Implementación de un framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL

Antonio García Domínguez, Manuel Palomo Duarte e Inmaculada Medina Bulo

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

Resumen Los servicios web están cambiando la informática actual y serán clave para entenderla en un futuro próximo. En concreto, los lenguajes para la composición de servicios, como el estándar OASIS WS-BPEL 2.0, abren la posibilidad de programarlos a gran escala, facilitando su adopción. Sin embargo, este lenguaje presenta un reto para la realización de pruebas de caja blanca, debido a la inclusión de instrucciones específicas para manejar concurrencia, compensación, errores, o descubrimiento e invocación dinámicos de servicios. Por otro lado, la generación automática de invariantes ha demostrado ser una técnica eficaz para ayudar en la prueba y mejora de programas escritos en lenguajes imperativos tradicionales y creemos que también lo sería para WS-BPEL. En este artículo presentamos un framework para generar invariantes potenciales dinámicamente a partir de varios registros de ejecución de una composición de servicios web WS-BPEL. Describimos nuestra experiencia durante su implementación y sus resultados preliminares basados en el ejemplo del préstamo bancario.

Palabras clave Servicios web, Composición de servicios, WS-BPEL, Prueba de caja blanca, Generación dinámica de invariantes.

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]. Entre ellas destacada el estándar OASIS WS-BPEL 2.0 [2], que permite ofrecer servicios más potentes basados en otros disponibles.

Pero a la hora de aplicar técnicas de prueba de caja blanca [3] a composiciones WS-BPEL, estas representan un reto, debido a la inclusión de instrucciones específicas para el manejo de WS, como manejo de compensación, fallos, etc [4].

La generación automática de invariantes potenciales [5] ha demostrado ser una técnica eficaz para ayudar en la prueba y mejora de programas escritos en lenguajes imperativos tradicionales. 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 [5], 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» a cualquier propiedad que se mantiene en los distintos casos de prueba ejecutados.

Este artículo presentamos un framework que 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.

El resto de este artículo se organiza del siguiente modo: el segundo apartado justifica cómo la generación dinámica de invariantes puede ser un técnica apropiada para ayudar en la realización de pruebas de caja blanca de composiciones de WS. Después, el tercero presenta nuestro framework. En la sección cuarta pasamos a comentar trabajos relacionados. Por último, el artículo finaliza presentando conclusiones, así como unas líneas generales de nuestro trabajo próximo.

2 Invariantes y Composiciones WS-BPEL

La prueba de composiciones de WS es uno de los retos más importantes para su implantación generalizada a medio plazo. Su naturaleza dinámica dificulta la aplicación directa de técnicas clásicas de prueba, pues hay que gestionar aspectos poco comunes en otros lenguajes, como el descubrimiento e invocación en tiempo de ejecución de servicios y la compensación.

Hasta la fecha se ha investigado poco sobre la aplicación de técnicas de prueba de caja blanca directamente sobre el código de composiciones WS-BPEL ejecutado en un entorno real. Las principales propuestas [4] crean un modelo de simulación en un entorno especializado para pruebas.

Pero 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, dado que 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 como permite nuestro framework).

2.1 Uso de Invariantes

Los invariantes se han usado con éxito en demostraciones manuales de corrección de algoritmos. 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 [5].

Los invariantes generados a partir de un programa pueden usarse de diversas formas para mejorarlo. Por ejemplo, un invariante inesperado puede hacernos ver un fallo en el código que de otra forma podría haber pasado desapercibido. También, a la hora de ampliar un programa, se pueden comprobar qué invariantes deben mantenerse y cuáles

no entre dos versiones de él (por lo que cualquier diferencia indicaría que se ha introducido algún error en el nuevo código). Incluso 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.

2.2 Generación Automática de Invariantes

Básicamente, existen dos tipos de generadores automáticos de invariantes: estáticos y dinámicos. Los generadores estáticos de invariantes [6] son los más usados: deducen los invariantes de un programa estáticamente, es decir, sin ejecutar su código. Los invariantes generados de esta forma son siempre ciertos, pero el generador es dependiente del lenguaje concreto. Además, su 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 [5] 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. 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 derivado de él 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.

2.3 Generación Dinámica de Invariantes en Composiciones WS-BPEL

Consideramos que la generación dinámica de invariantes puede ser una técnica adecuada para ayudar en la prueba de caja blanca de composiciones de WS en WS-BPEL. Si se dispone de un buen conjunto de casos de prueba, los diferentes registros obtenidos tras su ejecución serán una buena muestra de la lógica interna de la composición, incluyendo sus aspectos más delicados, y, por lo tanto, el generador inferirá invariantes correctos y significativos.

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 su simulación.

Una situación específica de WS-BPEL que hay que tener en cuenta es que, 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, se debe permitir que, en el momento de la ejecución de los casos de prueba, las llamadas a ciertos servicios se sustituyan por otras que respondan otros con sus mismas interfaces. Estos se comportarán de acuerdo a la especificación del caso de prueba, pudiendo devolver un mensaje predeterminado o dar un fallo concreto si así se ha indicado. Por supuesto, esto complicará la implementación del framework que proponemos, pero nos dará una mayor operatividad.

3 Implementación del framework

El framework implementa la arquitectura que presentamos en [7]. En ella se integran diversos sistemas libres muy probados: ActiveBPEL, BPELUnit y Daikon.

El proceso de generación de invariantes se divide en tres etapas fundamentales, que pasamos a describir en los siguientes apartados. A la vez vamos mostrando como nuestro framework genera invariantes de una versión simplificada de la composición del préstamo bancario descrita en [2].

Nuestra versión de la composición del préstamo bancario usa actividades en vez de secuencias para implementar su lógica interna. Se han añadido variables para separar las entradas y salidas de cada servicio y de la composición en sí. Así pues, la variable `loanInfo` de la composición original se ha reemplazado por las variables `processInput` y `approverInput`, y la variable `approval` por `processOutput` y `approverOutput`.

3.1 Etapa de instrumentación

En esta primera etapa recibimos los ficheros de definición de la composición WS-BPEL y le añadimos la lógica necesaria para que durante su posterior ejecución en cada caso de prueba genere registros de ejecución con la información que necesitará el generador de invariantes. También se aprovecha para crear ficheros específicos de ActiveBPEL necesarios para ejecutar la composición posteriormente.

Para ello, hemos creado e integrado en ActiveBPEL funciones XPath que complementan la capacidad de registro del sistema. Estas funciones no modifican en ningún

momento el comportamiento de la composición: simplemente consultan los valores de las variables en ese punto del programa y lo escriben en el registro de ejecución de salida.

Por ejemplo, el listado 1.1 muestra un fragmento de código fuente de nuestra composición de préstamo instrumentada. Es un ejemplo sencillo, pero sirve para mostrar la instrumentación de cualquier instrucción WS-BPEL, sea lo compleja que sea. En concreto, indica que el préstamo, que ya sabemos que solicita una cantidad pequeña y presenta riesgo bajo (según el asesor externo) ha sido aprobado. En él las instrucciones resaltadas son las añadidas en la instrumentazación para registrar los valores de las variables antes y después de la asignación.

Listado 1.1. Instrumentación del código fuente del proceso

```
<sequence>
 <assign>
 <copy ignoreMissingFromData=”yes”>
 <from>reg:inspect(’$processOutput.output’)</from>
 <to>$dummy_processOutput.output</to>
 </copy>
 </assign>
   <assign name="approveLoan">
     <copy>
       <from>true()</from>
       <to>$processOutput.output/accept</to>
     </copy>
   </assign>
 <assign>
 <copy ignoreMissingFromData=”yes”>
 <from>reg:inspect(’$processOutput.output’)</from>
 <to>$dummy_processOutput.output</to>
 </copy>
 </assign>
</sequence>
```

3.2 Etapa de ejecución

De la etapa anterior recibimos la versión instrumentada de los ficheros de definición del proceso WS-BPEL y los ficheros específicos del motor que usamos. Así que ahora ejecutaremos la composición con el conjunto de casos de prueba. Y los registros de ejecución generados durante su ejecución se pasarán a la siguiente fase.

BPELUnit es el responsable de desplegar el proceso en ActiveBPEL (haciendo que invoque a nuestros servicios simuladores donde se desee), lanzar el servidor de servicios que reemplazarán a los reales que se desee (*mockup server*) de acuerdo a la especificación externa, llamar al proceso WS-BPEL con los parámetros indicados en la especificación de cada caso de prueba y replegar el proceso WS-BPEL al terminar.

Un trozo simplificado de un registro de ejecución del código del listado 1.1, se muestra en el listado 1.2 en la página siguiente.

Listado 1.2. Ejemplo de registro de ejecución del framework

```
Executing [(...)/sequence/assign]
 INSPECTION($processOutput.output/accept) = false []
Completed normally [(...)/sequence/assign]
Executing [(...)/sequence/assign]
Completed normally [(...)/sequence/assign]
Executing [(...)/sequence/assign]
 INSPECTION($processOutput.output/accept) = true []
Completed normally [(...)/sequence/assign]
```

En primer lugar, el valor original de `accept` se consulta. La segunda asignación de la secuencia almacena en la variable `$processOutput.output/accept` el valor `true` (verdadero), indicando de este modo que el préstamo se ha aceptado (esta instrucción es del código WS-BPEL original). Finalmente, la última consulta confirma el cambio en la variable.

3.3 Etapa de análisis

En la etapa previa cada caso de prueba de la especificación inicial generó su registro de ejecución. Ahora queda pasarlos a Daikon, nuestro generador de invariantes, para que haga el resto del trabajo. Para ello antes hay que adaptar dichos registros al formato de entrada de Daikon.

Tras pasar estos ficheros a Daikon, obtenemos finalmente el listado de invariantes deducidos. A continuación (listado 1.3) incluimos varios de los invariantes conseguidos en nuestra composición de ejemplo a aplicarle un conjunto de casos que solicitabas cantidades pequeñas (por debajo de los 10,000 dólares).

Listado 1.3. Invariantes generados por el framework

```
1 LoanApproval.LargeAmount:::EXIT
2 approverInput.input.amount == processInput.input.amount
3 approverOutput.output.accept == processOutput.output.accept
4 approverInput.input.amount == [150000.0]
5 approverOutput.output.accept one of { [0], [1] }
6 size(approverOutput.output.accept) == 1
```

En la primera línea Daikon indica que las invariantes que siguen se dan en el final de la composición. Se observa cómo Daikon usa vectores unidimensionales de para representar las variables de nuestra composición.

En dicho punto ha sido capaz de deducir (línea 2) que la cantidad originalmente solicitada es la que se envía al servicio aprobador. Igualmente deduce (línea 5, en la que verdadero y falso se representan con 1 y 0 respectivamente) que el servicio aprobador no aprueba todos los préstamos, y que su respuesta de aprobación se usa como salida final del proceso (línea 3).

Sin embargo la salida del framework se podría mejorar. Por ejemplo, nos indica en la línea 4 que la cantidad solicitada es siempre 150,000 dólares, lo que sabemos que es falso. La razón por la que lo deduce es porque sólo un subconjunto del conjunto inicial

de casos de prueba llegan a ejecutar dicho punto del programa. En concreto, todos los casos de prueba en los que se solicitaba más de 10.000 dólares (que es la cifra límite) pedían 150,000. De ahí el invariante erróneo. Para falsificarlo bastaría con añadir casos de prueba con cantidades superiores a 10,000 que no sean 150,000.

Además, existen invariantes redundantes: si sabemos que `accept[]` es el vector unidimensional `[0]` o `[1]`, no necesitamos que nos indique en la línea 6 que su longitud es siempre 1. Es más, ya sabemos que el valor que almacene la variable booleana `accept[]` sólo puede ser 0 o 1. Por lo que incluir tal invariante en la salida sólo la complica innecesariamente. Queda como trabajo futuro

4 Trabajos Relacionados

En este apartado presentamos algunos trabajos relacionados:

La relación entre los casos de prueba usados para la generación dinámica de invariantes potenciales y la calidad de los invariantes deducidos se estudia en [8]. Aumentar el conjunto de casos de prueba con ejemplos adecuados puede ser una forma de mejorar la precisión de los invariantes generados y, por extensión, la prueba del programa original.

En [9] se comenta la generación automática de casos de prueba para composiciones WS-BPEL de acuerdo a criterios de cobertura de estados y de transiciones. Usando dichos casos como parte de la entrada del framework se podría mejorar la confianza en los invariantes generados.

5 Conclusiones y Trabajo Futuro

En este artículo hemos presentado un framework que genera dinámicamente invariantes potenciales a partir de una composición WS-BPEL y un conjunto de casos de prueba. Creemos que una técnica adecuada para superar las dificultades que la aplicación de técnicas tradicionales de prueba de caja blanca presenta para WS-BPEL. Esto es, en gran medida, gracias a que la generación de invariantes está basada en la información de registros obtenidos a partir de ejecuciones reales de la composición.

Hemos demostrado como nuestro framework genera invariantes significativas de la composición del préstamo bancario. Pero además, se dedujeron algunos invariantes obvios y redundantes. Incluso algunos invariantes falsos aparecieron como consecuencia de las limitaciones del conjunto de casos de prueba usados para inferirlos.

Nuestro siguiente paso será estudiar la escalabilidad del sistema para composiciones más grandes, en las que haya una mayor cantidad de puntos del programa a observar sea mayor y se usen más variables. Después comprobaremos la relación entre la calidad de los invariantes inferidos por el framework y la del conjunto de casos de prueba empleado para generarlos. Por último, podríamos utilizar los invariantes generados como apoyo a la prueba de caja blanca para WS-BPEL y comprobar si mejora sus resultados.

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. (June 2007)
2. OASIS: WS-BPEL 2.0 standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (April 2007)
3. Bertolino, A., Marchetti, E.: A brief essay on software testing. In Thayer, R.H., Christensen, M., eds.: Software Engineering, The Development Process. 3 edn. Wiley-IEEE Computer Society Pr (2005)
4. Bucchiarone, A., Melgratti, H., Severoni, F.: Testing service composition. In: ASSE: Proceedings of the 8th Argentine Symposium on Software Engineering. (2007)
5. 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) (February 2001) 99–123
6. Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Computer Science **173**(1) (1997) 49–87
7. García Domínguez, A., Palomo Duarte, M., Medina Bulo, I.: Framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. In: Actas de las XIII Jornadas de Ingeniería del Software y Bases de Datos (en evaluación), Gijón (October 2008)
8. Gupta, N.: Generating test data for dynamically discovering likely program invariants. In: ICSE, Workshop on Dynamic Analysis. (2003)
9. Zheng, Y., Zhou, J., Krause, P.: An automatic test case generation framework for web services. Journal of Software **2**(3) (September 2007) 64–77