Los casos de prueba en la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL

Alejandro Álvarez Ayllón, Antonio García Domínguez, Manuel Palomo Duarte e Inmaculada Medina Bulo

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Cádiz
C/ Chile nº 1, Cádiz (España)
{alejandro.alvarez,antonio.garciadominguez,manuel.palomo,inmaculada.medina}@uca.es

Resumen Los lenguajes para la composición de servicios web, como el estándar OASIS WS-BPEL 2.0, abren la posibilidad de programar servicios web a gran escala, facilitando su desarrollo. Es por lo tanto de especial importancia disponer de técnicas para probar dichas composiciones. En trabajos anteriores hemos demostrado la viabilidad de la generación dinámica de invariantes (una técnica usada con éxito en la prueba de lenguajes de programación clásicos) para composiciones de servicios web con WS-BPEL. Estos invariantes se generan a partir de información recopilada durante la ejecución de la composición en un motor real bajo un determinado conjunto de casos de prueba. Por ello, la calidad y utilidad de los invariantes está condicionada por el conjunto de casos de prueba usado para obtenerlos. En este artículo analizamos la evolución de los invariantes de dos composiciones a medida que el conjunto de casos de prueba usado para inferirlos crece.

1. Introducción

Los servicios web (WS) y las arquitecturas orientadas a servicios (SOA) parecen ser una de las claves para entender el futuro de la informática a corto y medio plazo [1]. Sin embargo, a menudo surge la necesidad de ofrecer servicios más potentes basados en otros disponibles, por lo que los lenguajes de composición de servicios, como el estándar OASIS WS-BPEL 2.0 [2], están adquiriendo cada vez una mayor importancia en las SOA [3].

A la hora de probar el software existen dos enfoques principales [4,5]: por un lado, está la prueba de caja negra, que sólo considera las entradas y salidas de un programa. Por otro, está la prueba de caja blanca, que analiza su lógica interna. Esta última produce resultados más precisos, pero requiere disponer del código fuente. Al realizar composiciones WS-BPEL el desarrollador tiene el código fuente de la composición, pero no el de los servicios externos. Es más, puede que dichos servicios sean ofrecidos en diversos momentos del tiempo por varios proveedores usando distintos algoritmos y/o información. Es por ello que el uso de técnicas dinámicas (basadas en la ejecución de código) parece una aproximación idónea para la prueba de caja blanca de composiciones WS-BPEL.

La generación automática de invariantes potenciales [6,7] ha demostrado ser una técnica dinámica exitosa para ayudar en la prueba y mejora de programas escritos en lenguajes imperativos tradicionales. Esta técnica se basa en la recopilación de información de la ejecución de un programa sobre una serie de casos de prueba, que se utiliza posteriormente para derivar invariantes observados en ella. Conviene aclarar en este punto que en este artículo se utilizan los términos «invariante» e «invariante potencial» en el mismo sentido en que se usa en la bibliografía de la materia [6,7,8], refiriéndose «invariante» a cualquier propiedad que es cierta en un determinado punto del programa (como un aserto, pre-condición, invariante de bucle, etc), e «invariante potencial» (o «invariante generado dinámicamente») a cualquier propiedad que se mantiene en una serie de casos de prueba. Por lo tanto, estos invariantes potenciales no tienen que ser necesariamente propiedades que se mantengan en cualquier ejecución del programa. Pero sí tenemos la certeza de que, si son generados a partir de un buen conjunto de casos de prueba, representarán de manera fiable el comportamiento interno del programa.

Tras estudiar la viabilidad de la generación dinámica de invariantes potenciales en WS-BPEL [9,10,11] construimos el sistema libre Takuan [12]. Este sistema genera dinámicamente invariantes potenciales a partir de información recopilada en diversas ejecuciones de una composición WS-BPEL sobre un motor real. Para ello toma como entrada los ficheros de definición del proceso y un conjunto de casos de prueba, y da como salida un conjunto de invariantes que se cumplen en diversos puntos del programa para todos los casos de prueba.

Sin embargo, como hemos comentado, la validez de los invariantes obtenidos por Takuan depende del conjunto de casos de prueba usados para generarlos. Por ello en este artículo presentamos un estudio de la relación entre varios conjuntos con distinto número de casos de prueba aleatorios y los invariantes resultantes de su ejecución en dos composiciones.

El resto de este artículo se organiza del siguiente modo: en el siguiente apartado se presenta el lenguaje WS-BPEL y se comentan las tecnologías relacionadas con él. En la tercera sección se explica el proceso de generación dinámica de invariantes en WS-BPEL usando Takuan. A continuación analizamos los resultados obtenidos en las dos composiciones con distintos conjuntos de casos de prueba. Posteriormente se exponen los principales trabajos relacionados. Por último, el artículo finaliza presentando conclusiones, así como unas líneas generales de nuestro trabajo próximo.

2. WS-BPEL

La necesidad de componer diversos WS para ofrecer servicios más complejos y potentes fue detectada y satisfecha por las grandes empresas del sector TIC con la especificación no estándar BPEL4WS, que fue enviada a OASIS para su estudio en 2003 [13]. Poco después, OASIS creó el *WS Business Process Execution Language Technical Committee* para trabajar en su estandarización, publicando la primera versión estandarizada, WS-BPEL 2.0, en 2007 [2].

La estandarización de WS-BPEL ha sido un gran hito para su adopción en las principales herramientas SOA, siendo en la actualidad una de las principales características de interoperabilidad en la mayoría de ellas [14]. De este modo, se pueden componer servicios aprovechando las ventajas de este lenguaje de programación a gran escala: concurrencia, compensación o invocación de WS de diversos proveedores descubiertos dinámicamente atendiendo a diferentes criterios (como coste, fiabilidad o tiempo de respuesta).

WS-BPEL es un lenguaje basado en XML, que usa XML Schema como su sistema de tipos. WS-BPEL describe la lógica de composición de servicios con etiquetas XML que especifican actividades concretas, como asignaciones, bucles o envío de mensajes. Una de sus principales ventajas es su independencia de la implementación y plataforma usada tanto por el proveedor de servicios como por el usuario de la composición.

Así mismo, hay otras tecnologías (algunas ya estandarizadas) que pueden extender WS-BPEL. A menudo se denominan la pila de WS, *WS-Stack* [15]. Una de las más interesantes es UDDI, que permite mantener repositorios de especificaciones WSDL, facilitando que se puedan descubrir e invocar servicios dinámicamente a través de ellos. UDDI 3.0 es un estándar de OASIS [16].

3. Generación de Invariantes

Tras analizar el estado del arte observamos que hasta la fecha se ha investigado poco sobre la aplicación de técnicas de prueba de caja blanca directamente sobre código de composiciones WS-BPEL ejecutado en un entorno real. Las principales propuestas [17] crean un modelo de simulación en un entorno especializado para pruebas.

Sin embargo la simulación de un motor WS-BPEL es algo complejo, dado que hay una gran cantidad de características nada triviales que implementar. En caso de que alguna de estas características no se implementara correctamente, la composición no se estaría probando adecuadamente. Por ello, consideramos que es un proceso propenso a errores, pues no se basa en la ejecución del código WS-BPEL en un entorno real (es decir, un motor WS-BPEL que invoque a servicios reales).

Por contra, Takuan genera dinámicamente invariantes potenciales a partir de una serie de ejecuciones de la composición WS-BPEL en un motor real. Por lo tanto, creemos que se adapta mejor a la naturaleza del lenguaje.

3.1. Uso de Invariantes

Un invariante es una propiedad que es cierta en un determinado punto de un programa (antes o después de una determinada instrucción). Ejemplos clásicos son las pre-condiciones y post-condiciones de funciones, que son propiedades que siempre se cumplen al inicio y al final de una serie de instrucciones, respectivamente. Igualmente, los invariantes de bucle expresan propiedades que se

mantienen antes de cada una de sus iteraciones, incluida la primera, y tras la última de ellas.

Podemos ilustrar estos conceptos con un sencillo ejemplo. Supongamos que deseamos sumar todos los enteros desde 1 hasta un determinado entero positivo n inclusive. Podríamos definir un sencillo algoritmo que lo hiciera, como el siguiente:

1. $r \leftarrow 0$
2. Desde $i \leftarrow 1$ hasta n:
   $r \leftarrow i + r$
3. Devolver r

Este algoritmo tiene la pre-condición $n > 0$ en el paso 1, dado que n es positivo por definición. Si nos fijamos en el paso 2, podemos observar que el invariante de bucle $r = \sum_{j=0}^{i-1} j$ se mantiene antes de cada iteración y tras la última. Dado que en el paso 3 ya hemos salido del bucle, tendremos que $i = n+1$. Sustituyendo en el invariante de bucle anterior tendremos la post-condición del paso 3 y, por extensión, de todo el algoritmo: $r = \sum_{j=0}^{n} j$. A partir de esta post-condición podemos afirmar que el algoritmo realmente hace lo que se espera de él.

Los invariantes se han usado con éxito en demostraciones manuales de corrección de algoritmos, de manera similar al ejemplo anterior. Sin embargo, su generación puede automatizarse. De hecho, la generación automática de invariantes ha demostrado ser una técnica eficaz para ayudar en la prueba de caja blanca y la mejora de programas escritos en lenguajes de programación estructurados y orientados a objetos [6]. Los invariantes generados a partir de un programa pueden usarse de diversas formas para mejorarlo:

Depuración de errores Un invariante inesperado puede hacernos ver un fallo en el código que de otra forma podría haber pasado desapercibido. Esto incluye, por ejemplo, llamadas a funciones con valores no válidos en algún parámetro o fallos en las condiciones de bucles.

Asistencia al ampliar un programa El uso de invariantes puede ayudar en la ampliación de un programa. Tras comprobar qué invariantes deben mantenerse y cuáles no entre dos versiones de un programa, podrían compararse los resultados esperados con los realmente obtenidos. De este modo, cualquier diferencia indicaría que se ha introducido algún error en el nuevo código.

Documentación Los invariantes más destacados pueden incluirse en la documentación del código del programa, así los desarrolladores que tengan acceso a él podrán consultarlos en su trabajo.

Verificación Se puede comparar la especificación del programa con los invariantes obtenidos para ver si esta se cumple.

Además, un invariante potencial erróneo que se haya generado dinámicamente, como se observará en el siguiente apartado, puede indicar una deficiencia en el conjunto de casos de prueba usado para inferirlo y ayudar a mejorarlo.

3.2. Generación Automática de Invariantes

Básicamente, existen dos tipos de generadores automáticos de invariantes: estáticos y dinámicos.

Los generadores estáticos de invariantes [18,19] son los más usados: deducen los invariantes de un programa estáticamente, es decir, sin ejecutar su código. Para ello analizan su código fuente, principalmente los datos y el flujo de control, lo que los hace dependientes del lenguaje concreto. Los invariantes generados de esta forma son siempre ciertos. Sin embargo, su número y alcance es reducido, debido a la limitaciones inherentes al mecanismo formal que analiza el código, sobre todo al enfrentarse a lenguajes poco convencionales como WS-BPEL.

Por el contrario, los generadores dinámicos de invariantes potenciales [6,7] informan de posibles invariantes de un programa observados en la información recopilada en varias ejecuciones de él. En cada ejecución, parte de la información de la traza del programa se almacena en registros para su posterior análisis. Los generadores incluyen un mecanismo formal que analiza la información de los registros, fundamentalmente los valores almacenados por las variables en diversos puntos del programa, como las entradas y salidas de funciones y bucles, e infiere los invariantes observados.

Esta forma de generar invariantes no es una técnica estática, sino dinámica, pues está basada en la ejecución del código sobre una serie de casos de prueba. De este modo, la obtención de invariantes erróneos no implica necesariamente fallos en el programa, sino que puede venir originada por el empleo de un conjunto incompleto de casos de prueba.

Por ejemplo, si un programa recibe como entrada un entero, x, pero sólo le proporcionamos casos de prueba en los que reciba valores naturales, probablemente obtengamos el invariante falso $x \geq 0$ u otro que lo implique en algún punto del programa. Dicho invariante indicaría que es necesaria una inspección y mejora del conjunto de casos de prueba, incluyendo casos en los que x reciba un valor negativo, para que esto no ocurra (lo que se suele denominar falsificación del invariante potencial).

3.3. Generación Dinámica de Invariantes en Composiciones WS-BPEL

Como se ha comentado anteriormente, los invariantes generados dinámicamente a partir de un conjunto de casos de prueba adecuado son una buena muestra de la lógica interna de un programa. Por ello son especialmente adecuados para WS-BPEL, pues señalarán los aspectos más delicados de la composición como determinados comportamientos de los servicios externos, compensaciones de errores, manejo de fallos, etc.

Hay que tener en cuenta que, debido a la naturaleza dinámica del proceso, cuantos más registros proporcionemos al generador mejores resultados producirá por lo general. Es posible que en las primeras ejecuciones se obtengan invariantes aparentemente falsos. Estos pueden ser debidos a fallos en el código del

programa o a deficiencias en el conjunto de casos de prueba usado para generarlos. Para comprobarlo bastará con incorporar casos de prueba específicos para ello en posteriores ejecuciones, mejorando de esta forma el conjunto de casos de prueba original, y observar si se siguen infiriendo o no dichos invariantes.

Otra ventaja a destacar es que toda la información de los registros se recopila directamente de ejecuciones del código de la composición, sin usar ningún tipo de lenguaje intermedio. De este modo, se evitan errores que podrían producirse en la traducción del código WS-BPEL o el modelado del motor WS-BPEL y los servicios invocados en un entorno de simulación.

Conviene aclarar en este punto que Takuan permite que la composición WS-BPEL invoque a servicios reales u opcionalmente puede hacer que invoque a servicios simulados que el mismo Takuan controle. Esta opción se ha incluido porque, por lo general, no podemos suponer que todos los servicios externos vayan a estar disponibles a la hora de realizar las ejecuciones. Esto puede deberse a varios motivos: limitaciones en su uso, restricciones de recursos, costes, etc. Incluso puede ser que simplemente no se desee probar el comportamiento de una composición con la respuesta que dé un servicio en el momento de su ejecución, sino con un valor predeterminado concreto que defina un escenario del tipo «¿qué pasaría si el servicio x respondiera el valor y?». Por lo tanto, en caso de que el usuario desee hacer uso de dicha característica, él será responsable de ampliar los casos de prueba incluyendo las respuestas que desee que proporcionen los servicios o indicando que den un fallo concreto si así lo desea.

3.4. Takuan

Takuan es un generador dinámico de invariantes potenciales para WS-BPEL que se distribuye bajo licencia GPL, y está disponible para descarga gratuita en su web oficial [20]. En dicha web proporcionamos tanto el código fuente como una imagen de máquina virtual VirtualBox para disponer fácilmente en cualquier equipo de un sistema completo con la última versión de Takuan configurada. Además, para facilitar su uso hemos desarrollado un plugin para NetBeans (uno de los entornos de desarrollo WS-BPEL más populares) que permite el uso desde un asistente con interfaz gráfica [21].

La arquitectura interna de Takuan fue presentada y analizada en [9,10,11]. Se basa en modificaciones, extensiones y posterior interconexiones de los siguientes sistemas libres:

ActiveBPEL es un motor que respeta el estándar WS-BPEL 2.0. Comparado con otros motores es bastante ligero, lo que reduce el tiempo necesario para ejecutar casos de prueba. La empresa ActiveVOS [22] se encarga de su mantenimiento y ofrece servicios y productos empresariales basados en él.

BPELUnit es una biblioteca de prueba unitaria WS-BPEL [23] que puede usar cualquier motor que implemente WS-BPEL 2.0. Entre sus principales características está el uso de ficheros XML para describir los casos de prueba a ejecutar y la posibilidad de sustituir servicios externos con otros servicios que los simulen desarrollando el comportamiento indicado en la especificación proporcionada por el usuario.

Daikon es un generador dinámico de invariantes potenciales [7] que se ha usado con éxito para probar programas escritos en Java, C, C++ y Perl. Es muy configurable y maduro, y ofrece su salida en diversos formatos. Uno de ellos es el formato de entrada de *Simplify* [24], un demostrador automático de teoremas que permite eliminar invariantes redundantes.

Takuan se ha aplicado con buenos resultados a diversas composiciones, como el ejemplo clásico del préstamo bancario [12] o una composición de meta-búsqueda de información [25]. Tras dichas pruebas se observaron ciertas limitaciones en sus prestaciones, por lo que se le implementaron una serie de mejoras para reducir su consumo de recursos (disminuyendo el uso de CPU y memoria, lo que permite analizar composiciones más grandes) y mejorar su salida, evitando determinados invariantes redundantes no detectados anteriormente [26].

4. Experimentos realizados

Para comprobar la influencia del conjunto de casos de prueba en la generación de invariantes hemos realizado pruebas con dos composiciones de ejemplo. La primera de ellas es una modificación del ejemplo del préstamo bancario incluida en el estándar WS-BPEL [2] que ya analizamos en [12]. Dicha modificación usa condicionales en vez de ejecución paralela de órdenes para que Takuan genere más invariantes.

La segunda composición es un *mercado de compraventa* (en inglés Market-Place) que se ofrece como ejemplo en la web de ActiveVOS [22]. Básicamente su comportamiento es el siguiente: al recibir una petición de venta de un producto espera a que le llegue una oferta por él (o viceversa). Si la cantidad ofrecida en la oferta de compra es mayor o igual que lo pedido para la venta realiza la transacción, y en otro caso informa de que no se ha llegado a un acuerdo.

En ambos casos se incluyeron en los casos de prueba las respuestas que cada servicio debía dar al ser invocado, para que Takuan los simulara. En el caso del préstamo se pedían préstamos por cantidades entre 0 y 49,900 dólares en intervalos de 100 unidades (el umbral de la composición para llamar al asesor es de 10,000 dólares). Las respuestas del servicio de *asesor* eran riesgo alto o bajo al 50 % y las del aprobador aceptación o rechazo también al 50 %

En el mercado de compraventa los casos de prueba consistían en dos clientes que interactuaban por el mismo artículo con cantidades comprendidas entre 0 y 999 unidades monetarias. Aleatoriamente se les estableció un retraso de 0, 1 o 2 segundos independiente a cada uno, porque la creación de instancias se puede realizar bien por la recepción de un mensaje de petición de venta de producto, bien por la recepción de una oferta para un producto (al ejecutarse su recepción en paralelo dentro de una estructura *flow*).

4.1. Estructura de los experimentos

Los experimentos se realizaron con conjuntos de casos prueba aleatorios incrementales generados con una distribución uniforme, como se propone en [6].

Para cada composición se generó un lote inicial de cinco casos de prueba aleatorios. Se ejecutaron sobre Takuan y se almacenaron los invariantes obtenidos. Posteriormente se incrementó el conjunto con otros cinco casos hasta llegar a diez y se volvieron a almacenar los invariantes resultantes de su ejecución. Se repitió el mismo proceso con conjuntos de veinte, cincuenta, cien y doscientos casos de prueba aleatorios. Los invariantes resultantes de la ejecución del conjunto más amplio se tomaron como referencia sobre la que comparar los resultados anteriores, pues estimamos que para composiciones relativamente sencillas como las que se han usado son una cantidad de casos de prueba suficiente para lograr unos invariantes significativos.

Cada columna de la comparativa realizada muestra la información relativa a un conjunto de casos de prueba. En la parte superior se indican el número de casos de prueba que compone el conjunto. A continuación está el número total de invariantes. Después se comparan los invariantes de referencia con los obtenidos, y se indica el número de diferencias observadas. Esta comparación se realiza primero sobre el conjunto de invariantes etiquetado por Daikon como "interesantes" y después sobre el conjunto completo. Daikon (el generador de invariantes que Takuan usa internamente) clasifica los invariantes en "interesantes" y "no interesantes" con una heurística interna. Estos segundos son aquellos que hacen referencia a valores concretos de variables con listas de valores o intervalos y que posiblemente sean debidos a limitaciones de la muestra proporcionada. Por ejemplo, si en el caso de préstamo se solicitan préstamos por valor de 1000, 1200, 1400, 2800 y 3000 dólares es posible que Daikon informe que el valor de la cantidad solicitada está siempre entre 1000 y 3000, pero entiende que es probable que si tuviera más casos de prueba dicho invariante sería falsificado.

Tras comparar los invariantes se cuentan los puntos del programa donde se encuentran los invariantes distintos (por defecto Takuan considera como punto del programa donde generar invariantes justo antes y después de cada elemento *sequence* de la composición) y los tiempo empleados en la ejecución. El tiempo total empleado es el tiempo desde que se invoca Takuan hasta que se obtiene su respuesta final. El tiempo dedicado a ejecutar código del programa es el tiempo de CPU de usuario y el dedicado a llamadas al sistema el tiempo de CPU del sistema. La diferencia de tiempo entre el tiempo total y la suma de los dos tiempos de CPU es dedicado a entrada y salida, espera de mensajes de servicios web, etc.

Las pruebas se realizaron en una máquina Intel Pentium 4 a 1.6 GHz con 512 MB de memoria RAM. En dicha arquitectura se ejecutó la máquina virtual de Takuan sobre una instalación básica de OpenSUSE 11.1 con su núcleo estándar. Durante la ejecución no se ejecutó ningún otro proceso que pudiera cargar significativamente la CPU, memoria principal o canales de comunicación con disco. Aunque la máquina no es demasiado potente y la ejecución sobre una maquina virtual alarga un poco el tiempo de ejecución, no es nuestra intención estudiar los tiempos de ejecución de Takuan, pues han sido tratados detalladamente en [26]. Pero sí creemos que los datos temporales proporcionados tienen

interés comparativo entre ellos para medir el esfuerzo que supone la ejecución de Takuan con conjuntos de casos de pruebas de distinto tamaño.

4.2. Resultados obtenidos

Los resultados de las pruebas en la composición del préstamo bancario se pueden observar en el cuadro 1, mientras que los del mercado de compraventa están en el cuadro 2. Como la segunda composición convergió muy pronto se tomó como conjunto de referencia el de 100 casos de prueba en vez de el de 200.

Cuadro 1. Evolución de la composición del préstamo bancario

Número de casos de prueba	5	10	20	50	100	200
Total de invariantes	106	164	160	163	167	167
Invariantes distintos interesantes	116	45	18	4	0	
Invariantes distintos (incl. no interesantes)	124	62	38	14	10	
Total puntos de programa (TPP)	8	10	10	10	10	10
TPP con invariantes interesantes distintos	10	7	7	4	0	
TPP con invariantes (incl. no interesantes) distintos	10	8	8	6	4	
Tiempo total empleado	22s	57s	47s	1m14s	2m07s	3m00s
Tiempo de CPU de usuario	6s	8s	8s	12s	19s	24s
Tiempo de CPU de sistema	10s	14s	15s	21s	26s	46s

Cuadro 2. Evolución de la composición del mercado de compraventa

Número de casos de prueba	5	10	20	50	100
Total de invariantes	8	8	6	6	6
Invariantes distintos interesantes	6	2	0	0	
Invariantes distintos (incl. no interesantes)	14	14	8	8	
Total puntos de programa (TPP)	3	3	3	3	3
TPP con invariantes interesantes distintos	2	2	0	0	
TPP con invariantes (incl. no interesantes) distintos	2	2	2	2	
Tiempo total empleado	20s	32s	54s	2m09s	3m30s
Tiempo de CPU de usuario	3s	3s	4s	8s	12s
Tiempo de CPU de sistema	8s	10s	12s	21s	32s

La ejecución de Takuan se hizo con sus opciones por defecto, lo que implica que todos los campos y propiedades de todas las variables que se usen en cada elemento *sequence* de la composición se inspeccionen. En concreto, se inspeccionaron 120 elementos en el préstamo y 8 en la compraventa. Además, el resultado se ha analizado, "tal cual" sin aplicarle el simplificador lógico *Simplify*.

Si nos fijamos, por ejemplo, en la columna de cinco casos de prueba del préstamo vemos que estos generan un total de 106 invariantes. Sin embargo en la tercera fila observamos que al compararlos con los 167 resultantes de aplicar el conjunto de referencia (de 200 casos de prueba) se obtienen 124 diferencias entre ellos, una cantidad muy alta. En cuarta fila se informa de que con dicho conjunto sólo se han generado invariantes en 8 puntos del programa, mientras las siguientes dos filas indican que en los 10 puntos del programa donde el conjunto de referencia ha generado invariantes hay diferencias en el resultado. Hay que resaltar que es normal que se falsifiquen un alto número de los invariantes generados por un conjunto tan pequeño de casos de prueba, pues las trazas de ejecución difícilmente podrán reflejar adecuadamente la complejidad interna de la composición (los distintos valores que hacen ejecutar las ramas de una instrucción condicional, etc). Sin embargo a medida que el número de casos aumenta la diferencia respecto con el resultado del conjunto de referencia disminuye.

Si observamos los invariantes vemos como en ambas composiciones al incrementar el conjunto de entrada con más casos de prueba, los resultados van convergiendo hacia un conjunto estable. Esto se cumple tanto los invariantes que Daikon marca como interesantes como los "no interesantes". También se ve que la composición del mercado de compraventa converge mucho antes que la del préstamo (en concreto al añadir en el tercer lote diez ejemplos más sobre a los diez existentes en la prueba anterior), lo que nos indica que tiene una complejidad interna menor. Por su parte, en la composición del préstamo esta convergencia da un fuerte incremento también con veinte casos de prueba, pero se sigue reduciendo paulatinamente hasta los cien. Lo indicado para los invariantes es aplicable a los puntos del programa en los que se han observado dichos invariantes distintos.

Por lo tanto, podemos afirmar que, como suele suceder en la mayoría de técnicas de prueba, a mayor número de casos de prueba, mejores son los invariantes obtenidos. También hay que señalar que la generación de casos de prueba aleatorios es sencilla y poco costosa. Sin embargo es necesario tener en cuenta el tiempo necesario para obtenerlos. En composiciones relativamente simples como las usadas, los tiempos son aceptables. Pero como se apuntaba en artículos anteriores [12,26], este puede crecer significativamente en otras composiciones más complejas (en especial al aumentar el número de instrucciones o la cantidad de variables a inspeccionar).

Además en ambos experimentos se observa un determinado umbral a partir del cual la mejora en la salida es marginal (o incluso nulo), por lo que no merece la pena añadir más casos de prueba a partir de él. En la composición del préstamo bancario dicho umbral se puede establecer en 20 casos de prueba, mientras que para el préstamo está entre 50 y 100.

Por otro lado hay que aclarar que al observar los resultados puede resultar extraño que por lo general el tiempo necesario para ejecutar la composición del mercado de compraventa sea mayor que el del préstamo (cuando esta segunda es más compleja en código y cantidad de variables). Eso es debido a que, como se comentó anteriormente, los casos de prueba usados incluyen distintos retrasos

en el envío de cada mensaje para probar la composición adecuadamente (pues el primer mensaje que llegue será el que cree la nueva instancia de la composición). Este retraso provoca tiempos de espera en la ejecución de la composición. De hecho, se puede ver como el tiempo de CPU (tanto de usuario como de sistema) es menor para el mercado de compraventa que para el préstamo en la práctica totalidad de los casos.

Resumiendo podemos afirmar que incrementar el número de casos de prueba de una composición aleatoriamente es, por lo general, una forma segura de mejorar y confirmar los invariantes generados por Takuan. Sin embargo, debemos ser prudentes y tener en cuenta el coste temporal que tienen dichas pruebas, pues existe un umbral de eficiencia a partir del cual los nuevos casos mejoran poco o nada la salida.

5. Trabajos Relacionados

A continuación se comentan otros trabajos similares, relacionados con la generación dinámica de invariantes con Daikon y las pruebas de composiciones de servicios WS-BPEL.

En [27] se presenta una propuesta interesante para la generación dinámica de invariantes potenciales con Daikon como método para medir la calidad de servicio de un WS. En concreto, recopila los datos resultantes de diversas invocaciones a un WS desde una composición WS-BPEL y las respuestas obtenidas para evaluar su contrato de calidad de servicio (en inglés *Service Level Agreement*). Por lo tanto, este enfoque está limitado a pruebas de caja negra de servicios concretos. Mientras que, por el contrario, Takuan ayuda a la prueba de caja blanca de composiciones WS-BPEL mediante la generación de invariantes potenciales referentes a su lógica interna.

Podemos observar otro uso de Daikon en [28]. En dicho trabajo se infieren invariantes durante la prueba de una infraestructura automática de comunicaciones. De ese modo, durante su ejecución, un sistema podría detectar errores en su funcionamiento y activar los mecanismos necesarios para gestionar dicha situación no contemplada inicialmente. De modo similar se podría usar Takuan para generar invariantes de una composición durante su desarrollo y posteriormente, durante su ejecución real, detectar posible errores en su funcionamiento a partir de sus registros de ejecución y actuar en caso de observar problemas en ella.

Dynamo [29] es un sistema que usa un *proxy* para monitorizar si una composición WS-BPEL respeta ciertas restricciones durante su ejecución. Consideramos que sería una forma de comprobar si los invariantes que se espera que una composición cumpla tras la fase de prueba realmente no se violen durante su ejecución en su entorno real.

6. Conclusiones y Trabajo Futuro

Hoy en día parece claro que el futuro de la informática pasa ineludiblemente por los WS. La necesidad de componerlos para proporcionar servicios más complejos con independencia de las plataformas usadas ha sido solucionada con el estándar WS-BPEL 2.0 de OASIS.

En este artículo hemos estudiado la relación entre la generación dinámica de invariantes potenciales y el tamaño del conjunto de casos de prueba aleatorios usado para inferirlos. Para ello hemos usado Takuan, un sistema que genera dinámicamente invariantes potenciales a partir de una composición WS-BPEL. Takuan toma como entrada los ficheros de definición de la composición y un conjunto de casos de prueba que ejecuta en un motor real.

El estudio que hemos presentado permite afirmar la importancia de utilizar un conjunto de casos de prueba lo suficientemente amplio para asegurar la calidad y estabilidad de los invariantes generados por Takuan. Como sucede por lo general en cualquier otra técnica de prueba, a mayor número de casos de prueba usados mejor son los resultados, pero igualmente también aumenta su coste computacional. Por ello se han usado varios conjuntos con distintas cantidades de casos de prueba aleatorios y se han comparado sus resultados con los producidos por un conjunto lo suficientemente grande como para tomar sus invariantes como referencia.

Hay que señalar que aunque la producción de casos de prueba aleatorios es poco costosa, la generación dinámica de invariantes con un conjunto grande de casos de prueba tiene un coste que es necesario estimar a priori. Además se ha observa la existencia de un determinado umbral a partir del cual la mejora en la salida es marginal al incrementar los casos de prueba, por lo que no merece la pena añadir más casos de prueba una vez alcanzado.

Nuestro siguiente trabajo será comparar los resultados de la prueba de caja blanca para WS-BPEL usando invariantes generados por Takuan con los resultados proporcionados por otras técnicas de prueba.

Agradecimientos

Este trabajo ha sido financiado por el Programa Nacional de I+D+I del Ministerio de Educación y Ciencia y fondos FEDER mediante el proyecto SOAQSim (TIN2007-67843-C06-04)

Referencias

1. Heffner, R., Fulton, L.: Topic overview: Service-oriented architecture. Forrester Research, Inc. (Junio 2007)
2. OASIS: WS-BPEL 2.0 standard. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html` (Abril 2007)
3. Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarana, S.: The next step in Web Services. Communications of the ACM **46**(10) (2003) 29–34

4. Bertolino, A., Marchetti, E.: A brief essay on software testing. En Thayer, R.H., Christensen, M., eds.: Software Engineering, The Development Process. third ed. Wiley-IEEE Computer Society Press (2005)
5. Myers, G.J.: The Art of Software Testing. segunda ed. John Wiley & Sons (2004)
6. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2) (Febrero 2001) 99–123
7. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1-3) (Diciembre 2007) 35–45
8. Gupta, N.: Generating test data for dynamically discovering likely program invariants. En: ICSE 2003 Workshop on Dynamic Analysis (WODA 2003). (2003)
9. Palomo-Duarte, M., García-Domínguez, A., Medina-Bulo, I.: An architecture for dynamic invariant generation in WS-BPEL web service compositions. En: Proceedings of ICE-B 2008 - International Conference on e-Business, Oporto, Portugal, INSTICC Press (Julio 2008)
10. García Domínguez, A., Palomo Duarte, M., Medina Bulo, I.: Framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. En: Actas de Talleres de Ingeniería del Software y Bases de Datos, Gijón, España, SISTEDES (Octubre 2008) 1–10
11. García Domínguez, A., Palomo Duarte, M., Medina Bulo, I.: Implementación de un framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. En: Actas de las IV Jornadas Científico-Técnicas en Servicios Web y SOA, Sevilla, España (Octubre 2008) 91–96
12. Palomo-Duarte, M., García-Domínguez, A., Medina-Bulo, I.: Takuan: A dynamic invariant generation system for WS-BPEL compositions. En: Proceedings of the 6th IEEE European Conference on Web Services, Dublín, Irlanda (Noviembre 2008)
13. OASIS: OASIS members form web services business process execution language (WSBPEL) technical committee. http://www.oasis-open.org/news/oasis_news_04_29_03.php (Abril 2003)
14. Domínguez Jiménez, J.J., Estero Botaro, A., Medina Bulo, I., Palomo Duarte, M., Palomo Lozano, F.: El reto de los servicios web para el software libre. En: Proceedings of the FLOSS International Conference 2007, Jerez de la Frontera, España, Servicio de Publicaciones de la Universidad de Cádiz (Marzo 2007) 117–132
15. Papazoglou, M.: Web services technologies and standards. Computing Surveys (submitted) (2007)
16. OASIS: UDDI standard 3.0. `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm` (Febrero 2005)
17. Bucchiarone, A., Melgratti, H., Severoni, F.: Testing service composition. En: Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07). (2007)
18. Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Computer Science **173**(1) (1997) 49–87
19. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. En Jr., W.A.H., Somenzi, F., eds.: CAV. Volumen 2725 of Lecture Notes in Computer Science., Springer (2003) 420–432
20. Grupo SPI&FM: Web oficial de Takuan. `http://neptuno.uca.es/~takuan` (2009)
21. García Domínguez, A., Palomo Duarte, M., Álvarez Ayllón, A., Medina Bulo, I.: Takuan: generación dinámica de invariantes en composiciones de servicios web con

WS-BPEL. En: XIV Jornadas Ingeniería del Software y Bases de Datos (a la espera de publicación). (2009)
22. ActiveVOS: ActiveBPEL WS-BPEL and BPEL4WS engine. `http://sourceforge.net/projects/activebpel` (Febrero 2008)
23. Mayer, P., Lübke, D.: Towards a BPEL unit testing framework. En: TAV-WEB'06: Proceedings of the 2006 workshop on Testing, Analysis, and Verification of Web Services and Applications, New York, NY, EEUU, ACM (2006) 33–42
24. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473
25. Palomo-Duarte, M., García-Domínguez, A., Medina-Bulo, I.: Improving Takuan to analyze a meta-search engine WS-BPEL composition. En: Proceedings of the 4th IEEE International Symposium on Service-Oriented System Engineering, Jhongli, Taiwan (Diciembre 2008)
26. Palomo-Duarte, M., García-Domínguez, A., Medina-Bulo, I.: Enhancing WS-BPEL dynamic invariant generation using XML Schema and XPath Information. En: ICWE'09: Proceedings of the 9th International Conference on Web Engineering. Volumen 5648 of Lecture Notes in Computer Science., Springer (2009) 469–472
27. SeCSE: Testing method definition V4 (final). http://www.secse-project.eu/wp-content/uploads/a1d34-testing-method-definition-v4-final.pdf (Marzo 2008) Service Centric System Engineering.
28. Denaro, G., Mariani, L., Pezzè, M., Tosi, D.: Adaptive runtime verification for autonomic communication infrastructures. En: Workshop on Autonomic Communications and Computing, Taormina, Italia (Junio 2005)
29. Baresi, L., Guinea, S.: Dynamo: Dynamic monitoring of WS-BPEL processes. En Benatallah, B., Casati, F., Traverso, P., eds.: ICSOC. Volumen 3826 of Lecture Notes in Computer Science., Springer (2005) 478–483