*) as item()*
{
    <bib> {
        for $t1 in $input//title
        for $t2 in $input//title
        where $t1 ne $t2
        return
            element book-pair {$t1, $t2}
    }</bib>
};


declare variable $Cs :=
   <cardinalities>
      <cardinality nodes="/bib/book-pair"
                   contents="/*"
                   minInclusive="2"
                   maxInclusive="2"/>
   </cardinalities>
```

Figura 4. Requisitos de comportamiento para el programa bajo prueba

intención es mostrar que es posible suministrar requisitos de comportamiento con bajo coste de especificación, y aunque ello pueda incurrir en cierta pérdida de precisión del oráculo, la efectividad final puede ser razonablemente alta.

4.3. Generación de salidas con fallos

Las salidas con fallos se han obtenido mediante la técnica de perturbación de datos [11]. Esta técnica consiste en aplicar operadores de perturbación (alteración) sobre determinados datos de referencia para generar nuevos datos útiles para la prueba. En este caso de estudio, los operadores de perturbación han sido aplicados sobre una salida esperada (correcta) de un caso de prueba, dando como resultado diversas instancias de salidas con fallos. Concretamente, como caso de prueba de referencia se ha empleado el par entrada/salida del programa Q12 proporcionado en el Caso de Uso XMP [8].

Los operadores de perturbación empleados (detallados con mayor profundidad en [4]) han sido: (1) la eliminación de nodos XML individuales y la eliminación de subárboles XML (nodos XML y sus descendientes) de la salida, (2) la permutación de nodos y subárboles—intercambiando nodos y subárboles entre sí dentro de la jerarquía XML—, y (3) la duplicación de subárboles XML.

Se han obtenido en total 122 ejemplares de salidas con fallos, cada una de las cuales ha

resultado de aplicar uno de los operadores sobre un nodo o subárbol de la salida esperada de referencia.

4.4. Resultados obtenidos

Los resultados obtenidos tras ejecutar el oráculo con los requisitos de comportamiento suministrados (Sección 4.2) y los datos de prueba simulados (Sección 4.3), se muestran en la Tabla 1. La primera columna de la tabla indica el operador de perturbación que se ha empleado sobre la salida esperada de referencia. La segunda columna representa el número de instancias de salidas con fallos que se han derivado al aplicar el operador de perturbación. La tercera columna indica el número de salidas que el oráculo ha juzgado correctamente con un veredicto de fallo. Por último, la cuarta columna indica el porcentaje de salidas juzgadas correctamente, el cual da una idea de la efectividad del oráculo ante cada tipo de perturbación.

Tabla 1. Resultados de la ejecución del oráculo sobre las salidas con fallos

Operador	Nº salidas	Nº salidas con fallo detectado	Efectividad (%)
Eliminación de nodos	6	4	66,67
Eliminación de subárboles	5	2	40,00
Permutación de nodos	71	71	100,00
Permutación de subárboles	34	34	100,00
Duplicación de subárboles	6	4	66,67

4.5. Discusión de los resultados

Los resultados muestran que aun sin disponer de requisitos de comportamiento muy precisos, el número de veredictos correctos producidos por el oráculo ha sido alto.

El oráculo muestra una efectividad máxima en la detección de fallos producidos por permutaciones inesperadas de nodos y subárboles, las cuales han sido detectadas por las restricciones del oráculo 2, 3 y 6 (ver Sección 3.2).

Las perturbaciones por eliminación de nodos han sido detectadas satisfactoriamente por las restricciones del oráculo 3 o 6, debido a que la eliminación de nodos produce inconsistencias de la jerarquía padre-hijo de los nodos. Existen mayores dificultades para detectar los fallos debidos a la eliminación de subárboles, ya que para detectar este tipo de perturbaciones, el oráculo sólo dispone de los requisitos de cardinalidad soportados por la restricción del oráculo 6.

Por último, las duplicaciones de subárboles son detectadas, o bien con la restricción del oráculo 6, o bien con la restricción 1 cuando la duplicación se produce sobre nodos de texto—la duplicación de subárboles aplicada sobre un nodo de texto equivale a concatenarle a dicho nodo su propio su valor, convirtiéndolo en un nodo de texto espurio que produce la violación de la restricción 1.

En principio, el número de salidas incorrectas empleadas en este caso de estudio no es muy amplio para validar la efectividad del oráculo. No obstante, en [4] se ha hecho una evaluación más exhaustiva con un mayor número de programas, y los resultados han sido análogos.

5. Conclusiones y trabajo futuro

Se ha presentado un oráculo automatizado para dar soporte a la prueba de programas de procesamiento XML. El oráculo opera con requisitos de comportamiento del programa bajo prueba suministrados a mano, y restricciones que definen el procedimiento de evaluación de las salidas de las pruebas. Los requisitos de comportamiento y las restricciones del oráculo han sido implementados empleando un lenguaje de especificación ejecutable facilitando así la automatización del oráculo.

Se ha evaluado la aplicabilidad del oráculo a través de un caso de estudio en donde se muestra que es capaz de detectar un número de fallos razonable, aun cuando los requisitos de comportamiento del programa bajo prueba se suministran con baja precisión y, por tanto, bajo coste.

Para mejorar las capacidades de detección de fallos del oráculo, se plantea como trabajo futuro identificar y estudiar requisitos de comportamiento y restricciones del oráculo

adicionales, tratando de mantener un bajo coste de especificación manual de cara a la aplicación del oráculo. También se considerará la posibilidad de mejorar la especificación del oráculo complementándola con información derivada del criterio de selección que sea empleado durante las pruebas.

6. Agradecimientos

Este trabajo ha sido financiado por el Gobierno del Principado de Asturias con la beca PCTI-FICYT (Ref. BP09080, y ha sido parcialmente financiado por el Ministerio de Educación y Ciencia de España dentro del Plan Nacional I+D+i, a través del proyecto Test4SOA (TIN2007-67843-C06-01).

Referencias

[1] Abiteboul, S. "Querying Semi-Structured Data." International Conference on Database Theory. 1997. 1-18.

[2] Barbosa, D., A. Mendelzon, J. Keenleyside, and K. Lyons. "ToXgene: a template-based data generator for XML." International Conference on Management of Data. 2002. 616-616.

[3] Bertolino, A. "TAXI—A tool for XML-Based Testing." International Conference on Software Engineering, 2007. 53-54.

[4] Dae S. Kim-Park, Claudio de la Riva, Javier Tuya. "An Automated Test Oracle for XML Processing Programs." International Workshop on Software Test Output Validation, 2010.

[5] de la Riva, C., J. García-Fanjul, and J. Tuya. "A Partition-Based Approach for XPath Testing." International Conference on Software Engineering Advances. 2006. 17--22.

[6] Weyuker, E. J. "On Testing Non-testable Programs." The Computer Journal, 25 (4), 1982: 465-470.

[7] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition). 28 de Noviembre de 2008. http://www.w3.org/TR/xml/ (último acceso: 2010).

[8] World Wide Web Consortium. XML Query Use Cases. 23 de Marzo de 2007. http://www.w3.org/TR/xquery-use-cases/ (último acceso: 2010).

[9] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model (XDM). Enero 23, 2007. http://www.w3.org/TR/xpath-datamodel/ (último acceso: 2010).

[10] World Wide Web Consortium. XQuery 1.0: An XML Query Language. 23 de Enero de 2007. http://www.w3.org/TR/xquery/ (último acceso: 2010).

[11] Xu, W., J. Offut, and J. Luo. "Testing Web Services by XML Perturbation." International Symposium on Software Reliability Engineering. 2005. 257-266.

Verificación de Datos en la GUI como un Aspecto Separado de las Aplicaciones

Pedro Luis Mateo Navarro
Departamento de Ingeniería de la Información y las Comunicaciones
Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30.071
Murcia, España
pedromateo@um.es

Gregorio Martínez Pérez
Departamento de Ingeniería de la Información y las Comunicaciones
Universidad de Murcia, 30.071
Murcia, España
gregorio@um.es

Diego Sevilla Ruiz
Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30.071
Murcia, España
dsevilla@um.es

Resumen

La verificación de datos es un elemento esencial y crítico dentro de un software. Con este fin, en este trabajo se propone un framework que describe una capa de verificación situada entre la interfaz gráfica (GUI) y la lógica de aplicación. Esta capa analiza dinámicamente los eventos GUI generados por la interacción del usuario para detectar cambios, ante los cuales lanza un proceso de comprobación de reglas que representan los requisitos asociados al elemento interactuado. Además, la capa puede realizar intervenciones sobre la GUI, consiguiendo así un proceso de verificación más rico e interactivo. Esta solución trata la verificación de datos como un aspecto independiente, donde la lógica del proceso es transparente a los desarrolladores, y las reglas, escritas en lenguaje interpretado, se incluyen en ficheros separados.

Palabras clave: *Verificación Entrada/Salida, Verificación GUI, Verificación en Tiempo Real, Software Orientado a Aspectos.*

1. Introducción

Según el estándar IEEE 1012-2004 [23], la *Verificación y Validación Software (V&V)* tiene como principal fin determinar, durante el ciclo de vida del software, si los requisitos del propio software y del sistema son correctos, completos, precisos y consistentes, y asegurar así la calidad de un producto software. Los procesos de *Validación y Verificación de la Entrada* surgieron para asegurar que sólo las entradas válidas sean aceptadas, y las que no, sean rechazadas [14]. Un caso especial es la *Verificación Runtime (RV)*, donde la especificación de requisitos se comprueba en tiempo de ejecución sobre la propia ejecución de la aplicación [5], en vez de sobre un modelo, un diseño, o un trozo de código como hacen las técnicas estáticas [4].

La solución propuesta también está basada en el *Desarrollo de Software Orientado a Aspectos (AOSD)*, ya que el proceso de verificación queda aislado de la lógica principal, consiguiendo así una mayor modularidad, encapsulación y especialización.

Este trabajo se centra en el procesamiento dinámico de la entrada de datos a través de la interfaz gráfica (GUI). En las GUIs, tanto las entradas proporcionadas por los usuarios como las generadas automáticamente son propensas a errores [10]. Por lo tanto, con el fin de evitar un mal funcionamiento del software, resulta imprescindible que los sistemas sean tolerantes a los datos, y puedan responder adecuadamente ante valores esperados e inesperados. Esto nos lleva a proponer este framework, el cual describe una capa de verificación situada entre la GUI y la lógica de aplicación. Esta capa de verificación analiza los eventos GUI para monitorizar los cambios producidos. Ante un cambio, se buscan todas las reglas del SRS (Especificación de Requisitos Software) relacionadas con el campo modificado, y se comprueban sobre el estado actual de la GUI.

El resto del artículo se estructura como sigue. El trabajo relacionado es comentado en la Sección 2. En la sección 3 se comenta el enfoque de verificación propuesto, y en la Sección 4 se incluyen algunas discusiones. Finalmente, en la Sección 5 se añaden unas conclusiones y se comentan las líneas de trabajo futuro.

2. Estado del arte

En esta sección se analizan una serie de soluciones que, al igual que la propuesta en este trabajo, se centran en mejorar la calidad del software mediante el uso de procesos RV. El primer ejemplo es *RAVEN* [9], un framework para la validación de propiedades de accesibilidad en GUIs Java. Se trata de un inspector de propiedades que valida mientras la aplicación está siendo usada. *RAVEN* posee un conjunto de reglas asociadas a cada tipo Java definidas en un lenguaje propio basado en XML.

Otras soluciones utilizan lógicas formales para validar propiedades. Barringer et al. describen en [2] un framework general para monitorizado de propiedades sin estado (no almacena la traza de ejecución) basado en la lógica *EAGLE*. *MaCS* [12] es otra herramienta cuyo objetivo es corregir el comportamiento del sistema en tiempo de ejecución. Utiliza un filtro de eventos (cambios en la aplicación); cada uno de los eventos está ligado a una serie de propiedades que deben ser validadas. Las validaciones pueden implicar acciones para corregir el comportamiento de la aplicación. En [26] Zee et al. describen una solución que combina validación estática y en tiempo de ejecución basada en *Jahob*. *Jahob* utiliza fórmulas de alto orden lógico (HOL) que son comprobadas constantemente en tiempo de ejecución como si de un invariante se tratara. *RuleR* y *LogScope* son otros dos sistemas basados en reglas propuestos por Barringuer et al. en [3]. El primero utiliza reglas single-shot (escritas en lógicas temporales) que operan sobre una colección de estados. El segundo es una derivación de *RuleR* en la que la especificación está basada en patrones de alto nivel que internamente son traducidos a un autómata.

En [5] Chen y Rosu presentan *MOP*, un framework que genera monitores encargados de comprobar ciertas restricciones sobre las propiedades seleccionadas. Éstos son posteriormente integrados en el código de la aplicación. Para realizar las comprobaciones, *MOP* utiliza instantáneas del estado de la aplicación extraídas en tiempo de ejecución, llamadas "eventos". En dicho trabajo hacen referencia a soluciones como *Tracematches* [1] (Alian et al.) y *J-LO* [4] (Bodden), las cuales usan pointcuts de AspectJ [6] para disparar la ejecución de pequeñas piezas de código que validan propiedades. Por último, Falcone et al. describen en [8] unos monitores especiales para modificar propiedades en tiempo de ejecución, llamados *Enforcement Monitors*. En caso de que la comprobación de ciertas reglas no tenga éxito, dichos monitores corrigen el error para que el flujo de ejecución no se desvíe del comportamiento deseado. Estos monitores están basados en el *Autómata de Seguridad* (Schneider [25]) y el *Autómata de Edición* (Ligatti et al. [13]).

En trabajos anteriores se propuso el *Open HMI Tester* [15,20], un framework abierto y adaptable para el desarrollo de herramientas de pruebas. Este framework es capaz de extraer la información incluida en los eventos GUI generados por la interacción del usuario utilizando introspección no intrusiva. Como resultado se presentó un prototipo de código abierto para el sistema de ventanas Qt [24], que fue implantado con éxito en escenarios industriales. En [16,21,20] se presentó una solución para generar automáticamente casos de prueba sobre GUIs. Para ello se utilizaban anotaciones que incluían, para cada elemento relevante de la GUI, los valores soportados y ciertas reglas de validación asociadas a esos valores.

3. Proceso de Verificación de Datos Orientado a Aspectos

Las GUIs se usan básicamente para la introducción de datos por parte del usuario y para que el sistema muestre los resultados del procesamiento. En ambos casos los datos deberían ser verificados para evitar efectos no deseados, y también para comprobar su coherencia y consistencia.

Tradicionalmente las comprobaciones suelen realizarse en la lógica de aplicación, y más raramente en el propio código de los widgets. En la primera, las reglas se incluyen en el código de los métodos manejadores, lo que empeora la legibilidad del código, y por tanto dificulta los procesos de depuración y mantenimiento. En la otra, las reglas están en la propia especificación de la GUI, haciendo que el código esté más disperso. Además, la gestión de un flujo de log único e independiente para el proceso de verificación suele ser compleja en ambos enfoques.

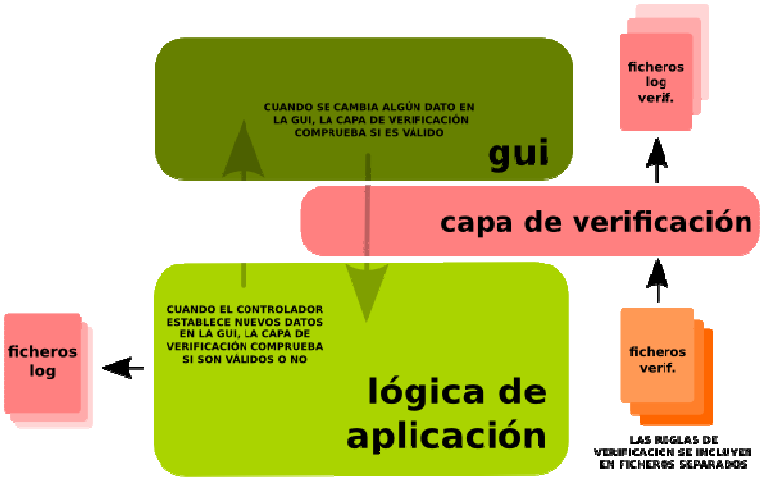


Figura 1. Vista general de la Capa de Verificación.

3.1. Propuesta de verificación de Datos como un Aspecto Separado del Software

El enfoque propuesto en este artículo describe un framework abierto y adaptable que despliega una capa de verificación entre la GUI y la lógica de aplicación (Figura 1). Esta capa representa un elemento independiente dentro del software capaz de analizar, dinámicamente y de forma transparente, el flujo de eventos GUI generado por la interacción del usuario y los cambios realizados por la lógica de aplicación. Como se aprecia en la Figura 2, ese análisis permite detectar los cambios realizados en la GUI. Una vez identificado un cambio en un elemento, se buscan sus reglas de verificación, las cuales representan los requisitos asociados a dicho elemento. Luego, éstas se comprueban sobre el estado actual de la GUI para asegurar que los requisitos sen satisfechos. Si todas las reglas se cumplen, el evento continúa el flujo normal hacia la lógica de aplicación para ser manejado. En caso contrario, la *Capa de Verificación* responde con el comportamiento adecuado (e.g. truncar el evento, mostrar un mensaje, colorear el elemento, sugerir el formato esperado, etc.)

Podemos diferenciar entre verificación *top-down* (intercepta los cambios realizados por el usuario para evitar entradas inválidas) y *bottom-up* (comprueba si los valores establecidos por la lógica de aplicación en la GUI también son correctos). Además, esta técnica es dinámica y permanente, ya que las reglas son comprobadas en tiempo de ejecución como si se trataran de un invariante en *Diseño por Contrato* [18]. Esta solución también está basada en el concepto AOSD, ya que el proceso de verificación se trata y gestiona como un aspecto aparte dentro de la propia aplicación. Por último, esta solución no utiliza un modelo *joint-point* como los lenguajes orientados a aspectos (AOP) más utilizados [6,7], sino que es no intrusiva en código.

En la Figura 3 podemos ver un ejemplo real de verificación *top-down*. El usuario establece el valor del campo *Buffer Size* a 25, el cual sólo soporta valores enteros menores o iguales que 20. Al abandonar dicho campo se genera un evento *Focus Lost*, el cual es interceptado por *la Capa de Verificación*. Una vez interceptado, se buscan las reglas asociadas al elemento y se comprueban sobre el estado actual de la GUI. Tras la comprobación (en este caso hay un error), se actualiza el fichero de log de verificación y se llevan a cabo las intervenciones GUI necesarias. En este caso, se establece un valor por defecto (i.e. 0) y se decora con color rojo para indicar el error durante la comprobación de requisitos.

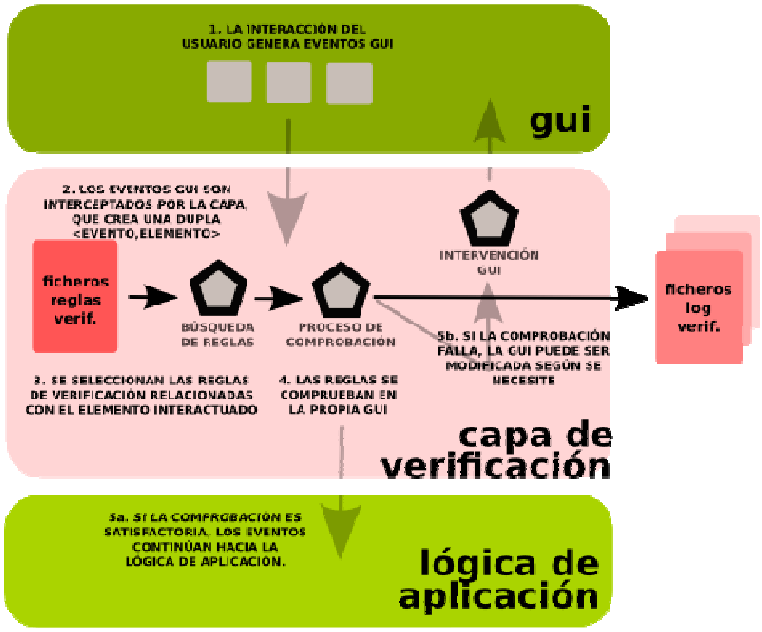


Figura 2. Funcionamiento de la Verificación top-down.

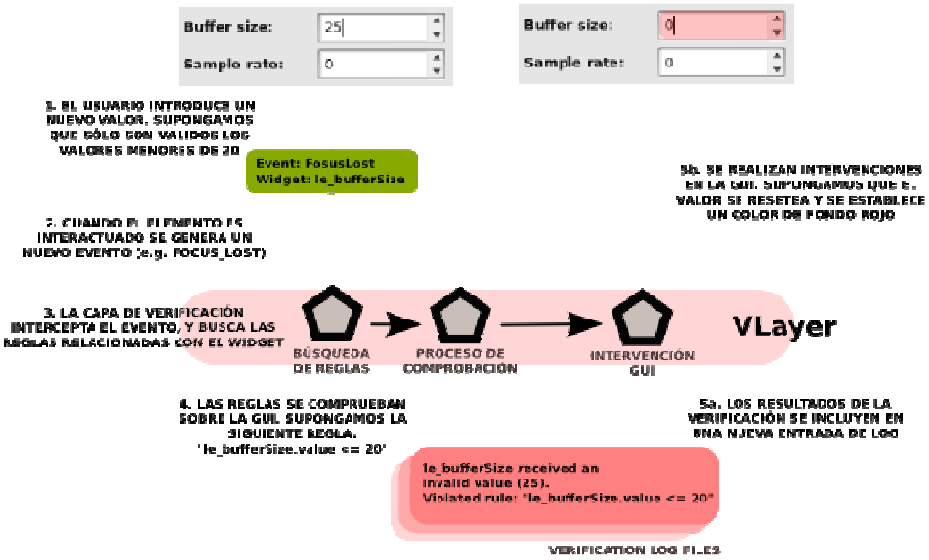


Figura 3. Ejemplo real de Verificación *top-down*.

3.2. Las reglas de verificación

Las reglas representan los requisitos SRS referentes a los valores de entrada/salida. Éstas pueden ser simples, y por lo tanto hacer referencia a un sólo elemento (e.g. "w1.value >= 10"), o pueden ser complejas, y hacer referencia a más de un elemento. Además, estas reglas pueden incluir estructuras de control de flujo, comparaciones, funciones, etc. (e.g. "if (w1.selectedIndex() == 2) then w2.isEmpty() == false"). Para definirlas es necesario un lenguaje que dé a los desarrolladores la flexibilidad suficiente para representar fielmente cualquier tipo de requisito. Es por esto por lo que la solución propuesta no está ligada a ningún lenguaje en concreto, sino que son los propios desarrolladores los que eligen el lenguaje a utilizar (e.g. lenguajes de restricciones como OCL [22], de script como Ruby [17] o JavaScript [19], etc.)

Ya que la solución propuesta es orientada a aspectos, se utilizan ficheros externos para separar las reglas del código fuente. Así, las reglas relacionadas se incluirían en el mismo fichero, y el resto en ficheros diferentes. La estructura de estos ficheros también queda a elección de los desarrolladores. De esta manera se aumenta la encapsulación y modularidad, facilitando la reutilización de la especificación de verificación. Además, facilita la realización de otros procesos tales como la depuración o la auditoría. Usar ficheros externos para las reglas también permite cambiar la especificación de verificación en tiempo de ejecución, muy útil durante el *Proceso de Desarrollo*. Por ejemplo, durante la fase de pruebas, los desarrolladores podrían cambiar o ajustar las reglas sin necesidad de recompilar ni de volver a lanzar la aplicación. Otro ejemplo sería el de cambiar la especificación completa en caso de que aparezca un nuevo diálogo en la GUI, cambie el estado de la aplicación, etc.

La Figura 4 representa el proceso de carga de reglas, las cuales se obtienen a partir de los ficheros facilitados por los desarrolladores (o los propios usuarios). Los ficheros de reglas deben ser cargados al lanzar la aplicación, y se debería actualizar el conjunto de reglas cada vez que alguno de estos ficheros cambie. Después, los elementos GUI se registran en la *Capa de Verificación* para que puedan ser verificados. Finalmente, en la fase de enlace, se decide qué reglas deberán ser comprobadas cuando se interactúa con un elemento concreto de la GUI.

4. Discusiones

La solución propuesta describe un entorno para diseñar, implementar, probar, y desplegar una solución completa de verificación en un software. Además, las técnicas de *Intervención GUI* mejoran la experiencia del usuario, ofreciendo un proceso de verificación más rico e interactivo. Tras las pruebas realizadas, se ha observado un coste computacional despreciable (≈ 0.3 milisegundos por cada proceso de verificación lanzado); además, se utilizan los intervalos de tiempo existentes entre las acciones que componen la interacción del usuario.

Su naturaleza orientada a aspectos hace que la solución propuesta sea no invasiva e independiente del código. La separación lógica entre código y reglas permite una mayor encapsulación y modularidad. Por una parte se mejora la legibilidad del código, facilitando los procesos de desarrollo y depuración. Por otra, mejora la eficiencia en los procesos que trabajan con las reglas, y mejora su reutilización. El hecho de que las reglas puedan ser modificadas en tiempo de ejecución es otra ventaja interesante de este modelo. Finalmente, poder utilizar lenguajes de script proporciona a los desarrolladores una gran flexibilidad al especificar las reglas.

Otro beneficio inherente parte de la necesidad de trabajar en profundidad, y desde el inicio del proceso de desarrollo, en el análisis de requisitos. Esto permitiría detectar posibles errores no detectados previamente en la fase de análisis, mejorando así la especificación de requisitos (SRS) y el propio diseño de la GUI [10] desde el inicio.

Entre sus principales limitaciones podemos destacar la desestructuración de los requisitos (reglas), la imposibilidad de establecer todos los requisitos en ciertos casos excepcionales (e.g. los requisitos que dependen de valores almacenados en BBDD para ser comprobados), y las limitaciones gráficas de los sistemas de ventanas con los que se programan los módulos de intervención GUI.

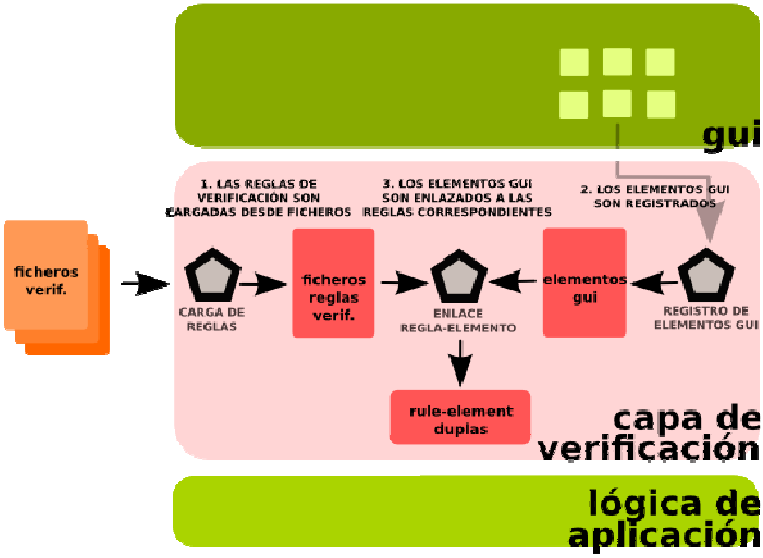


Figura 4. Proceso de carga de reglas.

5. Conclusiones y Trabajo Futuro

La verificación de los datos de entrada representa un elemento esencial y crítico dentro del Proceso de Desarrollo Software. Éste, junto con la complejidad asociada a los procesos de validación y verificación, son los principales motivos por los que se propone un framework abierto y adaptable que describe una capa de verificación orientada a aspectos, situada entre la GUI y la lógica de aplicación.

Esta solución AOSD se integra fácilmente en un software, y trabaja en paralelo mientras la aplicación está en ejecución. La lógica de verificación es transparente a los desarrolladores, y las reglas se incluyen en ficheros separados. Este framework permite implementar una solución completa de verificación que convierte al proceso en una parte integral de las etapas de desarrollo, pruebas, y ejecución del software. Además, su carácter abierto no lo restringe a ningún sistema de ventanas o lenguaje de verificación concretos.

Nuestro siguiente objetivo es trabajar en un framework más general y flexible, el cual no esté restringido sólo a los procesos de verificación. En dicho framework se pasaría de un modelo (Evento GUI → Comprobación de Reglas) a un modelo (Elemento GUI + Evento GUI → Acción). Ésto facilitaría la implementación ágil de soluciones GUI orientadas a aspectos, como por ejemplo el Prototipado Rápido de GUIs, GUI Mocking Ágil, procesos de GUI Logging & Auditing, Software Adaptable en tiempo de ejecución, etc., o incluso para usarlo como un nuevo modelo de desarrollo donde la definición de la GUI y los métodos manejadores están separados, y además son intercambiables.

Agradecimientos

Este trabajo ha sido parcialmente financiado por la *Cátedra SAES de la Universidad de Murcia* (http://www.um.es/catedrasaes). Este acuerdo engloba un esfuerzo conjunto entre la empresa *SAES (Sociedad Anónima Electrónica Submarina)* (http://www.electronica-submarina.com) y la *Universidad de Murcia*, para trabajar en software de código abierto y en sistemas de tiempo real y aplicaciones críticas.

Referencias

[1] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. Adding trace matching with free variables to AspectJ. In OOPSLA (2005), R. Johnson and R. P. Gabriel, Eds., ACM, pp. 345–364.

[2] Barringer, H., Goldberg, A., Havelund, K., and Sen, K. Rule-based Runtime Verification. Verification, Model Checking, and Abstract Interpretation (2004), Springer, pp. 44–57.

[3] Barringer, H., Havelund, K., Rydeheard, D. E., and Groce, A. Rule systems for runtime verification: A short tutorial. In RV (2009), Bensalem and Peled, Eds., vol. 5779 LNCS, Springer, pp. 1–24.

[4] Bodden, E. J-LO-A tool for runtime-checking temporal assertions. In AOSD'08: Proceedings of the 7th international conference on Aspect-oriented software development (2008), ACM, pp. 36–47.

[5] Chen, F., and Rosu, G. Mop: an efficient and generic runtime verification framework. In OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object Oriented programming systems and applications (New York, USA, 2007), ACM, pp. 569–588.

[6] Xerox Corp., AspectJ: The Aspect-Oriented language for Java. http://www.eclipse.org/aspectj/, 2010.

[7] Olaf Spinczyk et al,AspectC++: The Aspect-Oriented language for C++. http://www.aspectc.org/, 2010.

[8] Falcone, Y., Fernandez, J., and Mounier, L. Runtime Verification of safety-progress properties. In RV (2009), S. Bensalem and D. Peled, Eds., vol. 5779 of LNCS, Springer, pp. 40–59.

[9] Feigenbaum, B., and Squillace, M. Accessibility validation with raven. In WoSQ '06: Proceedings of the 2006 international workshop on Software quality (New York, NY, USA, 2006), ACM, pp. 27–32.

[10] Hayes, J. H., and Offutt, J. Input validation analysis and testing. Empirical Software Engineering 11, 4 (2006), 493–522.

[11] Ierusalimschy, R. et al. Lua - The Programming Language. http://www.lua.org/, 2010.

[12] Kima, M., Leeb, I., Sammapunb, U., Shinb, J., and Sokolskyb, O. Monitoring, checking, and steering of real-time systems. In RV'02, (FLoC Satellite Event) (December 2002), vol. 70, Elsevier, pp. 95–111.

[13] Ligatti, J., Bauer, L., and Walker, D. Enforcing non-safety security policies with program monitors. In ESORICS (2005), Vimercati, Syverson, and Gollmann, Eds., vol. 3679 of LNCS, Springer, pp. 355–373.

[14] Liu, H., and Tan, H. B. K. Covering code behavior on input validation in functional testing. Information and Software Technology 51, 2 (2009), 546 – 553.

[15] Mateo, P., Sevilla, D., and Martínez G., Open HMI Tester: An Open and Cross-Platform Architecture for GUI Testing and Certification. JCSSE (The International Joint Conference on Computer Science and Software Engineering) Special Issue on Open Source Certification (February 2009).

[16] Mateo, P., Sevilla, D., and Martínez, G. A Proposal for Automatic Testing of GUIs Based on Annotated Use Cases. Advances in Software Engineering 2010, 671284 (2010).

[17] Matsumoto, Y. The Ruby Scripting Language. http://www.ruby-lang.org/, 2010.

[18] Meyer, B. Applying "design by contract". Computer 25, 10 (1992), 40–51.

[19] Sun Microsystems. JavaScript Scripting Language. http://developers.sun.com/javascript/, 2010.

[20] Mateo, P., Martínez, G., and Sevilla, D. Open HMI Tester: un Framework Open-source para Herramientas de Pruebas de Software. Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos 3, 4 (2009).

[21] Mateo, P., Martínez, G., and Sevilla, D. Automated GUI Testing Validation Guided by Annotated Use Cases. In GI Jahrestagung (2009), Fischer, Maehle, and Reischuk, Eds., vol. 154 of LNI, pp. 2796–2804.

[22] Object Management Group. Object Constraint Language (OCL), version 2.0, OMG document formal/2006-05-01, 2006. http://www.omg.org/spec/OCL/2.0/.

[23] IEEE. IEEE 1012-2004 - ieee standard for software verification and validation. IEEE, IEEE, pp. 0-110. Revision of IEEE Std 1012-1998.

[24] Cátedra SAES of the University of Murcia, OpenHMI-Tester Prototype. http://sourceforge.net/projects/openhmitester/, 2009.

[25] Schneider, F. B. Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3, 1 (2000), 30–50.

[26] Zee, K., Kuncak, V., Taylor, M., and Rinard, M. C. Runtime checking for program verification. In RV (2007), O. Sokolsky and S. Tasiran, Eds., vol. 4839 of LNCS, Springer, pp. 202–213.

Pruebas Funcionales Evolutivas en Industria

Tanja E.J. Vos
Centro ProS
Universidad Politécnica de Valencia
tvos@pros.upv.es

Arthur I. Baars
Centro ProS
Universidad Politécnica de Valencia
abaars@pros.upv.es

Beatriz Marín
Centro ProS
Universidad Politécnica de Valencia
bmarin@pros.upv.es

Resumen

Las pruebas evolutivas han sido investigadas por la academia, obteniendo resultados prometedores. Sin embargo, estas pruebas casi no han sido aplicadas a sistemas complejos reales, por lo que se tiene escaso conocimiento acerca de la escalabilidad, aplicabilidad, y aceptabilidad de las pruebas evolutivas en entornos industriales. Este artículo contribuye a mejorar esta situación presentando los resultados obtenidos en el proyecto europeo EvoTest (IST-33472).

1. Introducción

Hoy en día, la técnica de aseguramiento de calidad más importante y más usada en la industria corresponde a las pruebas de software, que pueden tomar más del 50% del tiempo y coste de desarrollo [12]. Aunque actualmente existen herramientas que automatizan las pruebas, ayudando a planificarlas, controlarlas, y ejecutar y monitorizar los casos de prueba; todas ellas comparten una filosofía pasiva respecto al diseño de casos de prueba, la selección de los datos de prueba, y la evaluación de las pruebas. Esta filosofía es pasiva, ya que la persona que realiza las pruebas debe realizar estas actividades cruciales de forma manual. Esto se debe a que estas actividades son difíciles de automatizar de manera efectiva y escalable con las técnicas usadas por la industria, ya que la cantidad de posibles casos de prueba es demasiado grande para ser explorado exhaustivamente, incluso para programas triviales.

Teniendo en cuenta que los casos de prueba determinan el tipo y el alcance de una prueba, el diseño de éstos es *la actividad* que determina la calidad y efectividad de todo un proceso de pruebas. La falta de automatización de esta actividad primordial ha provocado que la industria gaste grandes cantidades de esfuerzo y dinero en las pruebas del software, y que la calidad de las pruebas resultantes es baja y no encuentran errores importantes del sistema.

Para generar automáticamente casos de prueba efectivos, técnicas como la optimización estocástica y las búsquedas inteligentes (ej. algoritmos evolutivos) han sido aplicadas exitosamente en la academia. Respecto a las pruebas de caja negra, una parte considerable de la literatura ha demostrado el éxito de las pruebas evolutivas frente a métodos tradicionales para las pruebas temporales [14][15][16], donde la prueba es interpretada como una optimización del problema y se emplea computación evolutiva para encontrar datos de prueba con tiempos de ejecución cortos y largos. Además, para el caso de las pruebas funcionales de caja negra, en [17][18][19] se muestran resultados prometedores para las pruebas evolutivas. Sin embargo, las pruebas evolutivas han sido escasamente aplicadas a sistemas complejos reales, por lo que se tiene poco conocimiento acerca de su escalabilidad y aceptabilidad en entornos industriales.

Este artículo describe dos casos de estudio realizados en el contexto del proyecto EvoTest [11] que reportan la escalabilidad, aplicabilidad, y aceptabilidad de las pruebas funcionales evolutivas de caja negra en entornos industriales. El artículo está organizado de la siguiente manera: la Sección 2 presenta los fundamentos utilizados en los estudios, la Sección 3 presenta el diseño de los casos de estudio, y la Sección 4 presenta la ejecución de los estudios. La Sección 5 presenta la evaluación de los resultados, y la Sección 6 presenta las conclusiones y trabajos futuros.

2. Pruebas Evolutivas

Para automatizar las pruebas de software utilizando pruebas evolutivas, el objetivo que persigue una prueba debe ser transformado en una tarea de optimización. Dependiendo del objetivo perseguido por la prueba, diferentes funciones de adecuación («*fitness funtion*» o «*objective funtion*») se deben especificar para la evaluación

de los datos de prueba. Si se puede definir una función de adecuación para el objetivo de la prueba, y se utiliza computación evolutiva para la búsqueda, las pruebas evolutivas se aplican así:
1) Se genera, usualmente de manera aleatoria, un conjunto inicial de datos de prueba.
2) Cada individuo dentro de la población (conjunto inicial) representa un dato de prueba con el cual el sistema sometido a pruebas («*Software Under Test*» - SUT) es ejecutado.
3) Por cada dato de prueba, se monitoriza la ejecución y se determina el valor de adecuación para el individuo correspondiente.
4) Los datos de prueba con altos valores de adecuación son seleccionados con una mayor probabilidad que aquellos que tienen bajos valores de adecuación, y se someten a procesos de combinación y mutación para generar nuevos datos de prueba (hijos).
5) Se genera una nueva población de datos de prueba mezclando los datos de prueba hijos y padres mediante procedimientos de supervivencia.

Este proceso de aplicación de pruebas evolutivas al software se repite hasta que el objetivo de la prueba se ha alcanzado o se ha alcanzado otra condición de detención dada.

2.1. Pruebas Funcionales Evolutivas

Cuando se realizan pruebas funcionales, el SUT es considerado como una caja negra, es decir, el probador no tiene conocimientos sobre cómo se ha implementado el software, y se limita a observar el comportamiento de sus entradas y salidas. El objetivo de las pruebas funcionales es verificar que el SUT satisface los requisitos funcionales.

Respecto a las pruebas funcionales evolutivas, se ha llevado a cabo poca investigación en comparación con la amplia investigación realizada respecto a las pruebas estructurales (ej. [1][2]). Para realizar una prueba funcional evolutiva, se utiliza un algoritmo evolutivo que busca datos de prueba para los cuales el comportamiento del SUT sea defectuoso con respecto a una propiedad funcional específica. Por este motivo, es necesario definir una función de adecuación que mida cómo de cerca está un dato de prueba de quebrar una propiedad seleccionada. Durante la búsqueda evolutiva, cada generación sucesiva de datos de prueba conseguirá estar más cerca de quebrar la propiedad funcional seleccionada. Así, cuando el óptimo es alcanzado se descubre un defecto, es decir, un dato de prueba para el cual el SUT falla en la satisfacción de un requisito. Si no se descubren estos datos de prueba, entonces se adquiere confianza en la correcta implementación de la propiedad funcional.

2.2. Entorno para Pruebas Evolutivas

Para facilitar el desarrollo de pruebas evolutivas, el proyecto EvoTest ha implementado un entorno para ejecutar estas pruebas («*Evolutionary Testing Framework*» - ETF). El entorno ETF es una extensión de Eclipse [9] que incluye el generador evolutivo GUIDE [10] y un componente de interfaz gráfica para afinar los parámetros del algoritmo. ETF provee varias vistas para visualizar el progreso de la búsqueda. Además, ETF provee un punto de extensión para que generadores de otras técnicas meta-heurísticas de búsqueda puedan ser conectadas fácilmente.

Para personalizar el entorno para un objetivo de prueba particular, se deben proveer los siguientes componentes específicos de dominio (vea el manual de usuario de ETF [11]):
1) Una especificación que describe la estructura de los individuos, es decir, los datos de prueba.
2) Un administrador («*driver*») de pruebas, que permite conectar ETF con el SUT. El driver convierte los individuos del proceso de búsqueda en datos de prueba, ejecuta el SUT usando estos datos, monitoriza la salida del SUT, y entrega los resultados de la monitorización a ETF.
3) Una función de adecuación, que calcula la adecuación de los datos de prueba usando los resultados de la monitorización.

Así, para establecer el entorno ETF para pruebas funcionales evolutivas, es necesario desarrollar el driver de pruebas y la función de adecuación como plugins de Eclipse para que sean cargados y ejecutados por ETF. La especificación individual se realiza en un archivo XML. Cuando el SUT dependa de señales de entrada continuas, el entorno ETF permite usar una herramienta generadora de señales, como han propuesto Windisch y Al Moubayed [3].

3. Diseño del Caso de Estudio

Ninguno de los enfoques automáticos para el diseño de pruebas propuestos por la academia ha alcanzado un nivel de aplicabilidad en entornos industriales [4][5][6]. Además, existe un alto interés en estudios empíricos de técnicas de pruebas evolutivas y su escalabilidad en prácticas

reales de la industria. Así, la pregunta de investigación a ser resuelta por este trabajo es:

¿Las pruebas evolutivas de caja negra son escalables a las prácticas reales en la industria?

3.1. Selección de los casos

Para asegurar que los casos seleccionados corresponden a sistemas complejos reales, nuestros socios industriales Daimler y BMS han elegido sistemas para los cuales decidieron que la aplicación exitosa de pruebas evolutivas es más importante en sus entornos industriales.

Dado que la aplicación exitosa de pruebas evolutivas en entornos industriales es el objetivo más importante del caso de estudio, los socios industriales no fueron forzados con demasiadas restricciones para seleccionar un sistema que se ajuste mejor al estudio empírico. Aunque el acceso a previas versiones (con información de sus fallos, cobertura y técnicas usadas para encontrar los fallos) hubiera sido ideal para nuestro estudio, esto no siempre fue posible en el entorno industrial, y otras veces no se cumplía con las necesidades y deseos del socio industrial.

Daimler y BMS han elegido sistemas de control embebidos para el dominio automoción que provienen de su producción en serie. Daimler ha elegido un módulo de control de velocidad adaptable («*Adaptive Cruise Control*» - ACC) cuyo propósito mantener una velocidad dada, y además controlar la distancia hacia los vehículos que le preceden – medida usando radares – para prevenir accidentes. BMS ha seleccionado un sistema de anti-bloqueo de frenos («*Anti-lock Braking System*» - ABS) que previene que las ruedas de un vehículo se bloqueen mientras se está frenando, permitiendo así que el conductor mantenga el control de la dirección durante las frenadas fuertes, y en la mayoría de las situaciones, acortando las distancias de frenado.

3.2. Proposiciones

Hemos refinado nuestra pregunta de investigación en 5 proposiciones que pueden ser evaluadas a través de variables serán medidas durante la ejecución del estudio. Para hacer esto, hemos estudiado y discutido profundamente con los socios industriales para entender qué significa ser escalable en estas aplicaciones industriales de acuerdo a los resultados de las pruebas evolutivas.

Proposición 1: Es posible usar el ETF sin conocimientos detallados sobre computación evolutiva para buscar datos de prueba interesantes.
Proposición 2: El ETF aplicado a ejemplos de tamaño real puede generar más casos de prueba orientados al problema que las pruebas aleatorias. Esto se debe a que mientras los valores de adecuación son mejorados, el sistema está continuamente siendo ejercitado más cerca de una condición de frontera (un valor de adecuación óptimo quiebra la frontera, ej. el requisito).
Proposición 3: El ETF es más efectivo que las pruebas aleatorias en la detección de errores revelados por los casos de prueba cuando se aplica a sistemas reales para pruebas de caja negra.
Proposición 4: La afinación automática de parámetros («*Automated Parameter Tuning*») mejora los resultados de las búsquedas en términos de su efectividad y eficiencia.
Proposición 5: Después de la instalación de ETF, el tiempo y esfuerzo que toma su configuración para su aplicación a sistemas reales es conveniente para la industria en comparación con las técnicas actualmente utilizadas.

3.3. Procedimientos de Recolección de Datos

Los siguientes pasos comunes han sido definidos para llevar a cabo los estudios:

1. Instalación y configuración del ETF de acuerdo al manual de usuario [11]; mantenimiento de diarios de trabajo que deben contener las tareas (incluyendo su fecha, duración y descripción) realizadas para configurar el ETF.
2. Implementación de componentes específicos (especificación de los individuos y drivers de pruebas) para los casos de estudio. Los diarios de trabajo deberán ser mantenidos para posibilitar la estimación del esfuerzo necesario.
3. Definir, refinar, e implementar la función de adecuación y validar su conveniencia para quebrar el requisito. Los diarios de trabajo deberán ser mantenidos.
4. Ejecutar cada búsqueda 30 veces para otorgar significado estadístico.
5. Entrevistas relacionadas con la idoneidad y aceptabilidad en el entorno industrial específico. Estas entrevistas son informales y no intentan realizar pruebas estadísticas sobre los datos porque los ejemplos de Daimler y BMS no son representativos para la población de interés. Sin embargo, estas entrevistas son interesantes [20]

dado que su objetivo es conseguir las experiencias que los profesionales tuvieron cuando aplicaron las técnicas y herramientas para pruebas evolutivas. Se fomentará que los entrevistados elaboren áreas importantes para ellos y que puedan desviarse de las preguntas de la entrevista. Por este motivo, las entrevistas son altamente interactivas, es decir, los entrevistadores pueden clarificar preguntas a los entrevistados e investigar respuestas inesperadas, y también pueden ganar la confianza de un entrevistado para mejorar la calidad de sus respuestas.

3.4. Procedimientos de Análisis de Datos

Los estudios se realizarán para controlar las variables independientes y medir su efecto en las variables dependientes, que han sido elegidas para dar respuesta a las proposiciones planteadas.

Variables Independientes

- Sistemas reales con su complejidad inherente.
- El conjunto de parámetros usados para configurar el generador evolutivo ETF (detalles de los parámetros pueden ser consultados en [10]). Se tienen cuatro configuraciones de parámetros diferentes asociadas a diferentes versiones de ETF: *ETF_Random* usa búsquedas aleatorias en vez de búsquedas evolutivas; *ETF_Default* usa un conjunto de parámetros seleccionado por defecto por el generador evolutivo; *ETF_Manual* usa parámetros configurados en base a la experiencia del probador según la observación del rendimiento de diferentes conjuntos de parámetros; y *ETF_Automated* usa técnicas de afinación automática de parámetros, las cuales son ejecutadas por el generador evolutivo para elegir el "mejor" conjunto de parámetros.

Variables Dependientes

a. Número de casos de prueba evaluados. Esto es equivalente al número de veces que el SUT es ejecutado para un requisito específico y al número de evaluaciones de la función de adecuación.
b. Número de casos de prueba inválidos que fueron generados.
c. Número de errores encontrados.
d. Progreso de los valores de adecuación.
e. Tiempo y esfuerzo necesitado para encontrar un conjunto de parámetros apropiado para el generador evolutivo - medido según el tiempo usado por los desarrolladores.
f. Número de versiones o refinamientos de funciones de adecuación que respectivamente fueron realizados.
g. Tiempo necesitado para definir y refinar las funciones de adecuación - medido según el tiempo usado por los desarrolladores.
h. Tiempo y esfuerzo necesitado para instalar el ETF - medido según el tiempo usado por los desarrolladores.
i. Tiempo y esfuerzo necesitado para personalizar el ETF para un objetivo de prueba específico, es decir, desarrollo del archivo XML para la especificación de los individuos, plugins de eclipse para el driver de pruebas y la función de adecuación - medido según el tiempo usado por los desarrolladores.
j. Líneas de código de los componentes específicos que necesitan ser integradas en el ETF para las pruebas funcionales.
k. Aplicabilidad y aceptabilidad en entornos industriales.

Por cada sistema industrial seleccionado, los estudios serán ejecutados con las cuatro versiones de ETF. Así, en total se realizarán 8 estudios.

3.5. Amenazas a la Validez

La *Validez de la Construcción* está relacionada a la admisibilidad de declaraciones acerca de los fundamentos tomados como base para el estudio. En este sentido, posibles amenazas son:

- El punto en el cual los errores en los sistemas están relacionados a los requisitos que intentamos quebrar. Para mitigar este riesgo, los errores encontrados fueron investigados para evaluar si efectivamente quiebran el requisito que se está probando.
- Las entrevistas no son representativas para la población de interés. Por esta razón, estas entrevistas son informales y no intentamos realizar pruebas estadísticas sobre los datos. Los datos son usados para encontrar errores en la herramienta, áreas que necesitan ser mejoradas, o extensiones necesarias para entornos industriales específicos.
- El tiempo de desarrollo medido para algunas variables depende de la experiencia y el nivel de educación de la gente involucrada. Para

mitigar esta amenaza, ambas empresas han seleccionado personas con perfiles similares para los casos de estudio. Este perfil corresponde a personas que normalmente ejecutan pruebas de manera automática.

- El generador de señales de ETF puede no permite la generación de señales totalmente realistas. Para mitigar este riesgo, la cantidad inválida de casos de prueba es medida en la variable dependiente *b*.

La *Validez Interna* está relacionada a la interpretación de los resultados, es decir, describe la certeza con la que los cambios en las variables dependientes son atribuidos a los cambios en las variables independientes. En este sentido, posibles amenazas son:

- La experiencia de la gente usando el ETF con computación evolutiva o pruebas evolutivas. La gente con experiencia en afinamiento de algoritmos evolutivos ejecutará mejor las pruebas que personas que no tienen esta experiencia, ya que seleccionan mejor los parámetros que pueden conducir una búsqueda más eficiente. Esta amenaza es mitigada seleccionando a personas que tengan conocimientos básicos sobre computación evolutiva.
- Inestabilidades y errores del ETF en fases tempranas del desarrollo pueden dificultar la ejecución de los estudios.
- La aleatoriedad intrínseca de los algoritmos evolutivos. Esta amenaza es mitigada ejecutando cada caso de estudio varias veces.

La *Validez Externa* puede influenciar el punto en el cual las conclusiones pueden ser generalizadas. En este sentido, posibles amenazas son:

- La representatividad de los sistemas seleccionados como casos de estudio, ya que ambos corresponden al dominio automoción. Esto puede causar que los resultados sean válidos sólo en este contexto industrial. Repetir los casos de estudio en otros dominios puede entregar más información acerca de las generalizaciones de los resultados.
- Las señales de entrada generadas pueden ser estrictamente sintéticas (sin ruido) y podrían no ser generalizables para sistemas reales.
- La cantidad de repeticiones necesarias para obtener significado estadístico puede tomar demasiado tiempo en entornos industriales.

4. Recolección de Datos

Para obtener una línea base de comparación, un número aleatorio de búsquedas fue realizado sin usar ningún tipo de optimización en el generador (*ETF_Random*). Después, las búsquedas fueron realizadas usando los parámetros especificados por defecto (*ETF_Default*). Posteriormente, los resultados de los estudios fueron optimizados adaptando los parámetros de optimización al problema específico seleccionado (*ETF_Manual*). Finalmente, un conjunto de búsquedas fueron llevadas a cabo usando *ETF_Automated*, que automatiza la costosa tarea de afinar los parámetros de optimización durante el progreso hacia la optimización final. Los parámetros que deben ser especificados son aquellos que limitan el tamaño de la población y el número de las generaciones evolutivas durante la optimización.

Durante estas actividades, se recolectaron los siguientes datos para las variables dependientes a hasta j y los sistemas ACC (Adaptive Cruise Control) y ABS (Anti-lock Braking System).:

	ACC	**ABS**
a	1,2 millones de pruebas ejecutadas	150.000 millones de pruebas ejecutadas
b	0	0
c	0	0
d	Vea Figura 1	Vea Figura 2
e	1 persona-mes	1 persona-mes
f	2	7
g	0,5 persona-mes	1 persona-mes
h	2 horas	2 horas
i	3 persona-mes	4 persona-mes
j	365	5195

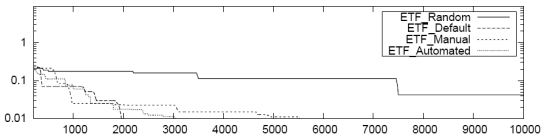


Figura 1: ACC

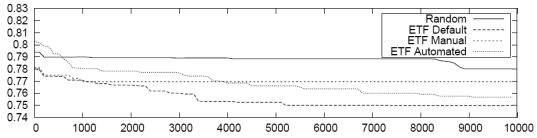


Figura 2: ABS

5. Evaluación

En esta sección se presentan respuestas a la pregunta de investigación planteada (*¿Las pruebas evolutivas de caja negra son escalables a las prácticas reales en la industria?*) utilizando el conjunto de proposiciones presentadas en la Sección 3.2 y las variables dependientes que fueron medidas.

Proposición 1: Respecto a las pruebas funcionales de caja negra, *no* es posible usar el entorno ETF para pruebas evolutivas sin conocimientos detallados sobre computación evolutiva para buscar datos de prueba interesantes, ya que es necesario diseñar funciones de adecuación sofisticadas. La función de adecuación es un factor crucial para el éxito de búsquedas evolutivas y, por lo tanto, para el proceso de generación de datos de pruebas evolutivas en general. La falta de experiencia en el diseño de funciones de adecuación convenientes puede fácilmente producir errores como la creación de óptimos locales o introducir riesgos similares que dificultan la búsqueda de los datos de prueba deseados.

Respecto a los parámetros del generador evolutivo, es necesario proveer consejos acerca de cómo realizar el afinamiento manual del transformador evolutivo utilizando computación evolutiva experta. Sin embargo, si el afinamiento manual no es necesario, entonces, se necesita poco conocimiento para buscar datos de pruebas interesantes, como se puede ver en los resultados de *ETF_Default* que no son tan malos en comparación con *ETF_Manual*.

Proposición 2: Las Figuras 1 y 2 muestran que el ETF genera un gran conjunto de casos de pruebas orientados al problema. Las pruebas aleatorias fueron incapaces de alcanzar resultados comparables con las generaciones evolutivas. Como era esperado y de acuerdo a sus capacidades usuales, las pruebas aleatorias gradualmente encontraron ligeramente mejores soluciones durante la optimización sin alcanzar áreas del espacio de búsqueda cercanos a la solución optima, sin mencionar la solución global. En contraste, el generador evolutivo, es decir, *ETF_Default*, *ETF_Manual* y *ETF_Automated* mostró un comportamiento que se acerca a la solución. Después de alcanzar áreas interesantes, ellos tratan de acercarse al óptimo generando soluciones ligeramente diferentes que son también casos de pruebas altamente orientados al problema, porque su proximidad a las buenas soluciones ya se ha encontrado.

Proposición 3: Los sistemas elegidos por los socios industriales para investigar la escalabilidad de las pruebas evolutivas provienen de su producción en serie.. Por lo tanto, durante el estudio fue poco probable encontrar errores. Así, no es posible evaluar esta proposición en el contexto los casos de estudio seleccionados. Esperamos que repeticiones de los casos de estudio en trabajos futuros posibiliten comentar sobre este aspecto.

Proposición 4: El afinamiento automático de parámetros funciona bien con respecto a la efectividad y eficiencia dada, obteniendo, en promedio, los mismos resultados que el afinamiento manual, pero relevando al probador de la tediosa y difícil tarea de especificar manualmente los parámetros del generador evolutivo que resultó ser considerable (entre 0,5 y 1 persona-mes - vea la variable dependiente e). Así, el afinamiento automático de parámetros contribuye a la aplicabilidad en la industria, donde asignar recursos humanos al afinamiento es más caro que utilizar recursos computacionales. Sin embargo, se debe tener en cuenta que si la ejecución del SUT toma una gran cantidad de tiempo (que es frecuentemente el caso con pruebas funcionales de caja negra), puede también requerir muchos recursos computacionales.

Proposición 5: Después de la instalación del entorno ETF para pruebas evolutivas, el tiempo y el esfuerzo tomado en su configuración para aplicarlo a las prácticas reales que son realizadas para la ejecución de pruebas funcionales fue encontrado razonable dentro de los entornos industriales estudiados. Si bien la creación de la función de adecuación es una tarea que consume tiempo y requiere conocimientos específicos de computación evolutiva, los socios industriales encontraron que este esfuerzo es compensado por la gran cantidad de casos de prueba orientados al problema generados y después evaluados automáticamente.

Cuando la función de adecuación ha sido diseñada, el proceso de generación de datos es llevado a cabo de manera completamente automática sin ningún esfuerzo humano. Mientras tanto el SUT es ejecutado con un número de datos

de prueba específicos del problema excepcionalmente grande y altamente relevantes, los cual no podría ser posible cuando se hace manualmente. Como puede ser visto en los datos recolectados, el SUT ha sido ejecutado 1.2 millones de veces o con 150.000 entradas diferentes, respectivamente.

Algunos problemas de usabilidad e ideas para mejoras adicionales obtenidos en las entrevistas informales estuvieron relacionados a la necesidad de interacción más directa del usuario con las pruebas funcionales de ETF. Por ejemplo, la página de preferencia provista para configurar manualmente los parámetros del generador evolutivo puede ser mejorada. Además, proveer guías sobre cómo desarrollar la función de adecuación serían muy apreciadas.

6. Conclusiones

En este artículo hemos mostrado que la implementación del ETF para las pruebas funcionales evolutivas es escalable en el dominio automoción. Los socios industriales se mostraron positivos acerca de la eficiencia de la herramienta y encontraron los resultados interesantes. El afinamiento automático de parámetros generalmente no requiere conocimientos avanzados sobre computación evolutiva. Sin embargo, para definir y refinar una función de adecuación apropiada, es necesario tener habilidades de computación evolutiva y debe ser factible dedicar una cantidad de tiempo significante para realizar esta tarea. Aunque en algunos casos se compensa con los resultados, todavía es un factor importante que influye en la aceptabilidad total de las pruebas funcionales evolutivas en la industria.

Para una aceptación más amplia en entornos industriales es importante automatizar el afinamiento de parámetros para ofrecer la posibilidad de que probadores inexpertos usen pruebas evolutivas y obtengan resultados interesantes. Lindlar [7] presenta un enfoque para simplificar el proceso de diseñar espacios de búsqueda y funciones de adecuación para las pruebas funcionales evolutivas, y de esta manera incrementar su aceptabilidad.

Además, la difícil tarea de diseñar una función de adecuación conveniente puede ser apoyada usando un enfoque basado en asistentes (wizards) o preguntas/respuestas. Probadores más avanzados demandarán la combinación de diferentes objetivos de pruebas en una búsqueda (ej. pruebas funcionales combinadas con pruebas del tiempo de respuesta), también llamadas búsquedas multi-objetivos. Distintas estrategias para sembrar datos de prueba [8] y paralelizar las pruebas son requeridas (ej. usando objetivos de pruebas múltiples y distribuidas) para diseñar una función de adecuación.

Un conjunto común de guías de desarrollo acerca de la utilización de las pruebas evolutivas como técnica de optimización podría incrementar la aceptación en la industria de estas pruebas, ya sea para usuarios medios y avanzados. Existe una necesidad real para sistemas embebidos de tener guías sobre qué técnicas de pruebas usar para diferentes objetivos de pruebas, diferentes niveles de pruebas, o diferentes fases del desarrollo de software; cuándo estas técnicas contribuyen con la fiabilidad general y la dependencia del sistema embebido; y qué tan eficiente y usable es su aplicación. Estas guías no existen para técnicas de pruebas tradicionales, e incluso se tiene menos conocimiento sobre pruebas evolutivas. Estudios empíricos son esenciales para sentar las bases para estas guías y así poder integrarlas en una estrategia general de *Pruebas y Aseguramiento de Calidad* para sistemas embebidos. Para esto, contar con un repositorio central de sistemas embebidos de ejemplo que puedan actuar como punto de referencia para la evaluación de técnicas de pruebas evolutivas para propiedades funcionales y no funcionales sería de ayuda.

Así, esperamos que el trabajo de este artículo constituya el primer paso para construir una familia de casos de estudio, como ha sido recomendado por Basili et. al. [13]. Los casos de estudio deben investigar la aplicabilidad de las pruebas evolutivas en la industria, y proveer señales claras acerca de las barreras que pueden evitar su amplia adopción en la industria. Finalmente, invitamos a otros investigadores a repetir nuestro estudio en otros entornos industriales, para posteriormente construir un repositorio con los resultados obtenidos.

Agradecimientos

Este trabajo ha sido soportado por el proyecto Europeo IST-33472. Queremos agradecer por su soporte y ayuda a Andreas Windisch y Felix Lindlar de Daimler; Joachim Wegener y Peter Kruse de Berner&Matter; Mark Harman, Kiran Lakhotia y Youssef Hassoun de Kings College London; Marc Schoenauer y Luis da Costa de

INRIA; Jochen Hänsel de Fraunhofer; Dimitar Dimitrov y Ivaylo Spasov de RILA; y Dimitris Togias de European Dynamics.

Referencias

[1] Buhler O,Wegener J. Automatic testing of an autonomous parking system using evolutionary computation.Proceedings of SAE 2004 World Congress, 2004; pages 115–122.

[2] Windisch A, Lindlar F, Topuz S, Wappler S. Evolutionary functional testing of continuous control systems. Proceedings of the 11th GECCO conference, ACM, 2009; 1943–1944.

[3] Windisch A, Al Moubayed N. Signal generation for search-based testing of continuous systems. Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops, IEEE Computer Society, 2009; 121–130.

[4] Hyunsook ES D, Rothermel G. Infrastructure support for controlled experimentation with software testing and regression testing techniques. Proceedings of the International Symposium On Empirical Software Engineering, ISESE '04, ACM, 2004; 60–70.

[5] Juristo, M, Vegas S. Reviewing 25 years of testing technique experiments. Journal of Empirical SW Eng 2004; 9(1):7–44.

[6] Briand LC. A critical analysis of empirical research in software testing, ESEM 2007, 1–8.

[7] Lindlar F. Search-based functional testing of embedded software systems. Doctoral Symposium in conj with IEEE ICST 2009.

[8] Arcuri A, White DR, Clark J, Yao X. Multi-objective improvement of software using co-evolution and smart seeding. Proceedings of SEAL'08, LNCS, 5361, 2008; pp 61–70.

[9] Eclipse. http://www.eclipse.org. Last accessed 2010-02-03.

[10] GUIDE. http://gforge.inria.fr/projects/guide/. Last accessed 2010-02-03.

[11] ETF user manual. http://www.evotest.eu. Last accessed 2010-02-03.

[12] Beizer B. Software Testing Techniques. International Thomson Computer Press, 1990.

[13] Basili VR, Shull F, Lanubile F. Building knowledge through families of experiments. IEEE TSE 1999; 25(4):456–473.

[14] Mueller F, Wegener J. A comparison of static analysis and evolutionary testing for the verification of timing constraints. Proceedings of the Fourth IEEE RTAS, IEEE, 1998; 144.

[15] Sthamer H, Wegener J. Using evolutionary testing to improve efficiency and quality in software testing. Proceedings of 2nd Asia-Pacific Conf on Software Testing, 2002.

[16] Gros HG. Evaluation of dynamic, optimisation-based worst-case execution time analysis. Int. Conference on Information Technology: Prospects and Challenges in the 21st Century, vol. 1, 2003; pp. 8–14.

[17] Wegener J, Bühler O. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. Proceedings of GECCO, 2004; 1400–1412.

[18] Baresel A, Pohlheim H, Sadeghipour S. Structural and functional sequence test of dynamic and state software with evolutionary algorithms. GECCO2003; 2428–2441.

[19] Windisch A, Lindlar F, Topuz S, Wappler S. Evolutionary functional testing of continuous control systems. GECCO, 2009; 1943–1944.

[20] Lethbridge TC, Sim SE, Singer J. Studying software engineers: Data collection techniques for software field studies. Empirical Softw. Engg. 2005; 10(3):311–341.

FITTEST – Testeo del Internet del Futuro

Tanja E.J. Vos
Centro ProS
Universidad Politécnica de Valencia
46022 Valencia
tvos@pros.upv.es

Oscar Pastor
Centro ProS
Universidad Politécnica de Valencia
46022 Valencia
opastor@pros.upv.es

Resumen

Con socios de España (Universidad Politecnica de Valencia), Inglaterra (University College London), Alemania (Berner & Mattner), Israel (IBM), Italia (Fondazione Bruno Kessler), Holanda (Utrecht University), Francia (Softeam) y Finlandia (Sulake), el proyecto Europeo FITTEST quiere abordar los desafíos del testeo del Internet del Futuro con técnicas de Testeo de Software Basado en la Búsqueda.

1. El Internet del Futuro

El Internet del Futuro (IdF) consistirá de una interacción compleja de servicios, aplicaciones, contenido y media, posiblemente aumentado con información semántica y basada en tecnologías que ofrecen una experiencia mejorada para el usuario extendiendo la actual navegación basado en hipervínculos. Nuestra sociedad cada vez más depende de los servicios que ofrece el IdF para actividades públicas como seguridad social, gobierno, educación, bancos, industria tanto como entretenimiento. Consecuentemente, las aplicaciones que ejecutan sobre el IdF necesitan satisfacer estándares estrictas de calidad y dependabilidad. Aplicaciones del IdF se caracterizan teniendo un nivel extremo de dinamismo. La mayoría de las decisiones que normalmente se toman durante el diseño, ahora se retrasan para tomar durante la ejecución, cuando la aplicación puede usar información obtenida a través de monitorización para adaptarse a contexto del usuario. La realización de esta visión depende de varias tecnologías, como:

- Reflección y monitorización para recoger información necesaria para decisiones durante el tiempo de ejecución.
- Descubrimiento y composición dinámica de servicios.
- Auto-adaptación.
- Comunicación asíncrono.
- Alta configurabilidad y conciencia de contexto.
- Componer sistemas de sistemas.

Ofreciendo una importante mejora sobre la Red actual, la complejidad de las tecnologías involucradas hacen que el testeo del IdF es extremadamente desafiante.

2. El proyecto FITTEST

FITTEST es un proyecto Europeo cuyo objetivo es abordar los desafíos del testeo del IdF. La base para la generación de casos de testo en FITTEST es Testeo de Software Basado en la Búsqueda (« *Search Based Software Testing*») (TSBB) porque esta técnica de testeo se puede adaptar fácilmente a niveles altos de dinamismo que caracterizan las aplicaciones del IdF. Ingredientes claves para un algoritmo de búsqueda son:

- Una función de adecuación que mide cómo la solución actual (es decir caso de testeo) consigue el objetivo del testeo (por ejemplo cobertura, romper un requisito, etc.).
- Operadores que modifican la solución actual para producir nuevos candidatos que se evalúan en la siguiente ronda del algoritmo.

Estos ingredientes se pueden obtener incluso cuando hay mucho dinamismo y poca observabilidad. Por consiguiente, esperamos que la mayoría de las técnicas de testeo que se desarrollaran en el proyecto se aprovechan del enfoque basada en la búsqueda.

Por ejemplo, ***inferencia de modelos*** para el testeo basado en modelos requiere escenarios de ejecución que limiten la aproximación poca definida. Podemos generar estos escenarios

utilizando TSBB con el objetivo de maximizar la cantidad de estados equivalentes que exploramos.

Para el ***aprendizaje de oráculos***, necesitamos data para entrenar y poder inferir especificaciones para el sistema que estamos testeando. Podemos seleccionar ejecuciones aproximadas a través de TSBB, utilizando una función de adecuación diferente para cada propiedad.

Podemos generar ***arboles de clasificación*** de modelos y usar TSBB para generar casos de testeo para estos árboles.

Para el ***testeo concurrente,*** los casos de testeo que producen condiciones de ejecución criticas tienen un valor de adecuación más alta.

Objetivos de cobertura se pueden utilizar para definir una función de adecuación para el testeo de cobertura y de regresión para sistemas ultra-grande.

En el proyecto FITTEST, TSBB es el marco conceptual para unificar todas las técnicas de testeo que serán adaptadas para testear el IdF. Diferentes herramientas serán integradas en el marco que permite el testeo continuo y automatizado de aplicaciones del IdF. Cuantificación de los resultados del proyecto se obtendrá a través de un número de casos de estudio con sistemas reales. Los casos de estudio se ejecutarán utilizando los mejores métodos de la ingeniería del software empírica.

3. Conclusión

Calidad baja de software es, mundialmente, una causa de problemas en la sociedad. Eso es así para todos los sistemas, no solo el Internet del Futuro. Por ejemplo, en 2002, el NIST ha estimado que fallos en el software cuestan $60.000.000.000 para la economía del Estado Unido. Estas estimaciones están basadas en el estado del arte de 2002 que principalmente no era basado en Internet. El mismo informe indica que un tercio de estos costes se pueden eliminar utilizando una infraestructura adecuada para el testeo. Teniendo en cuenta la creciente demanda de sistemas para el Internet que tienen que operar en entornos desafiantes y adaptivos, estas estimaciones se pueden considerar como una cota inferior para los costes que puede generar el testeo inadecuado el Internet del Futuro.

Como manifiestan estos datos, es difícil de exagerar sobre el impacto potencial del proyecto FITTEST.

Para seguir el proyecto, visita http://www.pros.upv.es/fittest.

Business Driven Test Management - una buena práctica en la gestión de las pruebas

Ewout van Driel
Dept. de Consultoría – Software Control & Testing
Sogeti España
Diagonal 67 2ª planta
08018 Barcelona

Resumen

En este artículo se describe la necesidad de la priorización de las pruebas en la validación de la calidad del software. El método Business Driven Test Management (BDTM) [1], elemento de la metodología Tmap®Next [2], da una respuesta contundente a esta necesidad, y pone en el centro de su atención las prioridades y objetivos del cliente final del software desarrollado. Se detallan las actividades del método, y cómo cada una de éstas contribuye a la mejora alineación y entendimiento entre los departamentos de tecnología, y sus usuarios: el negocio.

1. Introducción

Uno de los retos más importantes de la gestión de las pruebas es el hecho que es imposible "probarlo todo". Según argumenta Rex Black [3], para cualquier sistema no trivial, la cantidad de pruebas imaginables es infinito, mientras que la cantidad de pruebas que podemos alcanzar a ejecutar es finita. Por esta razón, la cobertura nuestras pruebas siempre será del 0%.

Necesitamos, por ello, identificar los casos de prueba más relevantes, y que tengan más potencial de descubrir defectos importantes en el software.

Supongamos el caso de definir las pruebas para una sencilla decisión del tipo a continuación:

```
IF (A and B and C) or (D and E)
THEN do ...
ELSE nothing
```

Algoritmo 1. Ejemplo

En el caso de usar la cobertura de múltiples condiciones[2], llegaríamos a definir 32 casos de prueba. Por lo contrario, aplicando la cobertura de condiciones y decisiones modificada, con 6 casos de prueba bastaría. Y en el caso de aplicar la cobertura de condiciones y decisiones, con solo 2 casos de pruebas habríamos acabado.

Es posible que, usando la técnica de condiciones y decisiones modificada, damos una cobertura adecuada al ejemplo. En este caso, nos supone un "ahorro" de 26 casos de prueba, y podríamos dedicar el esfuerzo correspondiente para probar otras funcionalidades más a fondo, o para reducir la duración del proyecto.

¿Quién debe decidir cuántos casos de prueba son suficientes? ¿El usuario, el tester, el desarrollador, el responsable de la seguridad de la información, marketing, recursos humanos o el departamento de operaciones? ¿O todos?

Para determinar la respuesta correcta, que puede depender de cada organización y de cada proyecto, precisamos de prácticas para asegurar que, con la cantidad reducida de pruebas que podemos ejecutar, exista esta "estrategia de pruebas".

Si en su elaboración hemos implicado a todos los interesados relevantes, podemos dar el mayor valor a la organización que invierte en la realización de las pruebas.

2. Conceptos esenciales del BDTM

En la figura a continuación se detalla el proceso de aplicación del Business Driven Test

[2] Los tipos de cobertura de puntos de decisión se explican detalladamente el capítulo 14.3.3 de Tmap®Next

Management a los proyectos de pruebas. Existen 6 actividades claves, realizadas por el responsable de pruebas, para asegurar la máxima alineación de las actividades de pruebas con los objetivos y las prioridades del negocio:

1. Establecer la misión y determinar los objetivos de pruebas.
2. Determinar las clases de riesgo.
3. Determinar la intensidad de las pruebas.
4. Estimar los esfuerzos, planificar el proyecto y consolidar la estrategia.
5. Asignar las técnicas de diseño de pruebas.
6. Dar visibilidad y opciones de control al cliente.

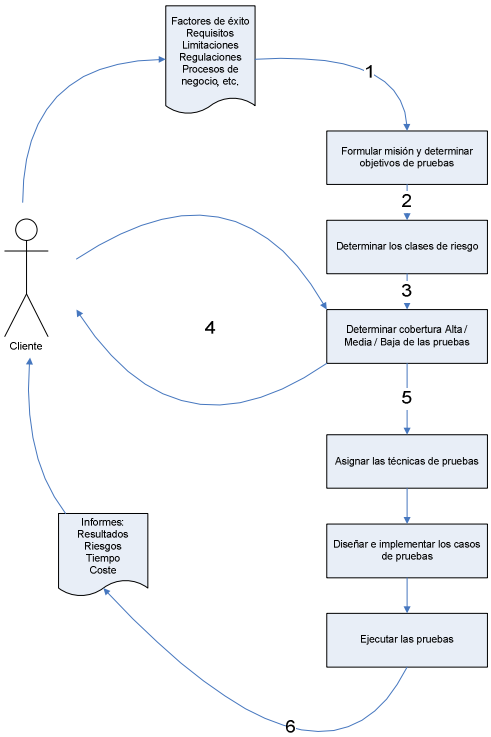


Figura 1. Proceso Business Driven Test Management

2.1. 1. Establecer la misión y determinar los objetivos de pruebas

El objetivo de esta actividad es entender perfectamente el contexto del proyecto, los objetivos del mismo, los requisitos, y posibles normativas o regulaciones a tener en cuenta.

De esta forma, se definen los objetivos de pruebas, que podrían establecerse de la siguiente manera:

- Validar que todos los datos se migran correctamente a la nueva base de datos.
- Validar el correcto funcionamiento de la gestión de pedidos.
- Verificar que el rendimiento del sistema cumple los requisitos establecidos.

Estos puntos son los objetivos principales de nuestro cliente para hacer las pruebas, y a partir de aquí podemos iniciar un análisis más detallado.

2.2. Determinar las clases de riesgo

Para determinar la estrategia de pruebas, nos enfocamos en lo que se debe probar – el objeto de pruebas – y qué se deber probar – las características de calidad. A continuación utilizaremos un ejemplo para las pruebas del tipo funcional, para unos requisitos de un sistema ficticio.

En función de un análisis de riesgos del producto, normalmente partiendo de los requisitos del sistema, se detallan y priorizan los riesgos. Tenemos en cuenta el impacto de un fallo en un requisito o componente, y la probabilidad que este fallo se produzca en una situación operativa. Esta información se incorpora en una tabla de análisis de riesgos según el siguiente ejemplo:

ID	Nombre	Impacto	Prob.
3.1.1	PANTALLA INICIAL	M	B
3.1.2	ALTA DE UN NUEVO CLIENTE	M	A
3.1.3	GESTIÓN DE PEDIDOS	A	M

Tabla 1. Análisis de riesgos de producto

En organizaciones con procesos de calidad de alta madurez, la cantidad de aspectos tenidos en cuenta puede aumentar, así cómo la cantidad de personas implicadas en el análisis de riesgos. De

esta forma, se obtendrá una ponderación del riesgo más detallada y refinada, y aumenta la probabilidad de realizar las actividades a continuación en un mínimo de iteraciones.

2.3. Determinar la intensidad de las pruebas

En función de los riesgos identificados, se establece el nivel de intensidad por cada objeto. Normalmente partimos de un modelo de asignación de intensidad, previamente acordada con el cliente, que tiene la estructura según el siguiente ejemplo. En este caso, para el requisito "Alta de un nuevo cliente", la intensidad de pruebas será Mediana.

Prioridad / Impacto			
	Alto	Medio	Bajo
Alta	A	M	M
Media	A	M	B
Baja	M	B	B

Tabla 2. Tabla de asignación de intensidad de pruebas

La tabla de intensidad, que nos detalla la intensidad de pruebas requerida para el proyecto, en nuestro ejemplo sería la siguiente:

ID	Nombre	Intensidad
3.1.1	PANTALLA INICIAL	B
3.1.2	ALTA DE UN NUEVO CLIENTE	M
3.1.3	GESTIÓN DE PEDIDOS	A

Tabla 3. Intensidad de pruebas por requisito

2.4. Estimación y planificación

En función del tamaño funcional de los objetos a probar, y teniendo en cuenta cómo la estrategia de pruebas influye en el aumento o la disminución del trabajo, se calcula el esfuerzo necesario para realizar las pruebas. El método más formal y exacto para esta tarea el "Test Point Analysis"[3], que a parte del objeto y los objetivos de prueba, tiene en cuenta factores del entorno, la organización, y de la tecnología y herramientas disponibles.

Las actividades se planifican, para que se alineen con las actividades de desarrollo del proyecto, y se presenta al cliente para su aprobación. Como mencionado anteriormente, es común que en este momento ocurren algunas iteraciones de los pasos anteriores, adaptando y refinando la asignación de la intensidad de las pruebas, hasta que se haya alcanzado una cobertura aceptable en términos de mitigación de riesgos, el presupuesto, y el tiempo disponible.

2.5. Asignación de las técnicas de prueba

En el siguiente paso, se asignan las técnicas de pruebas, teniendo en cuenta la intensidad de pruebas a alcanzar, y el tipo de requisito de que se trata. Existen diferentes técnicas para pantallas, componentes de negocio, y para diseñar pruebas a nivel de base de datos, etc.[4], y en este momento se elige el conjunto adecuado, para definir de esta forma la estrategia de pruebas. Siguiendo nuestro ejemplo, esto resulta en la siguiente tabla:

ID	Nombre	Técnica
3.1.1	PANTALLA INICIAL	Casos de uso
3.1.2	ALTA DE UN NUEVO CLIENTE	Semánticas, Combinatoria de datos
3.1.3	GESTIÓN DE PEDIDOS	Sintácticas, Semánticas, Combinatoria de datos, MCDC (cálculo de precios)

Tabla 4. Estrategia de pruebas

El uso de las técnicas de pruebas nos garantizará que cumplimos los objetivos de nuestro cliente, y al mismo tiempo, nos da respuesta a una de las preguntas más complicadas de responder en el ámbito de las pruebas: cuántas

[3] Tmap®Next capítulo 11 detalla el método TPA, y alternativas menos formales para la estimación de esfuerzos.

[4] Véase TMap®Next capítulo 14. También existe una amplia colección de técnicas de prueba de componentes y pruebas no funcionales disponible en http://www.testingstandards.co.uk/ [4]

pruebas son suficientes. Las técnicas son mecanismos reductores, identificando todos los casos relevantes para probar – y descartando los demás – y conjuntamente con el proceso BDTM, nos ayudan a establecer una misión alcanzable para las pruebas.

2.6. Informes

Durante el proceso de pruebas, el responsable de las pruebas reportará al cliente el progreso de las pruebas, y los posibles riesgos que se van produciendo. Gracias a la ejecución de los pasos anteriores, puede relacionar los eventos producidos en la ejecución de las pruebas con las necesidades y prioridades del cliente. Los informes de pruebas contienen información relevante a:

- Resultados: los objetivos de prueba cumplidos
- Riesgos: los riesgos mitigados por los objetivos cumplidos, y los riesgos actuales a causa de por ejemplo defectos en el software
- Tiempo: el cumplimiento de los hitos y planificaciones pasados, y la previsión para el resto del proyecto
- Presupuesto: el esfuerzo gastado, y si el proyecto se mantiene en el presupuesto inicial.

En función de estos 4 parámetros, el responsable de pruebas consigue proporcionar al cliente opciones de control, para realizar ajustes, basada en información real y entendible, y valorar si es preferible asumir más riesgos, o si es imprescindible asignar más recursos o tiempo para lograr sus objetivos.

3. Conclusiones

Con el método BDTM, disponemos de un mecanismo para identificar los objetivos de calidad de nuestro cliente, y priorizar entre ellos. Asimismo, nos permite diseñar una estrategia de pruebas, y aplicar las técnicas de pruebas adecuadas para asegurar la intensidad de pruebas adecuada en función de estos objetivos. Finalmente, nos permite realizar las pruebas, y comunicar los resultados al cliente en "su idioma", para asegurar que en cada momento las actividades de pruebas estén alineadas con sus prioridades.

Referencias

[1] TMapNext, Business Driven Test Management, 2008, ISBN 9789072194930
[2] TMapNext, for result-driven testing, 2006, ISBN 9789072194800
[3] Risk Based Testing: What it is and How You Can Benefit, Rex Black, 23/02/2010, http://www.rbcs-us.com/media/video/Risk%20Based%20Testing_%20What%20It%20Is%20and%20How%20You%20Can%20Benefit.wmv
[4] Module Testing, Testing Standards Working Party, sponsored by the BCS SIGiST, http://www.testingstandards.co.uk/