UNIVERSIDAD DE CÁDIZ

ESCUELA SUPERIOR DE INGENIERÍA

Generación dinámica de invariantes para composiciones de servicios web en WS-BPEL

Tesis doctoral

Manuel Palomo Duarte

Ingeniero en Informática

Cádiz, 2011

Departamento de Lenguajes y Sistemas Infomáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
Cádiz, España

TESIS DOCTORAL

Generación dinámica de invariantes para composiciones de servicios web en WS-BPEL

Autor:
Manuel Palomo Duarte
Ingeniero en Informática

Directora:
María Inmaculada Medina Bulo
Doctora en Informática

Cádiz, España, 2011

CONFORMIDAD DE LA DIRECTORA DE TESIS PARA LA TRAMITACIÓN DE LA TESIS DOCTORAL

Dª María Inmaculada Medina Bulo, profesora del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Cádiz, siendo Directora de la Tesis titulada *Generación dinámica de invariantes para composiciones de servicios web en WS-BPEL*, realizada por el doctorando D. Manuel Palomo Duarte dentro del programa de doctorado Ingeniería en Automática y Electrónica Industrial, Ingeniería Informática y Sistemas Eléctricos perteneciente al bienio 2003/05, para proceder a los trámites conducentes a la presentación y defensa de la tesis doctoral arriba indicada, en aplicación del art. 30 de la Normativa Reguladora de Estudios de Tercer Ciclo de la Universidad de Cádiz, informa que se autoriza la tramitación de la tesis.

La directora de tesis

María Inmaculada Medina Bulo

En Cádiz, España, a 8 de abril de 2011

Esta tesis, todos los programas necesarios para repetir los experimentos
que contiene y los datos resultantes de realizarlos están disponibles
en la siguiente URL persistente

`http://purl.org/mpalomo/tesis`

Si se desea más información sobre prueba de caja blanca en WS-BPEL
con invariantes puede consultar la web oficial de Takuan

`http://neptuno.uca.es/~takuan`

La mayor parte del capítulo 2 de este trabajo está publicado
en el capítulo "Service Composition Validation and Verification"
del libro "Service Life Cycle Tools and Technologies: Methods, Trends and Advances"
editado por IGI Global en inglés bajo licencia privativa.

En la Grecia Clásica las lentejas eran el principal alimento del pueblo llano. Pasó un ministro del emperador y al ver a Diógenes comiéndolas le dijo: «¡Ay, Diógenes! Si aprendieras a ser sumiso y adularas al emperador, no tendrías que comer lentejas».

Diógenes contestó: «Si tú aprendieras a comer lentejas no tendrías que ser sumiso y adular al emperador».

Diógenes de Sínope, "El Cínico"

AGRADECIMIENTOS

Gracias a los que han hecho posible este proyecto:
A todos los que han liberado el software usado en esta tesis.
A los miembros de SPI&FM y UCASE.
A Antonio, Alejandro y Javi por su buen hacer.
A Inma, por su infinita paciencia en la direción de esta tesis.
A Paco, por todo lo que me ha enseñado de esta ciencia llamada Informática … desde la videoconsola Phillips.
A Ignacio, mi fiel escudero desde que tengo uso de razón.
A Larisa, con quien me han pasado las mejores cosas de la vida.
A Luibov, la que más preguntaba por esta tesis.
A mis padres, a los que ahora entiendo muchísimo más.
A Pilar y Alejandra, que tanto me están enseñando … sin saber hablar todavía.
A los que ya no están y a los que vendrán.
Gracias a todos.

Manuel Palomo Duarte
Abril de 2011

AGRADECIMIENTOS INSTITUCIONALES

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).

En el presente documento se sigue la siguiente notación: Los términos extranjeros (por ejemplo, inglés) aparecerán *en cursiva*; los nombres de ficheros tendrán una `fuente monoespaciada`; los nombres de variables, así como sus campos, propiedades y valores estarán en *modo verb*, y los elementos de un documento XML (incluyendo las instrucciones de WS-BPEL) estarán contenidos entre <ángulos>.

Extractos de ficheros con texto plano o código XML aparecerán

Listado: *Ejemplo de listado*

```
1 Dentro de un recuadro como este, con numeracion a la izquierda
2 Y sin acentos, por compatibilidad con cualquier codificacion
```

Por último, cuando se indique una orden a ejecutar en línea de consola, se usará una fuente monoespaciada, y la orden irá precedida por el símbolo del dólar, como se observa a continuación.

```
$ orden parámetros
```

RESUMEN

En los últimos años, las arquitecturas orientadas a servicios están cambiando la filosofía de desarrollo de software en muchos entornos. El uso de servicios web facilita significativamente la interoperabilidad entre sistemas, permitiendo programar sistemas de gran tamaño usando otros más simples de manera sencilla. El principal lenguaje para componer servicios es WS-BPEL 2.0, que ha sido estandarizado por OASIS con la participación de las grandes empresas del sector informático. Sin embargo, las principales técnicas de prueba no han sido adaptadas a WS-BPEL, quedando como uno de los principales retos para su adopción.

El objetivo de esta tesis es estudiar la validez de la generación dinámica de invariantes (también conocida como generación de invariantes potenciales) para apoyar la prueba de caja blanca de composiciones de servicios web en WS-BPEL. En primer lugar, la tesis comprueba la viabilidad de la generación dinámica de invariantes para WS-BPEL. Para ello se presenta una arquitectura basada en el generador dinámico de invariantes Daikon, que ha demostrado ser útil para lenguajes imperativos como C/C++, Java y Perl. Daikon es software libre, y se ha integrado con código propio y otros dos sistemas libres: el motor de ejecución compatible WS-BPEL 2.0 ActiveBPEL y la biblioteca de prueba unitaria para WS-BPEL BPELUnit, que incorpora un mecanismo de simulación de servicios web (pues puede haber servicios no disponibles para pruebas). Los tres sistemas han sido adaptados para crear Takuan, el único generador dinámico de invariantes para WS-BPEL hasta la fecha.

Tras implementar Takuan se realizaron pruebas para comprobar la utilidad de los invariantes que generaba. En ellas se observaron diversas mejoras específicas de WS-BPEL que permitirían optimizar su rendimiento. Tras implementarlas y evaluarlas, se obtuvo una mejora sustancial tanto en el tiempo de ejecución de Takuan como en la cantidad de invariantes que produce, descartando muchos invariantes no informativos y redundantes. Una vez estuvieron dichas mejoras implementadas se demostró la estabilidad de los invariantes generados por Takuan, permitiendo asegurar su correcto funcionamiento con un conjunto soporte adecuado. Los resultados obtenidos permiten afirmar la validez de la generación dinámica de invariantes para apoyar la prueba de caja blanca de composiciones WS-BPEL.

ABSTRACT IN ENGLISH

Service-oriented architecture is changing software development in many ways. The adoption of web services has eased system interoperability, and simple services can be composed to create larger ones. The main language for service composition is WS-BPEL 2.0. It has been backed by the major companies in the IT sector, and standardized by OASIS. Nevertheless, the main testing techniques have not been updated for this language, this being one of the most important challenges for its wide adoption.

The aim of this PhD thesis is demonstrating the feasibility of using the dynamic invariant generation (also known as likely invariant generation) to support WS-BPEL composition white-box testing. Firstly, an architecture based on the Daikon dynamic invariant generator is introduced. Daikon has proven to be a successful system to assist white-box testing of programs written in imperative languages like C/C++, Java and Perl. It is open-source software, and it has been integrated with our own code and two well-tested open-source systems: the WS-BPEL 2.0 compliant ActiveBPEL engine and the BPELUnit unit testing library (that includes web service simulation with mockups). The three systems have been modified to create the only dynamic invariant generation workflow for WS-BPEL available up to date: Takuan.

After implementing Takuan, different executions showed that invariants produced were interesting. Nevertheless, some WS-BPEL-specific inefficiencies concerning its performance were detected. They were overcome by writing code to improve its running time and the quality of the invariants produced (discarding many non-informative or redundant ones). After that, an experiment studied the stability of Takuan output when using different test suites as its input. Results proved dynamic invariant generation an interesting technique to assist in WS-BPEL white-box testing.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

ÍNDICE DE LISTADOS

CAPÍTULO 1

Introducción

Según el *Software Engineering Body of Knowledge* (SWEBOK) [IEE04], «la verificación y validación [de software] persigue la calidad de un producto, y usa técnicas de prueba para localizar defectos y arreglarlos». La Verificación y Validación de software (V&V) ha sido un problema clásico en la Ingeniería Informática desde la *crisis del software*, y ha provocado cuantiosas pérdidas materiales y personales [Lev95]. Desde entonces se han realizado gran cantidad de esfuerzos para que los programas de ordenador cumplan sus requisitos, pero a fecha de hoy ninguno ha demostrado ser definitivo [Mye04].

Según Edsger Dijkstra una de las principales causas es la creciente complejidad de los sistemas software que se desarrollan [Dij72]:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

Las *Arquitecturas Orientadas a Servicios* (AOS, *SOA* en inglés) están cambiando la filosofía de desarrollo de software en muchos entornos. Mediante el uso de Servicios Web (SW, *WS* en inglés), se facilita significativamente la interoperabilidad entre sistemas, permitiendo que se pueda componer un sistema de gran tamaño usando otros más simples de manera sencilla.

El principal lenguaje para componer SW es WS-BPEL 2.0 [OAS07b], que ha sido estandarizado por OASIS con la participación de las grandes empresas del sector TIC (Microsoft, IBM y Oracle entre otras) [OAS03]. Para ello, WS-BPEL incorpora instrucciones no presentes en otros lenguajes de programación, como instrucciones para la invocación de SW, compensaciones de errores en SW, etc. Esto hace que sea necesario adaptar las técnicas clásicas de V&V a este nuevo lenguaje [Boz10].

El *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (perteneciente al *Computing Curricula Series*) [Sof04], afirma que «la verificación y validación de software hace comprobaciones sobre un sistema usando técnicas estáticas y dinámicas para asegurar que el programa resultante esté conforme a su especificación y cumpla las expectativas de los clientes». Las técnicas de prueba de software estáticas y dinámicas son complementarias [Ber05], de modo que ninguna de ellas tendría que descartarse para asegurar la producción de un software de calidad.

Dentro de las diferentes técnicas dinámicas de V&V, este trabajo se centra en la *generación dinámica de invariantes*, que se ha utilizado con éxito en la prueba y mejora de programas escritos en lenguajes imperativos [Ern07]. Hasta la fecha de realización de este estudio no existía ningún generador dinámico de invariantes para WS-BPEL. Por lo tanto, se plantea como objetivo de esta tesis estudiar la validez de la generación dinámica de invariantes (también conocida como generación de invariantes potenciales) para la prueba de composiciones de servicios web en WS-BPEL.

Conviene aclarar en este punto que en este documento se utilizan los términos *invariante* e *invariante potencial* en el mismo sentido en que se usa en la bibliografía de la materia [Ern01, Ern07, Gup03], 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 (también llamado *invariante dinámico* o *invariante generado dinámicamente*) a cualquier propiedad que se mantiene en una serie de casos de prueba.

1.1. Objetivos

El principal objetivo de esta tesis es estudiar la generación dinámica de invariantes para la prueba de composiciones de servicios web en WS-BPEL. Para ello, se divide en los siguientes seis sub-objetivos:

1. Análisis de la adecuación de la generación dinámica de invariantes para WS-BPEL.
2. Diseño de una arquitectura para implementar un generador de invariantes dinámico para WS-BPEL.
3. Implementación de un generador de invariantes dinámico para WS-BPEL y comprobación de su correcto funcionamiento.
4. Evaluación del rendimiento del generador de invariantes dinámico para WS-BPEL.
5. Optimización de la salida del generador de invariantes dinámico para WS-BPEL si se considera necesario tras la evaluación de su rendimiento.
6. Realización de pruebas para establecer las condiciones de uso del generador que aseguren resultados fiables.

1.2. Estructura de la tesis

La estructura del resto de este documento es la siguiente:

- En el capítulo dos se presentan los fundamentos de las AOS (prestando especial atención al lenguaje WS-BPEL y sus tecnologías relacionadas) y de la prueba de software. También se hace un estudio del estado del arte en la prueba de WS-BPEL, que incluye tanto las propuestas actuales para la prueba de SW (nótese que, externamente, una composición de SW es un servicio web) y de composiciones de SW.

- Después, en el tercer capítulo, se detallan las aplicaciones de la generación dinámica de invariantes. Esta técnica tiene diferentes utilidades como apoyo en la mejora de la calidad de un programa. Entre ellas profundizamos en su uso en la prueba de caja blanca.

- En el cuarto capítulo se presenta la arquitectura diseñada para la generación dinámica de invariantes en WS-BPEL, y el sistema resultante de su implementación, Takuan. Esta arquitectura parte de la base del generador dinámico de invariantes Daikon, sobre la que se hacen diversas adaptaciones y ampliaciones específicas para WS-BPEL. Entre otros aspectos técnicos destacan la implementación de SW simulados, que permitan probar la composición cuando no se pueden usar SW reales, y la implementación de dos métodos de correspondencia entre las variables WS-BPEL (de estructura arbórea) y las usadas por Daikon (de estructura lineal). Se incluyen dos ejemplos de uso de Takuan, en uno se detecta una limitación de un conjunto de casos de prueba y en otro se encuentra un fallo en el código de una composición.

- Posteriormente, en el capítulo cinco, se analizan los resultados obtenidos con Takuan y se comentan distintas optimizaciones realizadas para mejorarlo. Estas optimizaciones se centran en dos aspectos: mejora de su tiempo de ejecución y descarte de información no relevante en su salida. En el primer caso, se evita la comprobación de invariantes que estén incluidos en las restricciones de tipos XML Schema. En el segundo, se evita la generación dinámica de invariantes frutos del azar que incluyan variables no comparables.

- El siguiente capítulo, el sexto, muestra los resultados de uso de Takuan con distintos conjuntos de casos de prueba, demostrando las condiciones de uso adecuadas para generar invariantes dinámicos fiables. Se realiza en primer lugar un estudio cuantitativo (sobre la cantidad de casos de prueba), y posteriormente otro cualitativo (relativo a su cobertura).

- En el último capítulo se presentan las conclusiones del trabajo y se exponen las líneas de trabajo futuro que abre esta tesis.

Además, el documento incorpora dos apéndices con el siguiente contenido:

- El primer apéndice está dedicado a explicar las composiciones WS-BPEL que se usan a lo largo del texto. Se detalla su comportamiento general y se ofrece un diagrama de flujo de ellas.
- En el segundo apéndice se incorpora el manual de instalación y uso de Takuan. En él se explica su uso mediante consola, como extensión (*plug-in*) del entorno de desarrollo Netbeans o mediante la aplicación gráfica de escritorio en fase *beta IdigInBPEL*.

Asimismo, en [PD11] se pueden obtener los ficheros íntegros que se muestran en extractos en el texto, así como toda la información necesaria para repetir los distintos experimentos que se detallan en esta tesis.

CAPÍTULO 2

Fundamentos y estado del arte

En este capítulo se comentan las características de las AOS que influirán en el desarrollo de la tesis, presentando el funcionamiento de los SW y las técnicas para componerlos (en especial, WS-BPEL). Por último se hace una revisión del estado del arte en la V&V de SW y en las composiciones de SW.

2.1. Las arquitecturas orientadas a servicios

Aunque no existe una definición consensuada para las AOS, se pueden considerar como referencias las proporcionadas por OASIS y W3C. OASIS es «un consorcio de empresas creado sin ánimo de lucro con objeto de guiar el desarrollo, la convergencia y la adopción de estándares abiertos para la sociedad de la información a nivel global» [OAS98]. Mientras que el W3C se define como «una comunidad internacional donde las organizaciones miembro, personal a tiempo completo y el público en general trabajan para desarrollar estándares web» [W3C09].

- OASIS definió las AOS en su estándar *OASIS Reference Model for Service Oriented Architecture 1.0* de 2004 [OAS06] como «un paradigma para la organización y explotación de funcionalidades distribuidas que pueden estar controladas por distintos elementos propietarios».

  Sin embargo, esta definición ha evolucionado en 2009 dentro del *OASIS Reference Architecture Foundation for Service Oriented Architecture 1.0* [OAS09] (actualmente en fase de borrador) a esta otra: «desde un punto de vista integral, una AOS es una red de servicios y máquinas independientes, las personas que manejan, modifican, usan y gobiernan esos servicios, así como sus proveedores de equipamiento y personal. Esto incluye cualquier ente animado o inanimado que afecte o pueda ser afectado por el sistema. Dentro de un sistema de tal magnitud es evidente que nadie tiene el control ni la responsabilidad de todo el (eco)sistema, aunque sí que hay personas que tienen un determinado control e influencia sobre la comunidad».

- Mientras que de acuerdo al apartado *Service Oriented Architecture* incluido en el *Web Services Architecture Note* del W3C [W3C04a]: «una AOS es un tipo de arquitectura para sistemas distribuidos que suele tener las siguientes características: perspectiva lógica [...], orientada a mensajes [...], orientada a la descripción [...], granularidad [...], orientada a red [...] e independiente de la plataforma [...]».

Extrayendo las ideas comunes se pueden ver las AOS como una serie de tecnologías que facilitan y potencian la integración de software de manera independiente de lenguajes de programación, sistemas operativos y arquitecturas hardware. Para ello, se basa en el concepto de servicio web, entendiendo como tal un software con el que puede interoperar enviando mensajes a través de la web según protocolos estandarizados [Gol04].

Este modelo tiene muchas implicaciones en diferentes facetas, entre ellas destacan:

Reducción de costes en el desarrollo mediante el uso de interfaces no dependientes de ningún proveedor se facilita la integración de sistemas. Esto incluye también sistemas que no fueron diseñados para funcionar en red (en inglés se suelen denominar *legacy systems*), que en la actualización de sistemas de información en las empresas [Erl04, Cor07], es un elemento clave.

Implementación de modelos de negocio en software Esta arquitectura permite definir modelos de negocio, indicando qué información fluye entre qué servicios, qué elementos toman decisiones, en base a qué datos lo hacen, cómo se comportan ante determinadas circunstancias, etc. [Pan08, Mar06].

Gobernabilidad Una vez implementada la lógica de negocio, se tiene una visión de muy alto nivel del sistema informático en general. Gracias a ella es posible monitorizar y controlar el sistema en tiempo real aplicando directrices y restricciones [Bro08].

Interoperabilidad el uso de tecnologías estándares posibilita la colaboración entre sistemas de distintas empresas, independientemente de las plataformas que use cada uno [Erl05].

Bajo acoplamiento del software Al existir un acoplamiento muy débil entre componentes aumenta significativamente la reusabilidad del software. Por ejemplo, es posible usar un servicio determinado para una tarea, pero si cae o si un aumento de la carga de trabajo no permite que funcione en el tiempo esperado, se puede sustituir en la lógica de la AOS por otro (quizás proporcionado por otra empresa). Igualmente es posible seleccionar servicios dinámicamente, facilitando la ubicuidad [Yoo07].

Reinvención del modelo de desarrollo de software Las AOS permiten establecer metodologías de desarrollo de software más dinámicas que las convencionales [Gho08, Med09].

Por todas estas características, parece que las AOS serán una de las claves de la informática a corto y medio plazo [Hef07]. Pero para ello es necesario que existan una serie de tecnologías *de referencia* que aseguren dicha interoperabilidad, para lo cual se necesita una base de SW interoperables sobre la que opere el *software de conectividad* (del inglés *middleware*) encargado de las funciones comentadas anteriormente.

2.1.1. Los servicios web

Al igual que sucede con las AOS, tampoco existe una definición consensuada de SW por parte de W3C y OASIS.

Según el W3C [W3C04b] un servicio web «es un sistema software diseñado para hacer posible la interacción entre máquinas sobre una red. Su interfaz está descrita usando un formato procesable por una máquina (en concreto WSDL). Los sistemas que interactúan con el servicio web lo hacen según se indica en su descripción usando mensajes SOAP, normalmente enviados a través de HTTP (tras serializar su contenido XML) y junto con otros estándares relacionados con la web».

Mientras que según OASIS [OAS05a] el término servicio web «se refiere a un tipo de software diseñado para permitir la oferta y consumo de servicios sobre la web respetando estándares abiertos como XML, SOAP, WSDL y UDDI».

Además, existe una tercera organización, la *Web Services Interoperability Organization* participada por las grandes empresas de tecnologías de la información a nivel mundial, que recomienda el uso de determinadas tecnologías para implementar SW. Dichas recomendaciones se organizan en *perfiles* (del inglés *profiles*). Su web oficial ofrece herramientas para su verificación [Org]. El perfil más usado para SW es el *WS-I Basic Profile* [Org10], que está soportado por las principales herramientas AOS [Dom07].

Todas estas tecnologías están basadas en XML, lo que posibilita el intercambio de información en modo texto, facilitando su procesamiento automático. Entre ellas destacan las siguientes tecnologías estandarizadas por W3C[1], que son las bases de los servicios web [New04]: SOAP, WSDL, XML Schema y XPath. A continuación se comentan brevemente junto a UDDI (estandarizada por OASIS) y al término WS-Stack.

SOAP

SOAP es un protocolo para el intercambio de información en formato texto a través de la red. Es independiente de la plataforma, flexible y fácil de extender. SOAP 1.2 es una Recomendación del W3C [W3C07a]. Su principal ventaja es que puede operar sobre diversos protocolos. En concreto, lo más común es que lo haga sobre HTTP, lo que facilita su implantación sobre la infraestructura que tienen las empresas (evitando complicar así la administración de la red que pueda tener pasarelas, cortafuegos, etc.).

[1] De acuerdo a la nomenclatura de W3C, las Recomendaciones son equivalentes a estándares de otras organizaciones.

WSDL

WSDL es un lenguaje para describir interfaces entre servicios web. WSDL 2.0 (*Web Services Description Language*) es una Recomendación del W3C [W3C07c] que permite detallar la interacción entre entidades. WS-BPEL usa WSDL 1.1, versión recomendada por el WS-I Basic Profile, pero que no tiene categoría de estándar W3C [W3C05b]. Sin embargo, W3C ofrece un traductor automático de documentos WSDL 1.1 a 2.0 [W3C06].

WSDL describe los servicios que ofrece un SW. Por cada servicio se indican una serie de datos básicos, como la URL en la que escucha, el protocolo de comunicaciones que usa, el tipo de mensaje que espera, los tipos de datos con los que opera (que suelen describirse en XML Schema), etc.

XML Schema

XML Schema es una Recomendación del W3C [W3C10] para especificar tipos de datos. Se puede encontrar muchas veces nombrado como XSD (acrónimo inglés de *XML Schema Definition*), para evitar ambigüedad con el término inglés *XML schema* (que es el esquema de un documento XML concreto, no el lenguaje de definición de tipos).

XML Schema tiene una gran cantidad de tipos de datos básicos: cadenas de caracteres, números enteros, flotantes, fechas, idioma, URL, etc. A estos datos básicos se les puede añadir restricciones. Por ejemplo, se pueden definir tipos de datos para números enteros entre el 10 y el 20, cadenas de caracteres que empiecen por la cadena *CA* y tengan después 4 cifras, tipos enumerados, etc. Además, estos tipos básicos se pueden agregar para formar tipos complejos, por ejemplo, listas de entre 2 y 8 números enteros positivos. Se pueden consultar más ejemplos de tipos de datos en [W3C04c].

XPath

XPath es un lenguaje que permite extraer información de documentos XML de una manera fácil pero potente. W3C ha publicado dos versiones estándares de XPath: 1.0 y 2.0 [W3C07d]. WS-BPEL 2.0 usa por omisión XPath 1.0.

A continuación se muestra un ejemplo aplicado a una variable XML de la composición de la metabúsqueda en Internet detallada en el apéndice A.2. El contenido de la variable se muestra en el listado 2.1, mientras que en el listado 2.2 hay una serie de ejemplos de expresiones XPath aplicadas a ella. El listado corresponde a una búsqueda de información en Internet usando dos SW: Google y MSN. La información proporcionada incluye el número de elementos obtenidos en total y de cada servicio, además de la URL, el título, descripción (en inglés *snippet*) y origen de cada resultado concreto.

Listado 2.1: *Contenido simplificado de variable XML Schema en la metabúsqueda.*

```
1 <MetaSearchProcessResponse>
2   <noResult>3</noResult>
3   <noFromGoogle>1</noFromGoogle>
4   <noFromMSN>2</noFromMSN>
```

```
5
6    <result>
7     <url>http://softwarelibre.uca.es</url>
8     < title >OSLUCA</title>
9     <snippet>Oficina de Software Libre, Universidad de Cadiz</snippet>
10    <from>Google</from>
11   </result>
12
13   <result>
14    <url>http://cudisol.ourproject.org</url>
15    < title >CUDISOL</title>
16    <snippet>Cursos para la Difusion del Software Libre</snippet>
17    <from>MSN</from>
18   </result>
19
20   <result>
21    <url>http://jornadas.adala.org</url>
22    < title >Jornadas Andaluzas de Software Libre</title>
23    <snippet>Organizadas por ADALA, CAGESOL y OSLUCA, Algeciras 2004</snippet>
24    <from>MSN</from>
25   </result>
26 </MetaSearchProcessResponse>
```

XPath traduce una estructura XML en un árbol de nodos sobre el que ejecuta la consulta indicada. En la primera línea del listado 2.2 se consultan todos los nodos tipo `result` hijos de *MetaSearchProcessResponse*. En la segunda, se extrae el segundo nodo `result` hijo de *MetaSearchProcessResponse*. La tercera línea muestra el primer y segundo nodo `result` hijos de la variable. La siguiente selecciona todos los nodos `result` hijos de *MetaSearchProcessResponse* que tengan en el nodo `from` el valor "Google". Mientras que la última obtiene todas las descripciones de la variable, da igual quien sea su nodo padre. Se pueden consultar más ejemplos en [W3S10].

Listado 2.2: *Ejemplos de expresiones XPath.*

```
1  MetaSearchProcessResponse/result
2  MetaSearchProcessResponse/result[2]
3  MetaSearchProcessResponse/result[position()<3]
4  MetaSearchProcessResponse/result[from='Google']
5  MetaSearchProcessResponse//snippet
```

UDDI

UDDI permite gestionar repositorios de especificaciones WSDL. Así se facilita que los proveedores de servicios puedan dar sus servicios a conocer y los consumidores puedan dinámicamente descubrir e invocar servicios. UDDI 3.0 es un estándar de OASIS [OAS05b], sin embargo, su implantación es muy limitada en la actualidad. Parece que una de las principales razones es la falta de información semántica en las descripciones de servicios web, lo que dificulta la consulta automática de repositorios. Existen

diversas propuestas académicas [Akk03, Pao02, Mar07a, Sri05], pero ninguna ha sido adoptada por la industria y la mayoría de los servidores UDDI actualmente disponibles tiene un uso bastante escaso.

WS-Stack

Existen otras muchas tecnologías relacionadas con las AOS y los SW. Normalmente suelen denominarse la *pila de SW* (en inglés *WS-Stack*) [Pap07]. Algunas de estas tecnologías, por ejemplo, amplían SOAP para proporcionar determinadas características que no incluye por defecto: negociación de condiciones de operación (WS-Policy [W3C07b]), seguridad en las comunicaciones (WS-Security [Com06]), transacciones atómicas (WS-Transaction [Com09]), etc. Parece que a medio plazo pueden implantarse muchas de ellas, pues se encuentran en diversas fases del proceso de estadarización. Sin embargo, su soporte en las herramientas AOS actuales es muy limitado [Dom07].

2.1.2. Composición de servicios web

Cuando se trabaja con SW no siempre se dispone de un servicio que haga exactamente la tarea que se desea (o no de la forma que se desea). En estos casos es necesario crear servicios más específicos usando otros ya definidos. Esta tarea se denomina composición de servicios, para ello hay dos técnicas principales: orquestación y coreografía [Pap07]. A continuación se describen los detalles de cada una de ellas:

- La orquestación se basa en un sistema central que coordina los demás, siendo responsable de invocarlos cuando sea necesario, en el orden adecuado y componer la lógica del nuevo servicio. Este sistema central se suele denominar *director*, y es el único que interactúa con el cliente. El lenguaje de orquestación de servicios más conocido es WS-BPEL, que tiene el respaldo de ser estándar OASIS [OAS07b].

- Por su parte, la composición por coreografía supone que cada servicio tiene parte de la responsabilidad de que la composición funcione de manera correcta sin que exista un control centralizado. De esta forma, la lógica se distribuye entre los distintos SW. Uno de los lenguajes más habituales de coreografía es WS-CDL [Gro05], cuya versión 1.0 es Candidata a Recomendación del W3C [W3C05a].

En la figura 2.1 en la página siguiente se muestra un ejemplo de una orquestación y de una coreografía en el que se puede observar la diferencia entre los dos modelos. La orquestación ha logrado una mayor implantación debido a su mayor sencillez, menor acoplación entre componentes y facilidad de control. De hecho, el lenguaje de orquestación WS-BPEL fue estandarizado en 2007 por OASIS a petición de las principales empresas del sector TIC: Oracle, Microsoft, IBM, HP, etc. [OAS07b], mientras que el de coreografía WS-CDL, se mantiene como Candidata a Recomendación del W3C [W3C05a] desde 2005.

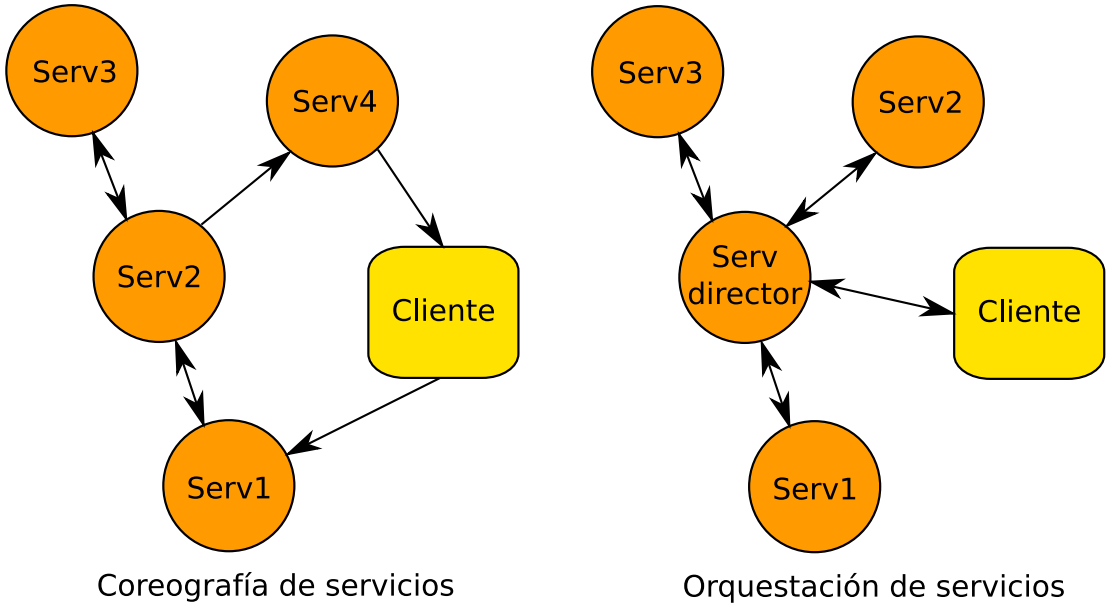


Figura 2.1: *Ejemplo de composición por coreografía y orquestación.*

2.1.3. El lenguaje WS-BPEL

El acrónimo WS-BPEL proviene del inglés *Web Services Business Process Execution Language* (en español «Lenguaje de Ejecución de Procesos de Negocio con Servicios Web»). WS-BPEL 2.0 está normalizado por OASIS, que lo define en su estándar como «un lenguaje para especificar el comportamiento de procesos de negocio basados en servicios web» [OAS07b]. Básicamente se puede considerar WS-BPEL como un lenguaje de programación a gran escala (en inglés *programming in-the-large*) que permite componer estáticamente un SW más potente orquestando otros disponibles [Cur03].

WS-BPEL nació por la unión de los lenguajes WSFL (de IBM) y XLANG (de Microsoft), y fue propuesto como especificación no estándar bajo el nombre BPEL4WS por varias de las grandes empresas del sector TIC. Estas enviaron una propuesta de estandarización de BPEL4WS 1.1 a OASIS en 2003 [OAS03]. Como respuesta, OASIS creó poco después el *WS Business Process Execution Language Technical Committee* para trabajar en su estandarización, publicando la primera versión estandarizada, y la última hasta la fecha, WS-BPEL 2.0, en 2007 [OAS07b]. En la bibliografía sobre el tema puede encontrarse el término BPEL a secas para referirse a cualquier versión del lenguaje.

La estandarización de WS-BPEL fue un gran hito para su adopción en las herramientas AOS más usadas. En la actualidad es una de las principales características de interoperabilidad en la mayoría de ellas [Dom07]. Sin embargo, algunos sistemas, como Oracle BPEL Process Manager (uno de los entornos más implantados en la actualidad) incluye en sus procesos WS-BPEL determinadas instrucciones no estándares, lo que constituye un peligro para su portabilidad y el futuro del estándar [Ora10].

WS-BPEL usa XML Schema como sistema de tipos, SOAP para el envío de mensajes y WSDL para describir las interfaces del servicio. WS-BPEL describe la lógica de composición de servicios en ficheros de extensión BPEL con etiquetas XML, normalmente generadas mediante una interfaz gráfica (pues está soportado por los principales entornos de desarrollo en la actualidad: Eclipse, Netbeans, etc.). Estas etiquetas especifican actividades concretas, como asignaciones, bucles o envío de mensajes. Al igual que el resto de tecnologías AOS, es independiente de la implementación y plataforma usada tanto por el proveedor de servicios como por el usuario de la composición.

El lenguaje permite definir dos tipos de procesos: ejecutables y abstractos. Los primeros pueden ejecutarse en un motor WS-BPEL, mientras que los segundos sirven para describir cierto comportamiento de una composición, pero dejando aspectos por concretar (a modo de plantilla). Los procesos WS-BPEL pueden incluir, además de instrucciones típicas de lenguajes imperativos (asignaciones, bucles, etc.), otras específicas de AOS para, por ejemplo, la invocación de SW, temporizadores, compensaciones para SW que fallen, etc. Esto obliga a actualizar las técnicas clásicas de prueba para soportarlo adecuadamente.

A continuación se comentan algunas de las características que diferencian WS-BPEL de otros lenguajes de programación tradicionales. Para un mejor entendimiento del estándar WS-BPEL 2.0 se recomienda la lectura de la guía básica (traducción del término inglés *primer*) producida por el OASIS WS-BPEL Technical Committee [OAS07a]. Los principales aspectos del lenguaje se ilustran en el código del listado 2.3 en la página siguiente, que incluye un extracto del código WS-BPEL de la composición del préstamo bancario descrita en el apéndice A.1.1. Este código está basado en el ejemplo del préstamo incluido en el estándar WS-BPEL, y en él se usan puntos suspensivos entre corchetes para indicar trozos del fichero omitidos.

El ejemplo consiste en un sistema que tramita peticiones de préstamos bancarios. Recibe peticiones de clientes, y si la cantidad solicitada es pequeña (por debajo de un umbral) y un servicio externo de asesoría identifica al cliente como un cliente de bajo riesgo de impago, se le concede. En caso de que la cantidad sea alta o haya riesgo de impago, se consulta a un servicio externo de aprobación de préstamos y se concede o no según este responda.

El código comienza con la etiqueta XML de proceso (<process>), y los atributos `name` (nombre del proceso), `targetNamespace` y `xmlns`. A continuación se definen los servicios socios con los que podrá interactuar el proceso. Para ello se usa una sección <partnerLinks>, que contiene elementos de tipo <partnerLink>. Cada uno de ellos define en sus atributos el nombre del socio, tipo de relación que se establece entre el proceso WS-BPEL y él, y el rol que juega el proceso WS-BPEL en dicha relación. De manera similar, el elemento <variable> declara las variables del proceso WS-BPEL, indicando nombre y tipo de cada una en sus atributos.

A continuación aparece la definición del comportamiento del proceso. Este debe comenzar por una actividad de inicio, que puede ser <receive> o <pick>, con el atributo `createInstance` a `yes`. La primera lanza la ejecución secuencial de las instrucciones que

contiene, mientras que la segunda (<pick>) permite indicar una acción concreta a ejecutar dependiendo del mensaje recibido.

Posteriormente el código ejecutable en sí, comprueba en una sentencia <if> si la cantidad está por debajo del umbral de 10.000 euros. En ese caso se ejecutaría una rama que según la respuesta del servicio externo de asesoría concedería el préstamo directamente (si se ha identificado como un cliente de bajo riesgo de impago) o consultaría al servicio asesor en caso contrario. Si la cantidad es superior al umbral directamente se consulta al servicio asesor, para ello se copia el campo `input/ns0:amount` de la variable de entrada *processInput* en el campo `input/ns0:amount` de la variable *approverInput* (que se usará en la llamada al servicio asesor). Posteriormente se llama al servicio asesor con la instrucción <invoke>, indicando el nombre identificador de dicha llamada, el socio (definido anteriormente en un elemento <partnerLink>), la operación que se le solicita (`operation`), el puerto en que se realizará la petición (`portType`), la variable de la que se leerán los datos de la solicitud (`inputVariable`) y la variable a la que se volcarán los datos que se reciban como respuesta (`outputVariable`).

Por último, el proceso termina copiando el campo `output/ns0:accept` de la variable de respuesta del servicio asesor a la variable de respuesta del proceso WS-BPEL, que se envía al cliente mediante la actividad <reply>.

Listado 2.3: *Extracto de fichero BPEL.*

```
1  <process name="LoanApproval"
2   targetNamespace="http://enterprise.netbeans.org/bpel/[...]/LoanApproval"
3   xmlns="http://docs.oasis−open.org/wsbpel/2.0/process/executable" [...] >
4
5   <partnerLinks>
6    <partnerLink name="approver" partnerLinkType="ns1:ApproverService1"
7                 partnerRole="ApproverServicePortTypeRole"/>
8    [...]
9   </partnerLinks>
10
11  <variables>
12   [...]
13   <variable name="approverOutput" messageType="ns1:ApproverServiceOperationReply"/>
14   <variable name="approverInput" messageType="ns1:ApproverServiceOperationRequest"/>
15  </variables>
16
17  <sequence name="Main">
18   <receive name="Receive1" createInstance="yes" partnerLink="client" variable="processInput"
19            operation="requestCredit" portType="ns3:LoanApprovalPortType"/>
20   [...]
21
22   <if  name="If1">
23    <condition>
24     ( number(string($processInput.input/ns0:amount)) &lt; 10000 )
25    </condition>
26
27    <sequence name="SmallAmount">
28     [...]
```

```
29      </sequence>
30
31      <else>
32       <sequence name="LargeAmount">
33        <assign name="copyLoanInfoToApproverInput2">
34         <copy>
35          <from>$processInput.input/ns0:amount</from>
36          <to>$approverInput.input/ns0:amount</to>
37         </copy>
38        </assign>
39
40        <invoke name="queryApprover2" partnerLink="approver" operation="approveCredit"
41                portType="ns1:ApproverServicePortType" inputVariable="approverInput"
42                outputVariable="approverOutput"/>
43
44        <assign name="copyApproval2">
45         <copy>
46          <from>$approverOutput.output/ns0:accept</from>
47          <to>$processOutput.output/ns0:accept</to>
48         </copy>
49        </assign>
50       </sequence>
51      </else>
52
53    </if>
54
55    <reply name="Reply1" partnerLink="client" operation="acceptCredit"
56           portType="ns3:LoanApprovalPortType" variable="processOutput"/>
57   </sequence>
58  </process>
```

Otras características a destacar de WS-BPEL son las siguientes:

Despliegue Las composiciones WS-BPEL se ejecutan en un motor WS-BPEL que suele estar incluido en un servidor web. Al solicitar su despliegue, el servidor le asigna una dirección URI única y queda a la espera de peticiones. Cuando llega un mensaje a la URI, el servidor comprueba si va dirigido a una instancia en ejecución (indicado por su identificador de correlación), y si es así, se lo hace llegar. En otro caso, se comprueba si es un mensaje definido por la composición como *de creación de instancia* (indicado con el atributo `createInstance`), y se crea un nuevo proceso que lo sirva. Este es independiente de los demás procesos creados por el servidor, y puede ejecutar código, llamar a otros servicios, etc. para finalmente dar una respuesta al cliente que envió el mensaje original.

Dependencia del entorno Al igual que cualquier otra tecnología que haga uso de servicios externos, la ejecución de un proceso WS-BPEL es altamente dependiente del entorno. Por ejemplo, una caída de red haría que algunos servicios no fueran alcanzables (o que habiendo sido contactados anteriormente no respondieran en ese momento), y fuera necesario gestionar el fallo y actuar en consecuencia.

Estructura del código En WS-BPEL no se incluyen elementos para encapsular el código (como módulos, funciones, etc.). Por ello, el código se suele estructurar mediante elementos <sequence>, que aseguran la ejecución secuencial de las instrucciones que contiene en el orden en que han sido definidas, o mediante una actividad <flow>, que realiza una ejecución paralela de las instrucciones que contenga. Por ejemplo, en el listado 2.3 se puede observar cómo existe un <sequence> general para todo el código, y uno por cada rama de la instrucción condicional.

Ámbitos Todo proceso WS-BPEL tiene un ámbito principal, en el que toda variable definida será global al proceso. Después, durante el proceso en sí, se pueden definir otros ámbitos (con la instrucción <scope>) en los que las variables que se definan estén disponibles para él y los ámbitos que contenga. Por lo tanto, como WS-BPEL no permite definir procedimientos ni funciones, la comunicación entre los elementos de la composición se debe hacer mediante variables del ámbito global.

Invocación de servicios Las composiciones interaccionan con servicios web denominados *socios* (del inglés *Business Partners*). Por cada servicio socio existen una o más relaciones (en inglés *Partner Links*), en la que cada socio tiene un rol asignado. En el ejemplo se puede observar la definición de una relación con su rol correspondiente (líneas 6 y 7 del listado 2.3). Como se observa en el código, la definición de socios es estática en WS-BPEL. Por lo que en caso de cambiar de socio es necesario modificar el código de la composición y desplegarla de nuevo.

Para interactuar con servicios se dispone de las instrucciones <invoke>, <reply> y <receive>. Con <invoke>, un proceso WS-BPEL puede llamar a un servicio y continuar su ejecución, para lo que se debe proporcionar una variable que contenga la información a enviar en el mensaje SOAP. O bien, es posible llamarlo y esperar a que responda. En este caso deben proporcionarse dos variables, una de entrada (indicada en el atributo <inputVariable>) y otra de salida (en el atributo <outputVariable>), como se puede ver en las líneas 40 a 42 del listado 2.3.

Por otro lado, con <receive> el proceso puede recibir mensajes. Si un elemento <receive> tiene el atributo *createInstance* a *yes*, cuando llegue el mensaje indicado se crea una nueva instancia de la composición. En caso contrario, se asocia el mensaje entrante con la instancia del proceso correspondiente según el campo de correlación del mensaje. Además, una instrucción <receive> puede llevar asociada una actividad <reply> para proporcionar una respuesta a un cliente. En las líneas 18 y 19 del listado 2.3 se observa una instrucción de recepción de mensaje que inicia la ejecución del proceso WS-BPEL.

Por su parte, la instrucción <reply> puede proporcionar bien una respuesta con datos o un código de fallo para indicar una situación excepcional. El listado 2.3 muestra un ejemplo en el que se da la respuesta final de la composición al cliente (líneas 55 y 56).

Uso de XPath WS-BPEL usa XPath para indicar condiciones en las instrucciones que las aceptan (como en las líneas 23 a 25, que usa <lt para indicar el símbolo < en una instrucción <if>) y para asignar valores entre campos de variables (que puede verse en los diversos usos de <copy> en el ejemplo).

También hay que destacar que, a pesar de haber sido estandarizado en un proceso público por OASIS, no existe una descripción formal de WS-BPEL. Por lo tanto, como suele suceder con las descripciones en lenguaje natural, existen algunos aspectos algo ambiguos que diversos motores han implementado de distinta forma [Hal08, Dum05]. Sin embargo, ninguno de ellos afectará al estudio que tiene por fin esta tesis.

2.2. Prueba de software

En esta sección se comentan brevemente los fundamentos de la V&V. El primer apartado describe las principales clasificaciones de las técnicas de V&V. Posteriormente, se dedica un apartado a cada una de las técnicas más usadas en AOS.

2.2.1. Clasificación de las técnicas de prueba de software

Las técnicas de prueba se suelen dividir atendiendo a diversos criterios:

- Según si ejecutan o no código. Las técnicas estáticas son las que se realizan sin ejecutar el programa a probar, mientras que las dinámicas sí lo hacen.
- Dependiendo de si se usa información del código fuente del programa o no, se pueden dividir en técnicas de caja blanca o de caja negra. Las técnicas de caja blanca, son aquellas que se realizan accediendo al código fuente del programa. Las de caja negra se hacen sin tener acceso al código fuente del programa.

  Por lo general, las pruebas de caja blanca son más completas que las de caja negra. Sin embargo, estas segundas son las únicas que se pueden realizar si solo se dispone del código binario del programa a ejecutar.
- También se pueden clasificar según el tipo de validación que realicen sobre el programa. La *prueba funcional* comprueba el comportamiento correcto de un software a nivel funcional (que opere correctamente). Mientras que la *prueba no funcional*, como su nombre indica, no comprueba *qué* hace el programa, sino *cómo* lo hace. Esto suele implicar aspectos de usabilidad, portabilidad, restricciones temporales, facilidad de mantenimiento, etc.

2.2.2. Técnicas estáticas de prueba

En lo que respecta a las técnicas estáticas, hay varias que se basan en actividades humanas (como las revisiones de código, inspecciones, etc.). Estas tienen escaso margen

de automatización, por lo que solo se comentarán las que se puedan programar para comprobar determinadas propiedades en el código mediante modelos.

Algunos de los modelos más usados para prueba estática son las redes de Petri, las máquinas de estado finito o el cálculo-π. Estos modelos permiten encontrar interbloqueos, trozos de código que no se pueden llegar a ejecutar o variables no inicializadas, entre otros fallos. El uso de estos modelos formales produce resultados limitados pero prácticamente fiables [Hal90], siempre que el proceso seguido sea correcto. Debido a esta limitación en sus resultados, suele ser interesante complementarlas con técnicas dinámicas.

2.2.3. Generación de casos de prueba

La generación de casos de prueba tiene por objetivo crear conjuntos de entradas válidas para ejecutar el programa. Por lo tanto, es la base de las técnicas dinámicas de prueba, que necesitan dichos casos de prueba para ejecutar el software y hacer las pruebas correspondientes.

La técnica más básica para generar casos de entrada para un programa se denomina *generación de datos de prueba* (en inglés *test data generation*), y únicamente genera entradas para llamar al programa. Si a esa información se le añade los resultados esperados para cada entrada, se denomina *generación de casos de prueba* (en inglés *test case generation*). La generación de datos de prueba también puede incluir la creación de datos erróneos para probar si el programa se comporta como es debido (normalmente devolviendo un mensaje de error adecuado y volviendo a un estado consistente). Esta se suele llamar *prueba de robustez* (*robustness testing* en inglés).

La generación de casos en la prueba de caja negra está bastante limitada debido a que la única fuente de información es la interfaz del programa. Como consecuencia, desde una perspectiva formal, ofrecen escasas garantías de su validez. Por ejemplo, si se genera un conjunto de 1.000 casos de prueba para un componente, *¿ejecutarán esos 1.000 casos todas sus instrucciones? ¿Y todas sus ramas de ejecución?* Con el código fuente disponible se podría, por ejemplo, comprobar que todas las ejecuciones han ejercitado una misma rama muchas veces, mientras que otra (que quizás contenga un error) permanece sin ejecutar, manteniendo el error oculto. Por lo tanto, es fácil concluir que el uso de información sobre la implementación del software permite definir conjuntos de casos de prueba más completos.

Dado que el número de casos de prueba que se puede crear a partir de una definición WS-BPEL puede ser muy alto (e igualmente el tiempo y esfuerzo necesario para probarlos), un nuevo paradigma está emergiendo: generación de datos de prueba mediante búsqueda [Mcm04]. Este usa técnicas de búsqueda para evaluar el interés de diversos casos de prueba para determinados objetivos.

Los casos de prueba individuales se suelen agrupar en conjuntos que cumplen determinadas propiedades. Cada conjunto se denomina *test suite* en inglés, y en español

simplemente *conjunto de casos de prueba*. A continuación se explican los criterios más simples sobre el ejemplo del listado 2.4

Listado 2.4: *Pseudocódigo de ejemplo para coberturas.*

```
1  si  (X > 1)
2      X = 3
3  else
4      X = 4
5  fin_si
6  si  (Y > 1)
7      Y = 0
8  fin_si
```

Por ejemplo, puede ser que el conjunto asegure la ejecución de todas las instrucciones del programa. En ese caso se dice que dicho conjunto ofrece *cobertura de instrucciones* o *de sentencias* (en inglés *instruction coverage* o *statement coverage*), y es uno de los criterios de cobertura más débiles. En el ejemplo se puede conseguir con los siguientes casos de prueba: *(X = 2 , Y = 2)* y *(X = 1 , Y = 2)*.

Un criterio más general es el de *cobertura de ramas* o *cobertura de transiciones* [Fan06] (*brach coverage* o *transition coverage* [Off99] en inglés respectivamente), que asegura la evaluación a verdadero y falso de cada expresión condicional del programa. Esto implica que se deben de ejecutar todas las instrucciones del programa, por lo que aquel conjunto que cumpla cobertura de ramas cumple igualmente la de sentencias. Para ofrecer cobertura de ramas en el ejemplo basta con dos casos de prueba, uno con *(X = 2 , Y = 2)* y otro con, por ejemplo, *(X = 1 , Y = 1)*.

Subiendo en la jerarquía está el criterio de *cobertura de caminos* (en inglés *path coverage*), que ejecuta cada camino del programa al menos una vez. Esta cobertura es de las más completas que se definen, pero solo puede conseguirse en programas sin bucles, como es el caso del ejemplo. Para ofrecer cobertura de caminos en él hacen falta los casos de la cobertura de ramas y además *(X = 1 , Y = 2)* y *(X = 2 , Y = 1)*. Por la dificultad que tiene conseguir la cobertura de caminos, a veces se usa cobertura de caminos de longitud n (del inglés *path length*) [Ben09]. Esta cubre todos los caminos de ejecución con n o menos instrucciones (lo que facilita significativamente su obtención).

También hay otras coberturas, como la *cobertura de bucles*, que asegura la ejecución cero, una y más de una vez de cada bucle. Es evidente que cuanto más general sea un criterio, mejor será la prueba que realice del programa. Sin embargo, también será más difícil de conseguir (a veces, determinados criterios no son posibles de conseguir dependiendo del programa). En la literatura sobre prueba de software existen muchas propuestas para la generación y evaluación de conjuntos de casos de prueba [Mye04, Ram07, Zhu97].

Todos los criterios de cobertura comentados hasta este punto se denominan *criterios estructurales*, pues atienden a la cobertura que hacen del flujo de ejecución del programa. Sin embargo, también existen otros criterios centrados en el flujo de datos [Zhu97], como el de cobertura de los caminos de definición a uso de variables (del inglés *all-du-*

path) [Rap85], que incluye todos los caminos desde toda definición de variable hasta todo uso que se haga de ella en el programa.

2.2.4. Prueba de mutación

Otra técnica clásica en la generación de conjuntos de casos de prueba es la *prueba de mutación*. Esta técnica se ha usado desde hace más de treinta años para medir la calidad de conjuntos de casos de prueba [Woo93].

Se basa en la definición de un conjunto de operadores de mutación para un lenguaje concreto. Estos operadores reciben como entrada un programa y producen *mutantes*. Un mutante es un programa idéntico al original, pero que incluye intencionadamente un error. Los errores que modelan los operadores imitan errores típicos que puede cometer un desarrollador al codificar un programa [Jia10]. Un operador de mutación clásico es el cambio de un operador matemático en una expresión aritmética, modificando la expresión $x = y + 1$ por $x = y - 1$, por ejemplo.

Las versiones mutadas del programa se ejecutan posteriormente bajo un conjunto de casos de prueba. La salida de cada uno de ellos se compara con la que produce el programa original con la misma entrada. Si se aprecia alguna diferencia, se dice que el caso de prueba *mata* al mutante (detectando el error), siendo un indicador de bondad del conjunto.

2.2.5. Prueba unitaria

La *prueba unitaria* comprueba el comportamiento correcto de un software de manera independiente al resto del sistema. Por contra, si se prueba el sistema completo con el software integrado se llama *prueba de sistema*. También existe la *prueba de integración*, que prueba si un elemento software se integra correctamente con el resto del sistema. Sin embargo, el uso de la prueba de integración es prácticamente nulo en AOS, pues el uso de estándares evita problemas en esa fase.

Una de las cuestiones abiertas más importantes para llevar a cabo la prueba funcional es el *problema del oráculo*. Un oráculo es un sistema que dado un programa y una entrada, indica la salida que este debe dar. Existen muchas propuestas en la literatura, aunque ninguna ha demostrado ser netamente mejor que el resto [Dav81]. Los oráculos humanos se usan a menudo porque son sencillos de implementar (basta con contratar personal), pero son lentos para pruebas masivas y propensos a errores a medida que la complejidad del sistema aumenta.

La *prueba de robustez*, trata el comportamiento de un programa cuando recibe una entrada incorrecta. Normalmente se crean los datos, se ejecuta el programa y se comprueba su salida en cada caso para detectar inconsistencias en su funcionamiento.

2.2.6. Prueba metamórfica

La *prueba metamórfica* puede aplicarse a problemas concretos para mitigar el problema del oráculo. Para ello crea nuevos casos de prueba a partir de casos de prueba anteriores [Zho04].

Esto se realiza partiendo de una cierta propiedad que relaciona las entradas y salidas de un programa, de modo que a partir de un conjunto de datos de entrada puede generar datos de entrada nuevos y calcular su salida basándose en una relación entre ellas conocida. A veces se puede lograr la salida exacta del programa, pero a veces se obtiene un conjunto de condiciones que difícilmente podrá cumplir la salida si no es el resultado correcto (este segundo caso es típico de problemas matemáticos cuya salida exacta es difícil de obtener). Su principal inconveniente es que no siempre es fácil encontrar una relación entre las entradas y salidas de un programa, lo que lo limita principalmente a problemas matemáticos.

2.2.7. Prueba por referencia

Otra solución robusta para el problema del oráculo es la *prueba por referencia*. Esta se puede aplicar cuando existe una versión implementada del programa que se puede usar como oráculo. Por lo general esto es debido a que un programa se desea reimplementar en otra arquitectura o actualizarlo (como al hacer disponible un programa de consola como SW).

2.2.8. Prueba de regresión

De modo similar, la *prueba de regresión* (*regression testing* en inglés) se realiza al modificar un programa (por ejemplo, para implementar una funcionalidad nueva, reimplementar parte del sistema o incluso reparar errores).

La experiencia demuestra [Mye04] que la modificación de un programa suele propiciar la aparición de errores en él (ya sean errores previos que estaban sin descubrir, errores de integración o errores del nuevo código), sobre todo en sistemas grandes. En estos casos, una cuidadosa selección de casos de prueba disponibles para la versión anterior del programa, seleccionando entre aquellos que siguen siendo válidos para la nueva versión, la eliminación de los que ya no son válidos y la incorporación de otros que prueben el código nuevo, suele producir buenos resultados.

2.2.9. Prueba con invariantes generados dinámicamente

Otra técnica que puede usarse para ayudar a mitigar el problema del oráculo es la generación dinámica de invariantes [Ern01, Ern07]. 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 detectar propiedades observadas en ella.

Por lo tanto, para obtener invariantes que permitan detectar errores es necesario un conjunto de trazas de ejecución que reflejen adecuadamente la complejidad interna de un programa. De este modo, un invariante inesperado puede indicar un error en el programa, evitando la necesidad de disponer de un oráculo que indique la salida en cada caso de prueba. Además, el invariante indicaría aproximadamente en qué punto del programa se ha detectado el error y a qué variables afecta. Igualmente, se puede realizar prueba de regresión comparando los invariantes que generan dos versiones de un programa.

Esta técnica es la que se desarrolla en esta tesis, por lo que se describe en detalle en el capítulo 3.

2.2.10. Prueba no funcional

Como su nombre indica, no comprueba *qué* hace el programa, sino *cómo* lo hace. Esto suele implicar aspectos de usabilidad, portabilidad, restricciones temporales, facilidad de mantenimiento, etc. Muchas de estas características son difíciles de aplicar a los SW, por lo que en el estudio bibliográfico solo se considerará la *prueba de carga* (*workload testing* en inglés), que comprueba la cantidad de peticiones en paralelo que soporta un servicio, y restricciones temporales en la respuesta de los SW.

2.3. Prueba de servicios web

En este apartado se presentan las soluciones disponibles actualmente para probar un SW externo, cuyo código fuente no está disponible. Nótese que dicho SW puede estar implementado internamente en un lenguaje clásico, o podría ser una composición WS-BPEL. Por lo tanto, las técnicas que se comentan a continuación son válidas para la prueba de caja negra en WS-BPEL.

Una de las características más interesantes de los SW es que se pueden usar a través de la red de manera independiente de la plataforma hardware, el sistema operativo y el lenguaje de programación usados tanto en el servidor como en el cliente. De esta forma, se pueden construir fácilmente sistemas complejos basándose en servicios distribuidos. Sin embargo, implican un reto para la prueba de software por diversos motivos. En primer lugar, independientemente de ninguna característica propia de las AOS, el ingeniero de prueba no tiene acceso al código fuente del sistema por lo general, lo que limita las técnicas de prueba aplicables a las pruebas dinámicas de caja negra (técnicas que se basan en la ejecución del software).

Además, existen otros problemas específicos de los SW. Por ejemplo, cuando un programa llama a un servicio que ya llamó previamente, *¿se puede afirmar que es el mismo software el que atiende la petición?* El programa solo conoce la definición WSDL del SW y su URI, pero el software que responde a las peticiones podría haber cambiado su implementación interna desde la anterior llamada, lo que podría producir un comportamiento distinto.

Es más, por norma general, no se debe suponer que para realizar prueba de software se puedan hacer tantas llamadas al servicio como se deseen. Esto puede deberse a que las llamadas tengan asociado un coste (por ejemplo, en un servicio de transacciones bursátiles), o puede ser que bloqueen recursos valiosos (por ejemplo, en un servicio de reserva de vuelos), o simplemente tarden demasiado y monopolicen un recurso (como en un servicio de cálculos matemáticos complejos). A veces, el proveedor del servicio, intenta mitigar estas limitaciones ofreciendo una versión «de prueba» del servicio, sobre la que no existen dichas restricciones. Sin embargo, no siempre está disponible, y aún cuando lo está, su comportamiento no suele ser igual al del servicio real.

En esta sección se supone que el SW a probar no tiene problemas de compatibilidad para usarse (por ejemplo, porque cumple algunos de los perfiles de interoperabilidad del WS-I [Org], *Basic Profiles* en inglés). Cada uno de estos perfiles comprende varias normas, y suelen estar soportados por la mayoría de productos para AOS. Se supondrá que se desea probar el SW de manera exhaustiva antes de usarlo en producción. Para ello se atiende a cuatro criterios: generación de datos de prueba, prueba unitaria, prueba de regresión y prueba no funcional.

2.3.1. Generación de casos de prueba

La generación de casos de prueba tiene por objetivo la creación de conjuntos de datos de entrada que puedan usarse para probar un servicio. Para realizar la prueba de un SW independiente y externo, lo único que se conoce es su interfaz pública descrita en un documento WSDL (excepto en el caso de SW semánticos, pero a día de hoy no están implantados de forma general).

Por lo tanto, la aproximación más sencilla es la generación aleatoria de datos de prueba a partir de documentos WSDL. Para ello se dispone de la herramienta WSDL-Test [Sne07], y el programa privativo Parasoft SOAtest [Par]. Ambos pueden comprobar opcionalmente si los mensajes de salida del servicio cumplen determinadas propiedades que deben introducirse a mano. La ventaja de este método es que es totalmente automatizable para cualquier SW. Como complemento, se pueden generar datos de prueba para causar fallos en la entrada del servicio y así descubrir inconsistencias en las descripciones WSDL con el framework WebSob [Mar07b].

Otra propuesta complementaria es generar datos de prueba según dependencias que se deban dar entre distintas operaciones de un mismo servicio con su estado interno [Bai05]. Por ejemplo, la compra de un determinado artículo en un SW de venta on-line probablemente dependa de una operación previa de llegada de stock. De este modo se pueden probar efectos colaterales de la invocación de un SW. La herramienta libre soapUI [Evi] permite definir casos de prueba que dependan de ejecuciones previas de otros. Este sistema está desarrollado por la empresa Eviware, que ofrece soporte y versiones privativas del producto con funcionalidades extras.

2.3.2. Prueba unitaria

El objetivo de la prueba de unidad es asegurarse de que el SW funciona correctamente antes de ser integrado en un sistema. La mayoría de las técnicas y herramientas que se han comentado en la sección anterior generan datos de prueba que se pueden usar como entrada de un servicio. Pero solo permiten comprobar de manera automática propiedades generales que cumple la salida del SW. Para superar estas limitaciones y evitar problemas derivados de usar una persona para introducir dichas propiedades, es necesario implementar una alternativa que solucione el problema del oráculo.

Para ello, las diferentes propuestas tienen que usar cierta información adicional sobre el servicio. Por ejemplo, pueden usar contratos que el mismo proveedor de servicios proporcione [Hec04], hojas de pruebas [Atk08], extensiones de las definiciones WSDL del SW [Tsa02], o usar información semántica que el servicio incluya [Dai07].

La prueba de robustez se contempla en el *framework* WebSob [Mar07b]. Este genera automáticamente datos de entrada válidos e incorrectos para un SW, y comprueba su salida en cada caso para detectar inconsistencias en su funcionamiento. Además, permite comparar resultados previos para implementar prueba de regresión.

2.3.3. Prueba de regresión

La prueba de regresión está bastante más limitada en el software *clásico* que en las AOS. La razón es que al ejecutar un programa, por lo general, este usa bibliotecas que están instaladas en el sistema. Por lo tanto, al aparecer una nueva versión de dichas bibliotecas, el programa sigue usando la versión anterior, de modo que su comportamiento no se ve afectado hasta que el desarrollador decida instalar la nueva versión de dicha biblioteca. Sin embargo, en las AOS este problema es crítico, pues un servicio puede cambiar su implementación interna en cualquier momento. Esto provoca que los usuarios (tanto usuarios finales como integradores de servicios) deben tener especial cuidado con los problemas que se puedan producir.

Una de las propuestas más completas para la prueba de regresión de SW, aparte del framework WebSob, es el proyecto SeCSE [Mel]. Este proyecto implementa herramientas libres para sistemas centrados en servicios (del inglés *service-centric systems*). Para realizar la prueba de regresión, almacena la información que envía y recibe de cada servicio, minimizando de este modo el consumo de recursos. Igualmente propone añadir casos de prueba a la descripción de los servicios, de modo que puedan usarse para la prueba de regresión [Bru05].

Llegado este punto hay que destacar que el ingeniero de prueba debe tener en cuenta que, dependiendo del servicio, la prueba de regresión ha de ser flexible. Por ejemplo, la respuesta de algunos servicios pueden depender del momento de su invocación: servicios de búsqueda de información (por ejemplo Google no tiene por qué responder lo mismo a una búsqueda hoy que dentro de un mes), servicios de predicción meteorológica (que trabajan con fechas futuras y cuya predicción puede cambiar a medida que se acerca la fecha), información sobre valores reales en bolsa, servicios que informen de

clasificaciones que cambien, etc. En todos estos casos, un comportamiento distinto del SW ante una misma entrada no tiene por qué implicar un error en el sistema.

2.3.4. Prueba no funcional

En AOS, el principal aspecto de la prueba no funcional es la prueba de carga, que normalmente se trata como un aspecto más de *calidad de servicio* (*Quality of Service*, QoS en inglés). Suele ser de importancia en sistemas con restricciones temporales, por lo que es muy común que los usuarios de SW contraten un acuerdo de calidad de servicio (del inglés *Service Level Agreement*). Este aspecto lo consideran las herramientas ParaSoft SOAtest y soapUI anteriormente comentadas.

El proyecto SeCSE propone usar casos de prueba como contratos entre proveedor de SW y cliente [Bru05] que sirvan para medir características de calidad de servicio de nuevas versiones de un SW. Dicha propuesta [SeC08] usa la información almacenada para la prueba de regresión (datos de diversas invocaciones a SW y las respuestas obtenidas) para generar invariantes. Analizándolos se puede evaluar el cumplimiento del contrato de calidad de servicio.

Por último, también destaca un sistema para la gestión de la calidad relativa a contratos de calidad de servicio [Yeo09]. Permite la creación y despliegue de contratos de calidad de servicio, registro de SW y monitorización para la detección y notificación de violaciones en ellos.

2.3.5. Resumen

Las técnicas que se han comentado en esta sección se pueden aplicar a cualquier SW. Pertenecen al ámbito de la prueba de caja negra, por lo que pueden aplicarse a un SW independientemente de que se tenga acceso a su implementación interna o no [Boz10]. Dado que no hay acceso al código fuente del servicio, principalmente comprueban que el servicio se comporte de acuerdo a su especificación generando conjuntos de datos de entrada (generación de casos de prueba), de modo que se puedan hacer comprobaciones sobre el resultado de su ejecución (prueba de unidad) en aspectos funcionales o no funcionales, y verificar que los cambios internos que pueda sufrir no afectan a su comportamiento externo (prueba de regresión).

En el siguiente apartado se comentan propuestas complementarias que prueban SW programados sobre otros SW, y que son aplicables cuando se dispone de su código fuente. Estas técnicas realizan una prueba más exhaustiva del SW usando información de su implementación.

2.4. Prueba de composiciones de servicios web

Gracias a su independencia de la plataforma y sistema operativo, los SW no solo pueden llamarse desde programas tradicionales, sino que también pueden ser compuestos

de manera sencilla para crear otros SW de mayor capacidad, que satisfagan las necesidades de los clientes [New04] (lo que a veces se denomina *programación a gran escala*, *programming-in-the-large* en inglés). Pero al componer servicios, probar los SW que se invocan no es suficiente para asegurar la calidad del nuevo servicio que se compone: también hay que probar la lógica interna de la composición cuidadosamente.

La lógica interna de los SW compuestos puede ser sencilla. Por ejemplo, un envoltorio de un sistema antiguo con comunicación limitada (del inglés *legacy*), o un adaptador que añade o modifica cierta información a la respuesta de un SW y la envía al cliente. Pero también puede ser muy compleja: puede modelar actividades de negocio o SW críticos usando bucles, manipulación de datos, actividades de gestión de fallo, etc. [Pap07]. De este modo, un error que no se detecte en el código puede producir grandes pérdidas económicas, sobre todo si la composición (que es un SW por sí misma) se llama no solo por usuarios finales, sino también automáticamente desde otros programas o servicios.

Este apartado parte de la suposición de que existe una composición de SW en WS-BPEL lista para ser desplegada y llamada, pero que tiene que ser probada. Se revisan las técnicas tradicionales de prueba de software, observando si hay herramientas para aplicarlas a composiciones WS-BPEL, o al menos propuestas que digan cómo hacerlo. Tras revisar las propuestas para orquestación, se dedica una subsección a recopilar las iniciativas más interesantes en el campo de la coreografía.

2.4.1. Técnicas estáticas para la prueba en WS-BPEL

Atendiendo a las técnicas estáticas [Mor08], se puede observar propuestas que usan redes de Petri, máquinas de estado finito y cálculo-π entre otras. Todas están basadas en una cierta formalización de la especificación de WS-BPEL, formalización necesaria porque el estándar está escrito en lenguaje natural (inglés), que por su propia naturaleza no es riguroso y no incluye una especificación rigurosa [Ask04, Hal10]. Dependiendo del formalismo por el que se opte, se podrán realizar distintas operaciones y comprobaciones de propiedades [Bre06], aunque es igualmente importante evitar simplificaciones excesivas que obvien aspectos de interés de la composición [Buc07].

Las redes de Petri implementan un flujo paralelo basado en estados de un grafo, en el que es fácil comprobar propiedades sobre ellas [Pet77]. Por ello, encajan bien con la definición de una composición WS-BPEL [Loh06], lo que ha llevado a que se desarrollen muchos trabajos que las usen. Uno de los más interesantes realiza una traducción completa y automática desde cualquier composición que cumpla el estándar WS-BPEL 2.0 a una red de Petri extendida, optimizando la red resultante para minimizar su tamaño [Loh08a]. Esa traducción se realiza a través de BPEL2oWFN, una herramienta libre que produce ficheros oWFN (*open WorkFlow Net*) que se usan en dicho trabajo, además de PNML, PEP, LoLA, INA y SPIN. De este modo, puede comprobar si un flujo WS-BPEL cumple determinadas propiedades expresadas en lógica temporal. Existen otras propuestas con redes de Petri en [Loh09].

Las técnicas de máquinas de estado finito traducen un programa a un lenguaje formal en que puede comprobarse de manera automática determinadas propiedades, usando una herramienta que lo soporte [Cla99]. Hay un trabajo que traduce composiciones WS-BPEL a un modelo formal definido por un sistema de transiciones de estados, que después se procesa usando la herramienta SAL (*Symbolic Analysis Laboratory*). Las traducciones necesarias son automáticas gracias al framework libre VIATRA [Kov07, Kov08]. Las propiedades se indican en LTL (*Linear Temporal Logic*), que puede expresar propiedades usadas en lenguajes clásicos (del tipo «el programa termina toda ejecución», «toda variable que se lee ha sido inicializada con anterioridad», etc.) y otras específicas de composiciones de SW: comprobar si existen determinados estados que no se pueden alcanzar (en inglés *safety*) y que determinados estados sí se pueden alcanzar (en inglés *reachability*).

También hay un trabajo que usa *máquinas abstractas de estados multiagentes* (*Abstract State Machine*, ASM en inglés) [Fah05]. Estas modelan cada actividad WS-BPEL como un agente independiente. De este modo, la composición se puede modelar por completo definiendo las interacciones entre ellos. La propuesta se centra en lo que define como el *flujo de ejecución negativo* (*negative workflow* en inglés), esto es, ejecución de instrucciones para compensar fallos. Sin embargo, parece que no se ha profundizado en el trabajo en este campo: queda como línea futura obtener más resultados a partir del modelo.

El cálculo-π es un álgebra de proceso formal que se usa para comprobar propiedades en sistemas concurrentes. Hay muy pocos trabajos sobre WS-BPEL en este campo [Abo06, Luc07]. Una propuesta de traducción de WS-BPEL 2.0 a cálculo-π se usa para verificar la compatibilidad de los mensajes intercambiados entre determinados procesos en una composición y la consistencia de esta respecto a su especificación [Wei07]. Justifica que una traducción completa de WS-BPEL a un modelo no es necesaria para verificar según qué propiedades, y que descartar determinadas características puede producir un modelo más sencillo y fácil de manejar.

En este apartado se han revisado diversas propuestas formales para verificar determinadas propiedades de una composición WS-BPEL. En [Mor08] se estudian estas y otras propuestas. Como se ha comentado previamente, las técnicas estáticas producen resultados totalmente fiables, pero limitados para la correcta V&V de una composición WS-BPEL. En las próximas secciones se analizan propuestas dinámicas complementarias.

2.4.2. Generación de casos de prueba en WS-BPEL

Como se ha comentado anteriormente, la generación de datos de prueba para SW está limitada a la información disponible en el documento WSDL que lo describe. Por lo general solo pueden generar datos de prueba, pero no casos de prueba completos, y su calidad difícilmente puede evaluarse. Por contra, cuando el código fuente de la composición WS-BPEL está disponible, las distintas técnicas pueden producir mejores

resultados. Normalmente intentan generar conjuntos de casos de prueba que proporcionen un criterio determinado de cobertura [Mye04, Ram07].

Existen propuestas que usan SPIN, un sistema libre que ha sido utilizado para verificar programas escritos en diversos lenguajes [Hol97]. SPIN recibe como entrada un programa en lenguaje Promela y un conjunto de fórmulas LTL que comprobará. En [Fan06] se usa SPIN para proponer un método no automático de generación del conjunto de casos de prueba mínimo que proporciona cobertura de transiciones. Una propuesta similar es la de [Ben09], que se basa en *Sistemas de transiciones simbólicos* (*Symbolic Transition Systems* en inglés) para generar un conjunto de casos de prueba que proporcionen cobertura de caminos de longitud n.

Además, hay un sistema que implementa generación automática de conjuntos de casos de prueba compatibles con JUnit que proporcionan cobertura de estados y transiciones para la prueba del flujo de control en WS-BPEL, y cobertura de caminos de uso de variables (*all-du-path*) para la prueba del flujo de datos de WS-BPEL [Zhe07]. Estas funcionalidades están implementadas en un asistente libre para Eclipse. Pero lamentablemente no soporta WS-BPEL 2.0, y el proyecto parece abandonado. Fue subvencionado por el Programa Marco FP6 de la Unión Europea dentro del proyecto *Digital Business Ecosystem* [Nac07] hasta 2007. El proyecto fue renovado posteriormente, pero tomó una orientación distinta [Ass10].

Por otro lado, WSOTF es una herramienta libre que puede crear, ejecutar y depurar casos de prueba usando un modelo TEFSM (*Timed Extended Finite State Machines*) que permite expresar restricciones temporales [Cao10]. WSOTF se puede considerar como una mejora del framework "off-line" TGSE. Como punto negativo, necesita que la composición WS-BPEL de entrada se traduzca al formato de WSOTF, y la escasa documentación que ofrece en su web oficial que solo está disponible en francés [odt10].

La única propuesta que existe para WS-BPEL [Bla09] de generación de casos de prueba mediante búsqueda usa búsqueda dispersa (*scatter*), una técnica metaheurística basada en un algoritmo evolutivo que busca la solución óptima a un problema. No consta que esté implementada todavía, por lo que queda como una línea de trabajo futura la implementación de esta y otras técnicas de solución a problemas de prueba y otros problemas de ingeniería del software en general mediante búsqueda [Gut08].

2.4.3. Prueba unitaria en WS-BPEL

Al contrario que en la prueba unitaria clásica, los entornos de prueba unitaria de WS-BPEL tienen que incluir un mecanismo de simulación de SW por varias razones. La primera es para evitar que un fallo en un servicio llamado (que funcione mal) provoque un error en una composición correcta. Además, puede ser que los SW no estén disponibles para pruebas o que simplemente se desee probar la composición ante determinadas respuestas de algunos servicios. Sin embargo, es importante que dichos servicios suplantadores sean lo suficientemente flexibles para simular todos los aspectos del comportamiento del SW que pueden afectar a la lógica interna de la composición.

Estudiando las posibilidades de prueba que ofrecen la mayoría de entornos de desarrollo industriales, se observa que solo incluyen técnicas muy sencillas. Por ejemplo, Oracle BPEL Process Manager es probablemente el sistema de orquestación más potente en la actualidad, y cuenta con una gran aceptación empresarial. Sin embargo, en lo que se refiere a la prueba de composiciones, solo incluye un entorno para ejecutar casos de prueba que invocan SW reales o simulados (que solo pueden responder constantes) [Ora07]. Incorpora soporte de llamadas masivas automáticas, puede comprobar asertos sobre su salida y genera un informe sencillo sobre la cobertura de instrucciones. El escenario es similar en la mayoría de sistemas, siendo Parasoft SOAtest el más completo, pues incluye diversas herramientas para prueba en una solución [Par].

Por otro lado, existen propuestas del mundo académico para la prueba unitaria de WS-BPEL [Zak09]. Una de las más interesantes es el sistema libre BPELUnit [Eng10]. BPELUnit es un sistema totalmente funcional que facilita la prueba unitaria de WS-BPEL, y que incluye realización automática de pruebas masivas desde consola, desde una extensión (*plug-in*) de Eclipse, o desde un guión (*script*) de Apache Ant. Incluye simulación de SW y asertos que se comprueban en los mensajes intercambiados.

El proyecto BPELUnit lo mantiene actualmente Daniel Lübke [Lüb07]. Sin embargo, durante cierto tiempo el proyecto permaneció estancado en la versión 1.0 (liberada en junio de 2007). Pero en abril de 2009 se publicó la 1.1.0 y, desde entonces la actividad se ha incrementado de nuevo y su web oficial se ha reconstruido [dt10b]. Está muy orientada a la comunidad e incluye tutoriales, una lista de correo y un sistema GitHub para ayudar a su desarrollo continuo distribuido. Entre las mejoras recientes se incluyen generación automática de casos de prueba basada en plantillas (de hojas de cálculo Excel u OpenOfficeCalc entre otras fuentes), cálculo de la cobertura de un conjunto de casos de prueba (por el momento cobertura de instrucciones, pero en un futuro próximo está previsto otras coberturas), y simulación de SW más realista. Estos SW simulados permiten que los casos de prueba definan una serie de comportamientos que se dispararán dependiendo del mensaje recibido en cada ejecución.

Todas estas características hacen de BPELUnit una herramienta destacada a tener en cuenta: es estable, independiente del motor (soporta diversos motores WS-BPEL tanto privativos como libres) y su desarrollo parece que está respaldado por una comunidad bastante activa.

2.4.4. Prueba de mutación en WS-BPEL

El único sistema para la prueba de mutación de WS-BPEL disponible a día de hoy es GAmera [Dom09]. Está compuesto por dos subsistemas que pueden usarse de manera independiente: un sistema de mutación para WS-BPEL y un innovador generador de casos de prueba automático que se basa en un algoritmo genético que muta los casos de prueba (es el primer trabajo que propone esta combinación de técnicas). Solo el primero de los dos está implementado a día de hoy [Est10]. De hecho, recientemente se ha libe-

rado su versión 1.0.5, que reduce la cantidad de mutantes equivalentes que producían versiones anteriores.

Los operadores de mutación de GAmera no solo incluyen errores de programación típicos en lenguajes tradicionales, sino también otros específicos de WS-BPEL [Est08]. Estos tienen en cuenta que las composiciones se suelen programar usando un entorno gráfico, de modo que algunos errores típicos en otros lenguajes no se consideran. Mientas que se incluyen otros específicos de WS-BPEL, como seleccionar un SW erróneo al hacer una llamada, cambiar el tipo de error que lanza una actividad, o modificar el temporizador de determinadas actividades. GAmera es un sistema libre que puede descargarse de su web oficial [Grua].

2.4.5. Prueba de regresión de WS-BPEL

Existen dos propuestas para la prueba de regresión en WS-BPEL. La primera propone comparar las diferencias entre una nueva versión de una composición y su anterior para crear casos de prueba específicos que prueben las actividades que han cambiado [Liu07]. La segunda define varias técnicas específicas para WS-BPEL que permiten priorizar los casos de prueba de acuerdo con la cobertura de la composición que proporcionan [Mei09b]. Sin embargo, ninguno de ellas ofrece herramienta alguna que automatice el proceso, por lo que queda como una línea de trabajo futuro.

2.4.6. Prueba de coreografías de servicios

Debido a su escasa implantación industrial, hay pocos trabajos sobre prueba de coreografías de servicios [Buc07, Pal08a]. A continuación se comentan algunos de los más destacados.

Hay un trabajo que define diversos criterios de prueba de flujo de datos específicos para la coreografía [Mei09a]. Para ello hace uso de *sistemas de transiciones etiquetados* (*Labeled Transition Systems*, LTS en inglés), que modelan la interacción entre los distintos SW de una coreografía, y patrones de *grafos de reescritura XPath* (en inglés *XPath Rewriting Graphs*, XRGs) que modelan la consulta de datos que hace XPath en los documentos WSDL según la especificación de la coreografía.

Una técnica basada en modelos se implementa sobre *modelado de coreografías de mensajes* (*Message Choreography Modeling*, MCM), un lenguaje de coreografía desarrollado por SAP Research [Ste09]. La propuesta es interesante, porque realiza una traducción automática de MCM a modelos UML que se ejecutan en herramientas de prueba específicas de UML. Sin embargo, es necesario valorar las dependencias que implica y la madurez de esta herramienta que actualmente usa un lenguaje no estandarizado a la hora de adoptarla.

Por último, los autores de BPEL2oWFN han adaptado su herramienta para que pueda aplicarse a la prueba de composiciones *BPEL4Chor* [Loh08b]. BPEL4Chor es un lenguaje no estandarizado para coreografía de SW escrito sobre WS-BPEL. La herramienta

se usa para detectar interbloqueos en una coreografía, pero también se puede usar para comprobar otro tipo de propiedades más generales, como se comentó en la sección 2.4.1.

2.4.7. Resumen

Tras estudiar los distintos trabajos disponibles sobre V&V de composiciones de SW, se puede afirmar que aún existen retos en el futuro para proporcionar buen soporte de prueba en WS-BPEL. Las herramientas industriales incluyen soporte muy limitado (principalmente prueba funcional básica). Mientras, el mundo académico está proponiendo alternativas en distintas direcciones, adaptando técnicas y afrontando retos. Pero en muchos casos sus resultados no tienen soporte automático implementado en herramientas, por lo que queda como siguiente paso su integración en las herramientas líderes de desarrollo WS-BPEL. En otros casos hay muy poco trabajo hecho, como en la prueba metamórfica, donde la única propuesta que se ha encontrado implementa las composiciones de servicio en C++ [Cha07], un lenguaje no del todo adecuado para ello.

2.5. Líneas de trabajo futuro

Además de los diversos retos discutidos en cada campo, destacan cuatro frentes que hay que afrontar en general.

Para empezar, la V&V de composición de SW debería centrarse en proporcionar soporte completo al estándar WS-BPEL 2.0. El estándar cuenta con el respaldo de las principales empresas del sector TIC y herramientas AOS [Dom07]. Además, parece ser la base de otras tecnologías futuras: BPMN, BPEL4People, extensiones semánticas, ejecución de composiciones de SW dinámicas, BPELScript, etc. Desde una perspectiva investigadora, los resultados obtenidos en BPEL4WS 1.1 son prácticamente tan válidos como los logrados en WS-BPEL 2.0. Pero desde un punto de vista industrial, los segundos son absolutamente necesarios para asegurar la compatibilidad entre sistemas.

Por otro lado, las herramientas deberían intentar facilitar su uso proporcionando (cuando fuera posible) dos modos de operación: totalmente automático e interactivo. En primer lugar, el modo automático permite usar la tecnología desde otras herramientas, facilitando su uso masivo e interoperabilidad para construir sistemas más complejos sobre ellos. El modo automático puede implementarse como una biblioteca, de modo que se pueda compilar en los principales lenguajes de programación, una interfaz de *shell* que incluya muchas opciones, o incluso como un servicio web que pueda recibir llamadas remotas. Por otro lado, el modo interactivo debe ser todo lo sencillo de instalar y usar que se pueda, permitiendo una interacción intuitiva con el usuario sin que este necesite mucho tiempo de adaptación. Puede hacerse como una herramienta independiente (en ese caso parecen interesantes lenguajes de programación multiplataforma, como Java, Python o C++) o como una extensión para alguno de los entornos de desarrollo más usados: Eclipse o Netbeans. En ambos modos la herramienta debe implementar las traducciones necesarias a partir de ficheros estándares (WSDL, WS-BPEL,

etc.) a los distintos modelos internos, facilitando así su implantación en empresas TIC. Un buen ejemplo en este sentido es el sistema BPELUnit.

En tercer lugar, las propuestas académicas deberían hacer un esfuerzo por aprovechar la fuerza de la comunidad. Una web sencilla que ofrezca información sobre el sistema (como capturas de pantallas o vídeo-tutoriales que permitan a los visitantes ver el sistema sin tener que instalarlo) y proporcionar un cierto soporte en foros o listas de correos, puede contribuir mucho a su difusión. Adicionalmente, la web se puede complementar con utilidades como wikis, sistemas de compartición de transparencias, seminarios por la web, etc. Igualmente destacan las ventajas de un modelo basado en una licencia libre, que sacrifica el beneficio a corto plazo que ofrece la venta de licencias por contratos de mantenimiento más duraderos y la creación de una comunidad que aporte al proyecto. Así también se facilita su adopción en la industria.

Por último, a largo plazo es necesaria la definición de metodologías dirigidas que proporcionen soporte completo. La tecnología de composición de SW es reciente, y poco esfuerzo se ha puesto en la inclusión de prueba automática en metodologías de desarrollo para AOS [Gar09]. Lo mismo sucede con las principales técnicas de desarrollo que están apareciendo, como el *desarrollo dirigido por prueba* [Bec02].

Además de los retos en investigación y desarrollo, hay que destacar la escasez de ejemplos públicamente disponibles para probar una herramienta o técnica. La mayoría de SW tienen un coste de uso, lo que los hace inapropiados para evaluar una técnica. Por su parte, la mayor parte de las composiciones implementan modelos de negocios (a veces con toma de decisiones), lo que provoca que por lo general las empresas sean reacias a publicarlas (o solo publican parte de su contenido). Pero hace falta buenos bancos de prueba (*testbeds*) para medir la calidad de las técnicas de prueba, y hay muy poco disponible en este sentido. En lo que respecta a los SW, hay algunos sitios web como [XMe10, Web10] que recopilan SW disponibles al público gratuitamente. Y en cuanto a composiciones WS-BPEL, hay una colección de composiciones recopiladas de diversos artículos [Gru10], y un par de repositorios web con composiciones de ejemplo usadas en la prueba de los motores WS-BPEL Apache ODE [dt10a] y WSO2 [Tan10].

2.6. Conclusiones

En este capítulo se han revisado las distintas soluciones desarrolladas hasta ahora para la prueba de SW y composiciones de SW. En lo que respecta a la prueba de SW, por lo general, la industria ofrece herramientas maduras y útiles que implementan diversas técnicas. Pero en cuanto a composiciones de SW, las empresas han hecho pocos esfuerzos más allá de una prueba funcional básica, por lo que es necesario consultar propuestas académicas.

Mirando en perspectiva esta es una situación que cabía esperar: los SW se llevan usando desde hace algunos años, y las empresas han ido mejorando sus sistemas e incluyendo herramientas para su prueba. Pero la composición de SW es una tecnología bastante más reciente: el estándar WS-BPEL se aprobó en 2007 (aunque se estaba traba-

jando en la propuesta desde 2006), mientras que en cuanto a coreografías, no parece que a corto plazo ningún estándar vaya a recibir soporte sustancial de la industria.

Sin embargo, parece que los esfuerzos académicos están mostrando las líneas de trabajo para las futuras herramientas de prueba. Hay trabajos con técnicas estática y dinámica de caja blanca de composiciones de SW: prueba unitaria, prueba de mutación, y generación de casos de prueba entre otros. Pero queda el reto de hacer las propuestas (tanto académicas como industriales) fáciles de usar e implantar, soportando por completo el estándar WS-BPEL 2.0, e integrarlo en las principales metodologías de desarrollo.

CAPÍTULO 3

Prueba de software con invariantes

En este capítulo se presentan las principales características de la prueba de caja blanca con invariantes, se comentan los distintos métodos de generación de invariantes y la utilidad que tienen estos invariantes en la mejora del software.

Como se ha explicado en el capítulo 1, en este documento se utilizan los términos «invariante» e «invariante potencial» en el mismo sentido en que se usa en la bibliografía de la materia [Ern01, Ern07, Gup03], 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.

3.1. Generación de invariantes

Un invariante es una propiedad que es cierta en un determinado «punto de un programa». Por cada instrucción *i* del programa se definen dos puntos del programa: el punto de programa *antes de i* identifica los invariantes que se cumplen justo antes de ejecutar *i* y *después de i* aquellos inmediatamente después de su ejecución. Hay que destacar que si *i* es una instrucción de bloque, como un bucle o condicional, pueden existir otros puntos de programa entre el antes y después de esa instrucción.

Ejemplos clásicos de invariantes 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.

Se pueden ilustrar estos conceptos con un sencillo ejemplo. Supóngase que se desea sumar todos los enteros desde 1 hasta un determinado entero positivo *n* inclusive. Se podría definir un sencillo algoritmo que lo hiciera, como el del listado 3.1.

Listado 3.1: *Pseudocódigo del sumatorio.*

```
1 calculaSumatorio (n: entero)
2 r = 0
3 Desde i = 1 hasta n:
4     r = i + r
5 Devolver r
```

Este algoritmo tiene la pre-condición $n > 0$ en el paso 1, dado que n es positivo por definición. En el paso 3, se puede 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 5 ya se ha salido del bucle, se tiene que $i = n + 1$. Sustituyendo en el invariante de bucle anterior se obtiene la post-condición del paso 5 y, por extensión, de todo el algoritmo: $r = \sum_{j=0}^{n} j$. A partir de esta post-condición es posible afirmar que el algoritmo realmente hace lo que se espera de él.

3.1.1. Uso de invariantes

Los invariantes generados a partir de un programa pueden usarse de diversas formas para mejorarlo [Ern07]:

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

Depuración de errores Un invariante inesperado puede hacer 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. En general, los invariantes pueden ayudar a localizar e identificar los fallos de un programa.

Evitar la inclusión de errores al modificar un programa Es muy común que la modificación de un programa (ya sea para ampliar sus capacidades, reparar errores, etc.) introduzca nuevos errores en él. Errores que pueden presentarse incluso en partes del código que no han sido modificadas [Gra00]. El uso de invariantes puede ayudar a evitar que se introduzcan nuevos errores al modificar 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 fuente del programa, así los desarrolladores, responsables de prueba y otro personal que tenga acceso a él podrán consultarlos en su trabajo. Está demostrado que la inclusión de invariantes generados automáticamente es de utilidad incluso aunque el código ya esté documentado e incluya asertos (pues dicha información puede estar obsoleta o incluso ser errónea) [Lev90].

3.1.2. Generación automática 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 [Bjø97, Ern01].

Básicamente, existen dos tipos de generadores automáticos de invariantes: estáticos y dinámicos. Los primeros infieren los invariantes a partir del código fuente del programa, mientras que los segundos lo hacen con información de ejecuciones.

Los generadores estáticos de invariantes [Bjø97, Col03] 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.

La principal ventaja es que los invariantes generados estáticamente son siempre ciertos en la medida en que el procedimiento de razonamiento seguido se halle libre de errores. Sin embargo, su número y alcance es reducido, debido a las limitaciones inherentes al mecanismo que analiza el código, sobre todo al enfrentarse a lenguajes poco convencionales como WS-BPEL.

3.1.3. Generación dinámica de invariantes

La generación automática de invariantes potenciales [Ern01, Ern07] 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.

Los generadores dinámicos de invariantes potenciales informan de posibles invariantes de un programa observados en la información recopilada en varias de sus ejecuciones. 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 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.

Fases

El proceso de generación dinámica de invariantes está dividido en tres fases [Ern01], como se observa en la figura 3.1. En ella, cada proceso en amarillo representa una fase: instrumentalización[1], ejecución y análisis.

[1]Este término proviene del verbo inglés *to instrument*, que significa añadir instrumentos para medir, grabar o controlar algo. El diccionario de la R.A.E. no incluye ningún término totalmente equivalente.

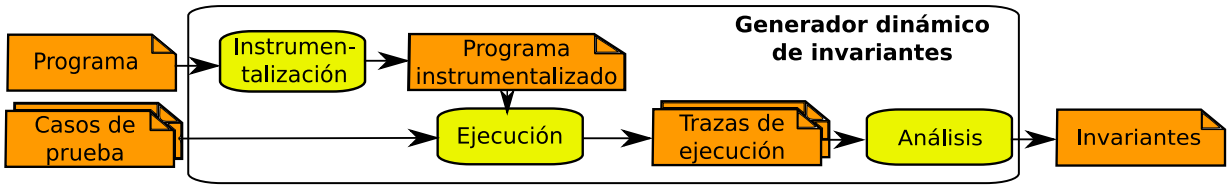


Figura 3.1: *Proceso de generación dinámica de invariantes. Adaptado de [Ern00a].*

A continuación se describen dichas fases sobre el siguiente código de ejemplo (listado 3.2), que aumenta la nota de un alumno en un punto si es menor que diez.

Listado 3.2: *Pseudocódigo de ejemplo no instrumentalizado.*

```
1 calculaNota (x: entero)
2 Si x < 10
3     y = x + 1
4 Else
5     y = 10
6 Fin Si
7 Devolver y
```

Fase de instrumentalización En esta fase se prepara el entorno para que cuando realicemos las ejecuciones del programa, se genere una traza detallada con los valores de las variables en distintos puntos del programa. Esta información se usará posteriormente para generar invariantes. Dependiendo del lenguaje puede realizarse de distintas maneras. La más usada es incluir en los puntos que interesen del código fuente instrucciones inocuas que impriman la información que se desee en un fichero.

Por ejemplo, el código anterior podría cambiar al siguiente (listado 3.3) si se instrumentaliza el punto de programa después de la instrucción condicional:

Listado 3.3: *Pseudocódigo de ejemplo instrumentalizado.*

```
1 calculaNota (x: entero)
2 Si x < 10
3     y = x + 1
4 Else
5     y = 10
6 Fin Si
7 print 'Tras␣Fin␣Si,␣x␣=' x
8 print 'Tras␣Fin␣Si,␣y␣=' y
9 Devolver y
```

Fase de ejecución En esta fase se ejecuta el programa con los casos de prueba correspondientes, generando los registros de ejecución que después se procesarán. En el

Por lo tanto, hemos decidido usar en este texto el verbo *instrumentalizar*, que hemos considerado el más similar, pues significa «Utilizar algo o a alguien como instrumento para conseguir un fin».

ejemplo anterior, si se ejecuta el programa tres veces con valores $x = 2$, $x = 1$ y $x = 9$, el registro de ejecución sería el que sigue (listado 3.4):

Listado 3.4: *Ejemplo de registro de ejecución del pseudocódigo.*

```
1  Registro1
2  Tras Fin Si, x = 2
3  Tras Fin Si, y = 3
4
5  Registro2
6  Tras Fin Si, x = 1
7  Tras Fin Si, y = 2
8
9  Registro3
10 Tras Fin Si, x = 9
11 Tras Fin Si, y = 10
```

Fase de análisis En esta última fase se adaptan los registros de ejecución al formato que espera el analizador en su entrada y se infieren los invariantes. En el ejemplo, se podrían derivar los siguientes invariantes (listado 3.5):

Listado 3.5: *Invariantes del pseudocódigo de ejemplo.*

```
1 Tras Fin Si
2 y > x
3 10 >= y
```

Influencia de los casos de prueba

Un generador dinámico de invariantes puede producir invariantes erróneos. Sin embargo, dichos invariantes no implican necesariamente fallos en el programa, sino que pueden venir originados por el empleo de un conjunto incompleto de casos de prueba.

Para ilustrarlo puede observarse como, en el ejemplo anterior, no se reciben valores de x superiores a 10. Y eso provoca el invariante falso de la línea 2 ($y > x$). Dicho invariante indica que es necesaria una inspección y mejora del conjunto de casos de prueba, incluyendo casos en los que x reciba un valor mayor o igual que 10. Si se añade un ejemplo con $x = 12$, dicho invariante desaparecerá de la salida (como se observa en el listado 3.6), lo que se suele denominar falsificación del invariante potencial.

Listado 3.6: *Invariante del pseudocódigo de ejemplo con casos de prueba adicionales.*

```
1 Tras Fin Si
2 10 >= y
```

Por lo tanto, la generación dinámica de invariantes añade a las cuatro utilidades vistas en 3.1.1 una aplicación adicional: detectar aspectos a mejorar en un conjunto de casos de prueba. En la figura 3.2 en la página siguiente se muestra el ciclo de uso de un generador dinámico de invariantes potenciales. Se observa un bucle de mejora del programa y/o del conjunto de casos de prueba las veces que sea necesario.

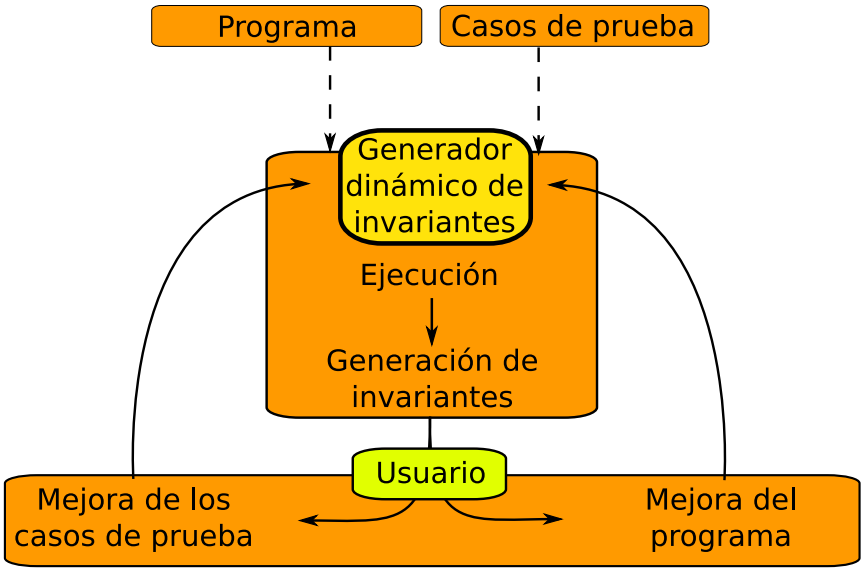


Figura 3.2: *Ciclo de uso de un generador dinámico de invariantes.*

También se puede contrastar la calidad de un conjunto de invariantes que se generen dinámicamente con los indicados en la especificación del programa. El análisis de las disonancias que se produzcan indicará si hay un error en el programa o, en caso contrario, una debilidad del conjunto de casos de prueba.

En el ejemplo anterior (listado 3.6), se observa que el invariante de la línea 2 indica que *y* no tiene valor tope inferior. Por lo tanto, está informando de un error en el tratamiento de la entrada. Si se corrige el código como se observa en el listado 3.7 (estableciendo una nota mínima de 0), se podría obtener el invariante del listado 3.8.

Listado 3.7: *Pseudocódigo de ejemplo mejorado.*

```
1  calculaNota (x: entero)
2  Si x < 10
3      y = x + 1
4      Si y < 0
5          y = 0
6      Fin Si
7  Else
8      y = 10
9  Fin Si
10 Devolver y
```

Listado 3.8: *Invariante a la salida del pseudocódigo de ejemplo mejorado.*

```
1  Tras Fin Si
2  10 >= y >= 0
```

Hay que tener en cuenta que, debido a la naturaleza dinámica del proceso, cuantos más registros se proporcionen al generador mejores resultados producirá por lo general. Es posible que en las primeras ejecuciones se obtengan invariantes aparentemente

falsos, que 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.

Como se ha comentado anteriormente, los invariantes generados dinámicamente a partir de un conjunto de casos de prueba adecuado pueden ser 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.

Nota: a partir de este punto el texto se centrará en el trabajo con invariantes potenciales generados dinámicamente. Así que para abreviar se usará el término *invariante* a secas para referirnos a ellos salvo que se indique lo contrario.

CAPÍTULO 4

Takuan: Generador dinámico de invariantes en WS-BPEL

En este capítulo se justifica, en primer lugar, la adecuación de la generación de invariantes potenciales para WS-BPEL. Después se presenta la arquitectura de Takuan, el generador de invariantes para WS-BPEL creado en este trabajo. Para ello se usan dos composiciones, una correcta (que muestra limitaciones en los casos de prueba usados) y otra con un error que Takuan descubre. A continuación se analiza su rendimiento, comentando las correspondencias de variables XML en árbol a vectores unidimensionales necesarias para su análisis.

4.1. Justificación y consideraciones

Tras analizar el estado del arte en el capítulo 2, se observa la escasa aplicación de técnicas de prueba de caja blanca directamente sobre código de composiciones WS-BPEL ejecutado en un entorno real. Las principales propuestas [Buc07, Pal08a] crean un modelo de simulación en un entorno especializado para pruebas, normalmente traduciendo el código de la composición a otro lenguaje.

Al realizar composiciones WS-BPEL el desarrollador no suele disponer del código fuente de los servicios externos que invoca. Es más, puede que dichos servicios sean ofrecidos en diversos momentos del tiempo por diferentes proveedores usando distintos algoritmos o información. Esto limita los resultados de las técnicas estáticas de prueba, y hace que el uso de alternativas dinámicas (basadas en la ejecución de código) sean una aproximación más adecuada.

Además, la simulación de un motor WS-BPEL y su entorno de ejecución (los servicios a los que llama, etc.) 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, se considera 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).

Frente a estas aproximaciones, la generación dinámica de invariantes presenta las siguientes ventajas:

Uso de código WS-BPEL No se realiza ninguna traducción del código WS-BPEL. De este modo, se evitan errores que podrían producirse en ella.

Ejecución en un motor real El proceso de generación dinámica de invariantes se basa en información recogida directamente de los registros de ejecución, sin usar ningún tipo de lenguaje intermedio.

Ejecución en un entorno real El procedimiento se basa en ejecuciones sobre un motor WS-BPEL que invoca a SW. Así se evitan imprecisiones derivadas de su modelado.

Conviene comentar en este punto que es deseable que un generador dinámico de invariantes potenciales para WS-BPEL permitiera que la composición invocara a servicios reales u, opcionalmente, llamara a servicios simulados que el mismo sistema controle (en inglés, este tipo de servicios se suelen denominar *mockups*). Esta funcionalidad es necesaria porque no puede suponerse que todos los servicios externos vayan a estar disponibles a la hora de realizar las ejecuciones necesarias para generar los invariantes. Esto puede deberse a varios motivos: limitaciones en el uso de determinados servicios, restricciones de acceso a recursos que estos necesiten, costes asociados, etc.

Además, esta opción permite una nueva posibilidad. Puede ser que 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. De este modo se podría probar la composición en escenarios alternativos del tipo *¿qué pasaría si el servicio x respondiera el valor y?*

Para ello será necesario ampliar la especificación de los casos de prueba con las respuestas que desee que proporcionen los servicios o indicando que den un fallo concreto si así se desea. Hay que tener en cuenta, eso sí, que en caso de hacer uso de dicha funcionalidad, el usuario debe ser consciente de las implicaciones que tenga las sustituciones en esos casos concretos.

Analizando esta ampliación de la especificación de los casos de prueba se observa que realmente las respuestas que proporcionan los servicios invocados no dejan de ser, en cierto modo, entradas que recibe el programa. Por lo tanto, como se observará en los experimentos que se detallan en el capítulo 6, será interesante que los invariantes se generen a partir de conjuntos de casos de prueba adecuados que incluyan no solo distintas entradas en los mensajes de creación de instancias de la composición, sino diversas respuestas de los servicios web que dicha composición llame.

Figura 4.1: *Logotipo de Takuan.*

4.2. Arquitectura de Takuan

A continuación se presenta la arquitectura de Takuan, el generador dinámico de invariantes implementado en este trabajo [Pal09b, Pal10]. El nombre de Takuan viene porque internamente se usa el generador dinámico de invariantes Daikon. En japonés, un daikon es una hortaliza asiática cuya raíz es comestible (en español se llama *rábano blanco* o *rábano japonés*). Cuando un daikon se encurte se denomina takuan, de ahí el nombre del sistema. Pero además Takuan Soho es el nombre de un legendario monje budista Zen, por eso en el logotipo (figura 4.1) de Takuan aparece un rábano vestido con un traje tradicional japonés.

Takuan implementa la arquitectura propuesta en [Pal08b], posteriormente refinada en [Gar08a, Gar08b] y presentada en [Pal08d]. En ella se integran tres sistemas libres muy maduros dentro de un flujo de ejecución que se puede observar en la figura 4.2 en la página siguiente:

ActiveBPEL es un motor de ejecución compatible WS-BPEL 2.0 [Act08]. Comparado con otros motores es bastante ligero, lo que reduce el tiempo necesario para ejecutar pruebas. Además permite incluir extensiones XPath definidas por el usuario, lo que será de utilidad para generar los registros de ejecución. ActiveBPEL está mantenido por la empresa Active Endpoints, que ofrece productos comerciales basados en él [Act09].

BPELUnit es una biblioteca de prueba unitaria para WS-BPEL [May06b] que puede usarse con varios motores WS-BPEL 2.0. Recibe como entrada ficheros XML que describen los casos de prueba a ejecutar. Además incluye un mecanismo para sustituir los servicios reales por otros que los simulen, siendo adecuado para las necesidades anteriormente descritas.

Daikon es un generador dinámico de invariantes [Ern07] que se ha usado para mejorar la calidad de programas escritos en C/C++, Java y Perl. Es muy configurable y maduro, y puede proporcionar su salida en varios formatos. Entre ellos destaca el del demostrador automático de teoremas *Simplify* [Det05], que permite eliminar invariantes redundantes de la salida de Daikon.

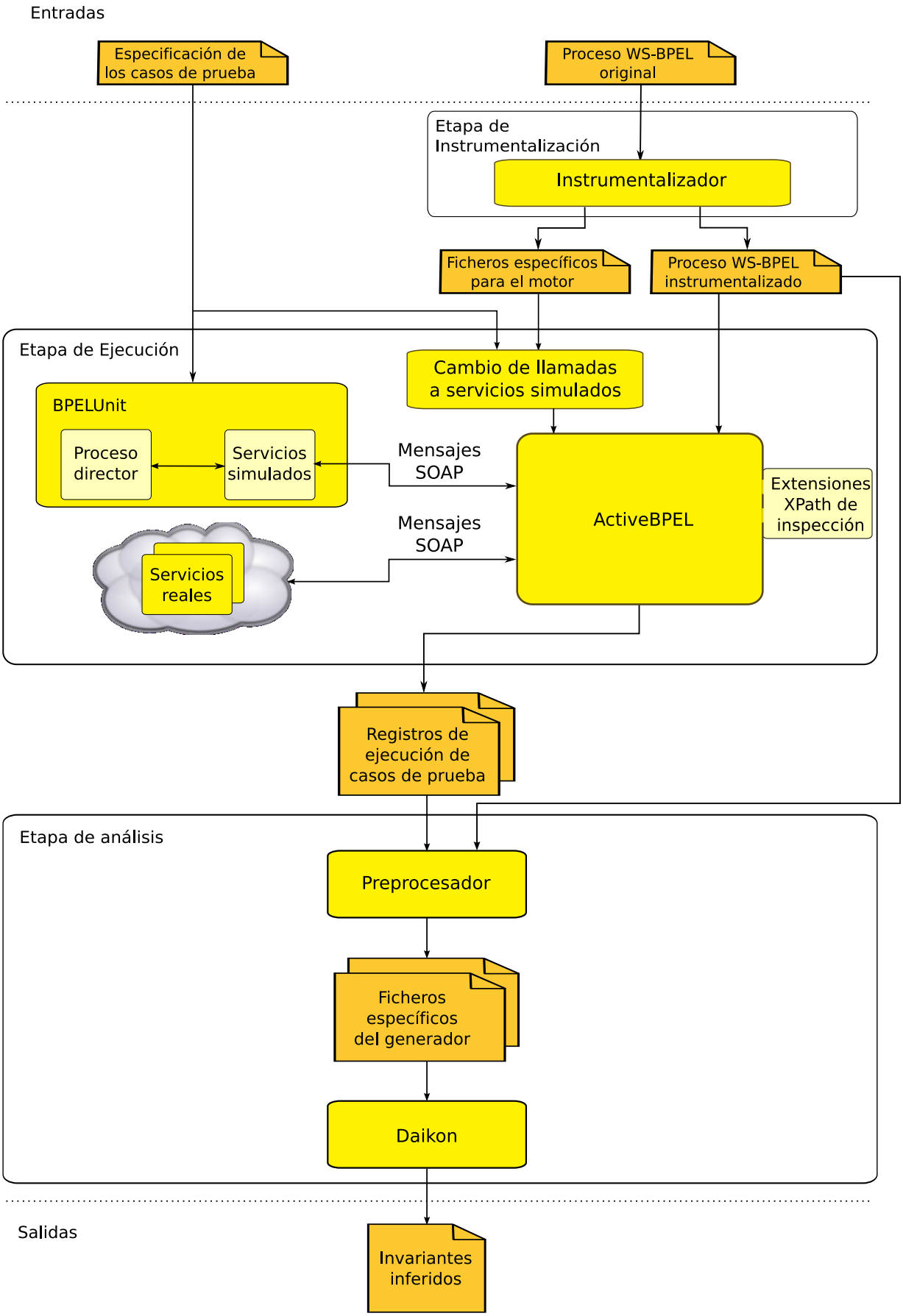


Figura 4.2: *Arquitectura de Takuan simplificada.*

El proceso de generación de invariantes en Takuan se divide en tres etapas, correspondientes a las tres fases comentadas en el apartado 3.1.3: instrumentalización, ejecución y análisis. A continuación se comenta la adaptación de cada una de ellas a WS-BPEL. Para ilustrar cada etapa se muestra su funcionamiento con la composición del préstamo bancario versión sequence descrita en el apéndice A.1.1. A lo largo de la explicación se incluyen fragmentos simplificados de los distintos ficheros implicados en el proceso. Si se desea consultar los ficheros completos se pueden encontrar en [PD11]. Igualmente, se puede consultar [Á10] si se desea información más detallada sobre el funcionamiento interno de Takuan.

La composición de ejemplo recibe peticiones de préstamos por parte de clientes de una entidad bancaria. Cada petición incluye una cantidad de dinero que se solicita, y la composición da como respuesta al cliente si el préstamo se ha aprobado o rechazado. Para ello, se basa en la cantidad solicitada y el riesgo que un servicio asesor externo le asigna al cliente. En el caso de que la cantidad sea inferior a 10.000 euros se consulta el riesgo y, si es bajo, el préstamo se aprueba. Pero si no se cumple alguna de estas dos condiciones, la composición invoca a un servicio externo de aprobación, cuya respuesta será la que se le dé al cliente.

4.2.1. Etapa de instrumentalización

En esta primera etapa se reciben los ficheros de definición de la composición WS-BPEL y se les añade la lógica necesaria para que durante su posterior ejecución se obtengan los registros de ejecución necesarios. También se aprovecha para crear algunos ficheros específicos del motor ActiveBPEL necesarios para ejecutar la composición posteriormente.

Daikon fue creado para funcionar sobre lenguajes estructurados tradicionales, en los que los puntos del programa a analizar eran las entradas y salidas en métodos de una clase, funciones o procedimientos. Como en WS-BPEL no existen dichos conceptos es necesario hacer una adaptación. Por defecto, Takuan considera puntos del programa las entradas y salidas de actividades <sequence> y <flow>, pues se suelen usar en WS-BPEL para estructurar el código. Además esto nos permite obtener invariantes dentro de instrucciones condicionales y bucles, así como fuera de invocaciones a servicios.

Por otro lado, el generador de invariantes Daikon necesita información sobre el flujo de control y los valores de variables (entendiendo por valores de variables tanto el contenido de sus campos como el de sus atributos). ActiveBPEL puede generar información de flujo de control durante la ejecución de una composición. Activando su nivel más detallado se obtiene información de ejecución de actividades y evaluación de condiciones, que es usada por Daikon.

Sin embargo, ActiveBPEL no incorpora mecanismos para el registro de valores de variables. Por ello, se han creado e integrado en ActiveBPEL funciones XPath que complementan la capacidad de registro del sistema. Estas funciones no modifican las variables de la composición: simplemente consultan los valores de sus campos y propiedades

en ese punto del programa y los escriben en el registro de ejecución de salida que genera ActiveBPEL.

En el listado 4.1 se muestra un fragmento simplificado del código WS-BPEL de la composición del préstamo descrita en el apéndice A.1.1. La asignación se ejecuta si el préstamo, que ya se sabe que es de una cantidad pequeña, es de poco riesgo según el servicio asesor. Esto se comprueba con la condición del <if> de las líneas 1 a 4, que ejecutará la rama <else> (líneas 6 a 15) si así es. Por ello, se asigna el valor *true* al campo `accept` de la variable *processOutput.output*, que se proporcionará como salida al cliente (indicando si el préstamo se concede o no) con la instrucción de copia de las líneas 9 a 12.

Listado 4.1: *Fragmento simplificado de código WS-BPEL no instrumentalizado.*

```
1  <if name="IfLowAmount">
2    <condition>
3      ( string(assessorOutput.output/risk) = 'high' )
4    </condition>
5     ...
6    <else>
7      <sequence name="SmallAmountLowRisk">
8        <assign name="approveLoan">
9          <copy>
10           <from>true</from>
11           <to>processOutput.output/accept</to>
12         </copy>
13       </assign>
14     </sequence>
15   </else>
16 </if>
```

Para registrar cómo cambia el elemento *accept* de la parte *output* del mensaje contenido en la variable *processOutput*, hay que consultar su valor antes y después de su uso. Dado que no es posible invocar a una función XPath directamente, se hace asignando el valor a registrar a una nueva variable que se define al efecto (y que no se vuelve a usar en la composición). Así pues, en el listado 4.2 en la página siguiente se muestra una versión simplificada del código instrumentalizado, donde la asignación se ha envuelto en un elemento <sequence> para asegurar que las tres instrucciones se ejecutan secuencialmente aunque estén dentro de un flujo concurrente. Nótese que solo se muestra el código que cambia, que es el que está incluido en la rama <else> del listado 4.1, y que en dicho listado solo se muestra la inspección de la variable *processOutput*, pero por lo general suelen inspeccionarse varias variables en los puntos de programa instrumentalizados.

Se observa igualmente como la función XPath encargada del registro (*reg:inspect*) envuelve la variable que se desea inspeccionar (*processOutput.output*, líneas 5 y 17). Dado que en WS-BPEL no es posible poner una expresión XPath directamente como una instrucción, se crean instrucciones de asignación a variables inocuas (*dummy*) añadidas al efecto, de modo que no interfieran en la composición (líneas 6 y 18). Además, para evi-

tar tener que definir dicha variable, se añade el atributo *ignoreMissingFromData="yes"* a las instrucciones de copia (líneas 4 y 16).

Listado 4.2: *Fragmento simplificado de código WS-BPEL instrumentalizado.*

```
1  <sequence name="SmallAmountLowRisk">
2   <sequence>
3     <assign>
4       <copy ignoreMissingFromData="yes">
5         <from>reg:inspect('processOutput.output')</from>
6         <to>dummy_processOutput.output</to>
7       </copy>
8     </assign>
9     <assign name="approveLoan">
10      <copy>
11        <from>true</from>
12        <to>processOutput.output/accept</to>
13      </copy>
14    </assign>
15    <assign>
16      <copy ignoreMissingFromData="yes">
17        <from>reg:inspect('processOutput.output')</from>
18        <to>dummy_processOutput.output</to>
19      </copy>
20    </assign>
21  </sequence>
22 </sequence>
```

Por último, una vez creada la versión instrumentalizada de la composición, se crean los ficheros que ActiveBPEL necesita para desplegar y ejecutar la composición. De este modo el usuario de Takuan puede ejecutar cualquier composición WS-BPEL 2.0 sin preocuparse del motor usado internamente. Estos ficheros son:

tiposReunidos.xml incluye las definiciones de todos los tipos y propiedades definidos en los ficheros WSDL y XML Schema, para facilitar su gestión posterior.

catalog.xml incorpora todas las dependencias que necesita la composición WS-BPEL instrumentalizada para ejecutarse.

proceso.pdd contiene información sobre dónde encontrar los distintos ficheros y servicios socios que necesita la composición.

Servicio.bpr es un fichero comprimido que incluye los ficheros anteriores organizados de forma que ActiveBPEL sólo tenga que descomprimirlos en su directorio de despliegues para que la composición esté disponible para recibir peticiones.

4.2.2. Etapa de ejecución

En la etapa anterior se generaron la versión instrumentalizada de los ficheros de definición del proceso WS-BPEL y los ficheros específicos del motor que se usa. En esta

etapa se ejecuta la composición con el conjunto de casos de prueba proporcionado por el usuario, y los registros de ejecución generados se pasarán a la siguiente fase.

Para ello, en primer lugar se tiene que desplegar la versión instrumentalizada de la composición. Después ejecutar cada uno de los casos de prueba (almacenando sus registros de ejecución). Y por último, replegar el proceso. BPELUnit es el responsable de todas estas tareas excepto de ejecutar la composición, que es llevada a cabo por ActiveBPEL. En concreto, el proceso director de BPELUnit se encarga de:

1. Desplegar la composición WS-BPEL en el motor. Para hacerlo en ActiveBPEL, BPELUnit envía el fichero `.bpr` creado en la etapa anterior al servicio de despliegue que tiene el servidor.

2. Si hay que simular SW, lanzar el servidor de servicios simulados, que contiene los servicios web que se sustituyen según la especificación del caso de prueba actual.

3. Para cada caso de prueba:

   a) Si procede, configurar el servidor de servicios simulados indicando las respuestas SOAP que dará a cada petición recibida de la composición. También es posible indicar que espere un determinado tiempo antes de responder, que dé un fallo concreto o que incluso no responda (simulando un servicio no disponible).

   Para hacerlo se usa el fichero de descripción de despliegue de proceso de ActiveBPEL (*ActiveBPEL Process Deployment Descriptor*, PDD), que es uno de los ficheros específicos de ActiveBPEL creados con anterioridad. Como las direcciones de servicios especificadas en el fichero PDD tienen prioridad sobre aquellas de los ficheros WSDL originales, simplemente es necesario modificarlo para controlar si se desea invocar a un servicio simulado. Puede verse un extracto del fichero completo en el listado 4.3, en el que se observa cómo se indica que se busque el servicio en una dirección local (*localhost:7777*).

Listado 4.3: *Extracto de fichero de despliegue* `.pdd`.

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <process ... >
3      <partnerLinks>
4          <partnerLink name="approver">
5              <partnerRole ... >
6                  <wsa:EndpointReference ...>
7                      <wsa:Address>http://localhost:7777/ws/approver</wsa:Address>
8                      ...
9                  </wsa:EndpointReference>
10             </partnerRole>
11         </partnerLink>
12     </partnerLinks>
13 </process>
```

b) Llamar al proceso WS-BPEL con los parámetros indicados en la especificación del caso de prueba.

c) Obtener los resultados de la invocación.

4. Replegar el proceso WS-BPEL del motor haciendo uso del servicio de repliegue que tiene ActiveBPEL.

En el listado 4.4 se puede ver un fragmento simplificado de la especificación XML de los casos de prueba para el ejemplo del préstamo. Se observa que tanto la petición del cliente como las respuestas de los servicios externos se definen escribiendo el cuerpo del mensaje SOAP que debe generarse. En este ejemplo, el caso de prueba comienza solicitando 1.500 euros (como se observa en la línea 12) dentro del *clientTrack* (que especifica el comportamiento del SW socio). Así mismo, y el aprobador está sustituido por un servicio simulado cuyo comportamiento se especifica con la marca *tes:partnerTrack*, respondiendo *true* cuando sea llamado (según se indica en la línea 29).

Listado 4.4: *Fragmento simplificado de especificación de caso de prueba BPELUnit.*

```
1  <testSuite>
2    <testCases>
3     <!-- Especific. del primer caso de prueba -->
4     <testCase name="lowAmount" ...>
5
6       <!-- Comportamiento del cliente -->
7       <clientTrack>
8         <sendReceive operation="approveLoan" ...>
9           <send fault="false">
10            <data>
11              <ex:ApprovalRequest>
12                <ex:amount>1500</ex:amount>
13              </ex:ApprovalRequest>
14            </data>
15          </send>
16          <receive fault="false"/>
17        </sendReceive>
18      </clientTrack>
19
20      <!-- Comportamiento de un socio -->
21      <tes:partnerTrack name="approver">
22        <tes:receiveSend
23            service="ap:LoanApprovalService"
24            port="LoanApprovalPort"
25            operation="approveCredit">
26          <tes:send fault="false">
27            <tes:data>
28              <esq:ApprovalResponse>
29                <esq:accept>true</esq:accept>
30              </esq:ApprovalResponse>
31            </tes:data>
```

```
32              </tes:send>
33              <tes:receive fault="false"/>
34            </tes:receiveSend>
35          </tes:partnerTrack>
36
37          <!-- Comportamiento de otros socios, si los hubiera -->
38        </testCase>
39
40        <!-- Especific. del siguiente caso de prueba -->
41      </testCases>
42  </testSuite>
```

Como se comentó anteriormente, ActiveBPEL es el encargado de ejecutar el proceso WS-BPEL. Durante dicha ejecución se genera información sobre el flujo de control y valores de variables que se almacena en los registros. En el listado 4.5 se puede ver un extracto del registro de ejecución del caso de prueba anterior en el que se ha sustituido cierta información irrelevante por puntos suspensivos.

Listado 4.5: *Fragmento simplificado del registro de ejecución de ActiveBPEL extendido.*

```
1  Executing [(...)/ sequence/assign]
2   INSPECTION(processOutput.output/accept) = false
3  Completed normally [(...)/sequence/assign]
4  Executing [(...)/ sequence/assign]
5  Completed normally [(...)/sequence/assign]
6  Executing [(...)/ sequence/assign]
7   INSPECTION(processOutput.output/accept) = true
8  Completed normally [(...)/sequence/assign]
```

Se observa en las líneas 1 a 3 que se registra el valor original del campo *accept* de la variable *processOutput.output*. Después, la segunda asignación (la del código WS-BPEL original) se ejecuta con éxito (líneas 4 y 5), poniendo *processOutput.output/accept* a *true*, lo que indica que el préstamo se concede. Y, finalmente, la última inspección registra el cambio en el valor de la variable.

4.2.3. Etapa de análisis

En la etapa previa, cada caso de prueba de la especificación inicial generó su registro de ejecución. Ahora es necesario pasarlos al generador de invariantes Daikon para que haga el resto del trabajo.

Para ello hay que adaptar dichos registros al formato de entrada de Daikon, y además es necesario generar un segundo fichero de declaraciones con información adicional al registro de cada ejecución. A continuación se describen cada uno de los ficheros generados:

- Un fichero de declaraciones (*declaration file*) que indica los puntos del programa instrumentalizados y las variables inspeccionadas en cada uno de ellos. Se puede

observar un ejemplo simplificado en el listado 4.6 (se ha eliminado la información de comparabilidad que se comentará en la sección 5.1).

Listado 4.6: *Extracto simplificado de fichero de declaraciones* `.decl`.

```
1  DECLARE
2  LoanApproval.SmallAmountLowRisk:::ENTER
3  processOutput.output.accept[]
4  xsd:boolean[]
5  boolean[]
6  1
7  LoanApproval.SmallAmountLowRisk:::EXIT
8  processOutput.output.accept[]
9  xsd:boolean[]
10 boolean[]
11 1
```

En el listado se observa que el bloque de declaraciones de variables comienza con la palabra *DECLARE*. A continuación, en la línea 2, aparece el punto del programa cuyas variables se describirán. En este caso es la entrada en la rama condicional que se instrumentalizó en el listado 4.2 y cuyo registro se mostró en el listado 4.5. Después aparece un bloque de cuatro líneas por cada variable que se registra en dicho punto del programa. La primera de las líneas es el identificador de la variable (el cambio de la barra invertida por el punto es por la aplicación de correspondencia XML Schema que se explicará en la sección 4.3). La segunda es el tipo de la variable en el sistema de tipos del programa original (en este caso, el XML Schema usado en la composición). A continuación aparece el tipo Java que Daikon usará para detectar invariantes referentes a ella. Por último, aparece un tipo abstracto (cuyo significado se explicará en la sección 5.1). Por ejemplo, para la variable *processOutput.output.accept[]* de la línea 3, su tipo en XML Schema es *xsd:boolean[]*, tiene tipo Java *boolean[]* y su tipo abstracto es *1*. A continuación se define la misma información para el punto de programa de la salida de la rama condicional.

- Una serie de ficheros de registro (*data trace files*). En concreto, uno por cada caso de prueba. Cada uno contiene los valores de las variables registradas en cada punto de programa ejecutado. En el listado 4.7 se incluye un ejemplo.

Listado 4.7: *Extracto simplificado de fichero de trazas* `.dtrace`.

```
1 LoanApproval.SmallAmountLowRisk:::ENTER
2 processOutput.output.accept
3  false
4 0
5 LoanApproval.SmallAmountLowRisk:::EXIT
6 processOutput.output.accept
7 true
8 0
```

Este trozo del fichero comienza indicando el punto de programa que se ha instrumentalizado. Después aparecen bloques de tres líneas por cada variable inspeccionada. La primera indica el nombre de la variable. La segunda es su valor (si no está inicializada aparecerá el término *nonsensical*) y la tercera es un número entero que indica si la variable ha cambiado de valor desde la última ejecución de ese punto de programa (0 si no se ha modificado, 1 si lo ha hecho y 2 para variables no inicializadas). En el ejemplo, se observa la modificación del valor de *processOutput.output.accept*, que al entrar en la rama condicional vale *false*, pero al salir ha cambiado a *true*.

El fichero de declaraciones se obtiene automáticamente al procesar la versión instrumentalizada de la composición WS-BPEL. A partir de ella se consiguen dos datos de cada variable:

- Su tipo, tanto en el sistema de tipos original como en el subconjunto del sistema de tipos de Java que Daikon acepta. Esta traducción permite que Daikon maneje sistemas distintos a los de Java (como XML Schema en nuestro caso). El subconjunto incluye la mayoría de los escalares nativos de Java y vectores unidimensionales de ellos.

- Un índice de comparabilidad (*comparability index*). De este modo se puede indicar a Daikon que dos variables del mismo tipo de datos pertenecen también a un mismo tipo abstracto (y por lo tanto pueden aparecer juntas en un invariante). Por ejemplo, una cantidad monetaria, la edad de una persona y los años para devolver un préstamo pueden ser todos de tipo entero. Pero solo la edad y los años pertenecen al mismo tipo (años naturales), por lo que no parece de interés generar invariantes que los relacione con la cantidad. Esta información se usará posteriormente en el apartado 5.1.

Una vez que se tiene el fichero de declaraciones, se hace una ligera adaptación de los registros de ejecución que ActiveBPEL produjo anteriormente al formato aceptado por Daikon. Tras pasar estos ficheros a Daikon, se obtiene finalmente el listado de invariantes deducidos, que es la salida de Takuan.

En el listado 4.8 en la página siguiente se incluyen algunos de los invariantes que se podrían obtener en el ejemplo desarrollado (nótese que los invariantes concretos que se obtengan dependerán del resto de casos de prueba proporcionados). En Daikon, el punto de programa antes de la instrucción i, se denota como i:::$ENTRY$, mientras que i:::$EXIT$ es el punto después de la instrucción i. En el caso de WS-BPEL los <sequence> y <flow> son instrucciones de bloque, por lo que pueden existir otros puntos de programa anidados entre sus ENTRY y EXIT.

El fragmento comienza indicando el punto de programa en que se cumplen los invariantes. Las líneas segunda y tercera confirman la condición de entrada en la rama condicional: el riesgo es bajo y la cantidad por debajo del umbral. En las líneas sexta y séptima se comprueba que dichas condiciones se mantienen a la salida de la rama. Por

último, las líneas cuarta y octava muestran que al entrar en la rama el préstamo no está concedido (pues es el valor inicial de *processOutput.output.accept*), pero sí al salir de ella.

Listado 4.8: *Selección simplificada de invariantes cuando hay riesgo bajo.*

```
1 LoanApproval.SmallAmountLowRisk:::ENTER
2 assessorOutput.output.risk == "low"
3 processInput.input.amount <= 10000
4 processOutput.output.accept == 0
5 LoanApproval.SmallAmountLowRisk:::EXIT
6 assessorOutput.output.risk == "low"
7 processInput.input.amount <= 10000
8 processOutput.output.accept == 1
```

4.2.4. Análisis de resultados

En el listado 4.9 se muestran algunos de los invariantes obtenidos a la salida de la composición de ejemplo. Intencionadamente, el conjunto de casos solicitaba siempre una cantidad de 150.000 euros (por encima del umbral de 10.000 euros).

Listado 4.9: *Selección simplificada de invariantes con cantidad alta.*

```
1 LoanApproval.LargeAmount:::EXIT
2 approverInput.input.amount == processInput.input.amount
3 approverOutput.output.accept == processOutput.output.accept
4 approverInput.input.amount == 150000
5 approverOutput.output.accept one of { 0, 1 }
6 size(approverOutput.output.accept) == 1
```

En la primera línea del listado 4.9, Daikon indica que los invariantes que siguen se cumplen al terminar la ejecución de la instrucción etiquetada como *LargeAmount*. Esta instrucción es una de las ramas principales de la composición, que se ejecuta cuando la petición de préstamo es de una cantidad considerada alta. A su terminación está en la variable *processOutput* la respuesta del servicio aprobador, por lo que a la composición solo le queda mandarle la información al cliente y terminar.

En la línea 2 Daikon ha sido capaz de deducir que la cantidad originalmente solicitada es la que se usa para consultar al servicio aprobador. Puede parecer que no es una propiedad demasiado interesante, pero nos asegura que no se producen modificaciones en un aspecto muy importante de la composición. De manera similar, confirma que su respuesta de aprobación se usa como salida final del proceso (línea 3). Igualmente detecta (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 la respuesta siempre es verdadero o falso, no existen otras respuestas que puedan confundir al usuario).

Sin embargo, existe un invariante falso: la línea 4 indica que la cantidad solicitada es siempre 150.000 euros, lo que se sabe que es falso. La razón por la que lo deduce es porque todos los casos de prueba pedían exactamente 150.000, lo que hace pensar a Daikon que se trata de una propiedad. Para falsificarlo bastaría con añadir casos de

prueba con cantidades superiores a 10.000 que no sean 150.000 (para que la variable tome otros valores en dicho punto del programa).

Es más, ya se sabe que el valor que almacena la variable *accept* solo puede ser 0 o 1, porque se define como booleana en la definición de tipos XML Schema. Por lo tanto, incluir tal invariante en la salida solo la complica innecesariamente. En el capítulo 5 se explica cómo Takuan implementa esta y otras optimizaciones.

Por último, la salida también se podría mejorar eliminando un invariante redundante: si se sabe que *accept* es una variable con un único valor 0 o 1, no es necesario que indique en la línea 6 que su longitud es siempre 1. Este invariante se podría eliminar pasando la salida por el demostrador de teoremas Simplify, haciéndola así más fácil de leer y comprender.

Esta sencilla ejecución ha mostrado como Takuan permite perfilar el funcionamiento de una composición, generando invariantes que indican su comportamiento y que, dado el caso, también pueden revelar limitaciones del conjunto de casos de prueba usado para inferirlos.

A continuación se aplica Takuan sobre la misma composición, a la que se ha incluido un fallo, y se muestra cómo los invariantes que genera ayudan a descubrirlo. El fallo se encuentra en la condición que decide, para préstamos de cantidades pequeñas, si es necesario llamar al servicio aprobador o se puede aceptar la petición directamente. En concreto, se cambia el operador relacional de dicha condición de $=$ a $!=$, pasando la condición de *si el riesgo es alto* a *si el riesgo no es alto*. El código resultante está en el listado 4.10.

Listado 4.10: *Fragmento simplificado de código WS-BPEL con error.*

```
1 <if name="IfLowAmount">
2   <condition>
3     ( string(assessorOutput.output/risk) != 'high' )
4   </condition>
5    ...
6   <else>
7    ...
8 </if>
```

En el listado 4.11 se incluyen invariantes de dos puntos de programa distintos (las entradas de las dos ramas de la condición que discrimina según el riesgo) que muestran el error. El primero indica que siempre que se entra en la secuencia de instrucciones que deben ejecutarse con cantidades pequeñas y riesgo alto (es decir, el punto de programa *LoanApproval.SmallAmountHighRisk*), dicho riesgo es bajo. Mientras que el otro invariante informa de lo contrario: las instrucciones que tratan los préstamos de bajo riesgo (en el punto de programa *LoanApproval.SmallAmountLowRisk*) se están ejecutando únicamente con casos de riesgo alto.

Listado 4.11: *Selección simplificada de invariantes que muestran errores.*

```
1 LoanApproval.SmallAmountHighRisk:::ENTER
2 assessorOutput.output.risk == "low"
```

```
3
4 LoanApproval.SmallAmountLowRisk:::ENTER
5 assessorOutput.output.risk == "high"
```

Este pequeño ejemplo muestra cómo un análisis de los invariantes generados por Takuan permite detectar las instrucciones de la composición afectadas por un error e informa sobre las variables afectadas, facilitando su corrección.

4.2.5. Rendimiento del sistema

Llegado este punto se ha mostrado cómo Takuan puede generar invariantes con información de interés sobre composiciones WS-BPEL. En este apartado se analizará el tiempo que tarda Takuan en ejecutarse según el tamaño del conjunto de casos de prueba recibido como entrada.

Para hacer el estudio se han realizado distintas ejecuciones de Takuan sobre una máquina con un procesador dual-core Intel Core Duo T2250, con 1GiB de RAM DDR2 a 533MHz y un disco duro de 80GB a 5400rpm HDD. El sistema operativo es una instalación de GNU/Linux Ubuntu 8.04 estándar, con el núcleo `2.6.24-18-generic` que incluye por defecto. Durante las pruebas los procesos activos fueron principalmente aquellos creados por Takuan: el Sun 6.0 JRE, Apache Ant 1.7.0, Perl 5.8.8, Daikon 4.3.4, ActiveBPEL 4.1, BPELUnit 1.0 y GNU `time` 1.7. Durante su ejecución no hubo otros procesos que consumieran tiempo de CPU, memoria o canal de disco de manera significativa.

En cada caso se midió el tiempo natural (del inglés *wall time*) medio requerido para ejecutar 10 iteraciones, pues se considera que es un buen indicador del total de recursos usados (tiempo de CPU, latencia de disco, etc.). En la figura 4.3 en la página siguiente se muestra el tiempo natural que llevó la instrumentalización, ejecución y análisis de cuatro conjuntos de casos de prueba de 5, 10, 20 y 50 casos sobre la composición del préstamo comentada en el apéndice A.1.1. Esta comprende 10 puntos de programa con 12 variables de media en cada uno.

Si se observa el tiempo necesario para instrumentalizar cada conjunto, se puede afirmar que el tiempo requerido es constante independientemente del número de casos. Esto es debido a que es un proceso que se realiza sobre los ficheros de la especificación de la composición, consistente en el procesado de código XML y el añadido de instrucciones en los puntos necesarios. Por lo tanto no se tiene que temer sobre su escalabilidad para composiciones o conjuntos de casos de prueba grandes.

En lo referente a la ejecución, hay que tener en cuenta la biblioteca de pruebas que se usa, BPELUnit, y el motor de ejecución ActiveBPEL. BPELUnit no impone ninguna restricción sobre el tamaño del proceso. Básicamente lanza el servidor responsable de los servicios simulados necesarios para cada caso de prueba e invoca al proceso WS-BPEL. También se considera que ActiveBPEL no debería dar problemas en cuanto a rendimiento: es un motor estable, maduro y con una cierta presencia en la industria (respaldado por la empresa Active Endpoints). Como se observa en la figura 4.3, el tiempo necesitado para ejecutar los casos de prueba se incrementa linealmente. Es evidente que a más

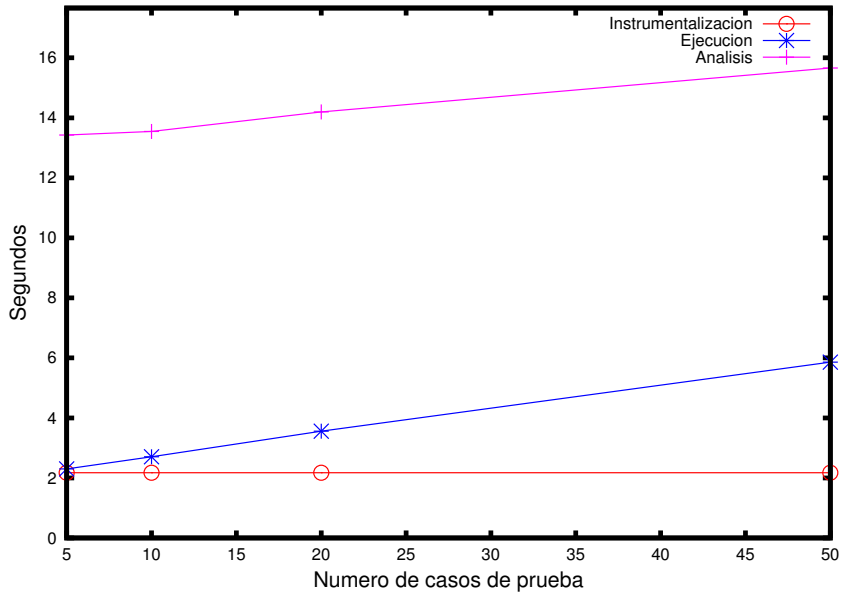


Figura 4.3: *Tiempo natural de cada fase de Takuan.*

casos (o mayor complejidad en la composición), mayor tiempo de ejecución, pero hay poco que hacer para mejorar esos tiempos.

Así que el análisis se reduce a Daikon. Daikon por sí mismo no tiene restricciones sobre los tamaños de las trazas a analizar (lo que deja como único límite la memoria del sistema), pero el tiempo que necesita para generarlos sí que puede ser un problema si la entrada es muy grande. En [Ern01] sus autores lo descomponen en la siguiente fórmula: $Time = O((vars^3 * falsetime + trueinvs * testsuite) * program)$, donde *vars* es la cantidad de variables en cada punto del programa, $falsetime$ es el tiempo medio necesario para falsificar un invariante, *trueinvs* es el mínimo número de invariantes que son ciertos en un punto del programa, *testsuite* es el número de casos de prueba en la entrada, y *program* la cantidad de puntos del programa instrumentalizados. Como se observa el número de variables es el elemento más crítico, seguido de la cantidad de puntos del programa instrumentalizados. En el capítulo 5 se explican las técnicas implementadas para reducirlos.

4.3. Correspondencia con XML Schema

En esta sección se presentan las técnicas de correspondencia implementadas para el tratamiento de variables con estructura de árbol XML Schema en Daikon. Posteriormente se evalúa su rendimiento para ver qué aspectos se pueden mejorar.

4.3.1. Justificación

En la sección 4.2 se ha explicado el proceso de generación de invariantes para WS-BPEL ilustrando su evolución con una composición sencilla. Dicha composición incluye solo dos instrucciones condicionales y variables escalares, que Daikon puede manejar directamente.

Sin embargo, las composiciones WS-BPEL pueden ser mucho más complejas, tanto en lógica interna como en la estructura de las variables que se usa. El aumento de la lógica interna no es problema para Daikon (ni para Takuan por lo tanto). Pero sí lo es el uso de variables XML Schema con estructura más compleja, pues Daikon no es capaz de procesar tipos de datos de más de una dimensión (matrices, árboles de más de un nivel, etc.).

XML Schema organiza las variables de más de una dimensión en árboles. Por lo tanto, Takuan necesita dividir cada variable tipo árbol de más de un nivel en varias variables tipo vector unidimensional, y modificar igualmente sus referencias en las trazas de ejecución y demás ficheros internos para que Daikon pueda procesarlas. En esta división se puede respetar parte de la estructura del árbol, pero no hay más remedio que perder otra parte.

Por lo tanto, se han implementado dos técnicas para solucionar este problema: aplanado y división (en inglés *matrix slicing* y *matrix flattening*, respectivamente) [Pal08c]. La base de ambas técnicas es similar: convertir la estructura arbórea en otra menos compleja (lineal) respetando parte de la estructura original. Esto hace que sean dos métodos complementarios, cada uno con sus puntos fuertes y débiles. Dependiendo de la composición y los invariantes que se deseen obtener sobre ella será preferible usar uno u otro.

Para ilustrar estos métodos se usará, en primer lugar, una variable que almacena información sobre ventas de artículos. Dicha variable no pertenece a ninguna composición concreta, sino que está creada al efecto, para mostrar toda la complejidad que debe afrontar el mapeador (programa que implementa la correspondencia) en un caso general. Para ello, su estructura es un árbol de tres niveles, en el que el primero de ellos almacena varias ventas, cada una de ellas se compone de varios pedidos en un segundo nivel, que a su vez contienen varios artículos con sus precios en el tercero y último. La estructura de esta variable se puede ver visualmente en la figura 4.4.

Además, se muestra la aplicación de los métodos descritos a la composición del motor de *metabúsqueda* en Internet detallada en el apéndice A.2.1. Esta composición implementa un sistema de consulta de información en Internet usando dos servicios simples de búsqueda (uno de Google y otro de MSN). Para ello los invoca a ambos y ofrece al usuario sus resultados intercalados empezando por el primer resultado de Google si lo hubiera.

Esta composición es la más compleja tratada con Takuan hasta la fecha, pues hace uso de bucles, concurrencia y variables no escalares. Aunque estas variables no llegan a la complejidad de la variable anterior, pertenece a una composición WS-BPEL, lo que permite acompañar la explicación con los resultados de Takuan al ejecutarla.

En concreto, se desarrolla un ejemplo de ejecución de Takuan con una llamada que recibe dos resultados de MSN y uno de Google. Dichos resultados se almacenan en el elemento `MetaSearchProcessResponse` dentro del campo `payload` de la variable de respuesta de la composición, *outputVariable*. El listado 4.12 muestra la estructura simplificada de dicho campo. La variable contiene en primer lugar el número total de respuestas obtenidas en la búsqueda. Después indica el número de ellas que provienen de Google y de MSN. Por último, aparece por cada resultado un elemento *result* con la URL, el título, una descripción (en inglés *snippet*) y el buscador de origen de cada resultado concreto.

Listado 4.12: *Contenido simplificado de variable en la metabúsqueda.*

```
<MetaSearchProcessResponse>
  <noResult>3</noResult>
  <noFromGoogle>1</noFromGoogle>
  <noFromMSN>2</noFromMSN>

  <result>
   <url>http://softwarelibre.uca.es</url>
   <title>OSLUCA</title>
   <snippet>Oficina de Software Libre, Universidad de Cadiz</snippet>
   <from>Google</from>
  </result>

  <result>
   <url>http://cudisol.ourproject.org</url>
   <title>CUDISOL</title>
   <snippet>Cursos para la Difusion del Software Libre</snippet>
   <from>MSN</from>
  </result>

  <result>
   <url>http://jornadas.adala.org</url>
   <title>Jornadas Andaluzas de Software Libre</title>
   <snippet>Organizadas por ADALA, CAGESOL y OSLUCA, Algeciras 2004</snippet>
   <from>MSN</from>
  </result>
</MetaSearchProcessResponse>
```

El registro generado por ActiveBPEL con las funciones XPath de este árbol es el que aparece en el listado 4.13.

Listado 4.13: *Registro simplificado de ejecución de la metabúsqueda.*

```
I (.../ noResult[1]) = 3
I (.../ noFromGoogle[1]) = 1
I (.../ noFromMSN[1]) = 2
I (.../ result[1]/url[1]) = http://softwarelibre.uca.es
I (.../ result[1]/ title [1]) = OSLUCA
I (.../ result[1]/snippet[1]) = Oficina de Software Libre, Universidad de Cadiz
I (.../ result[1]/from[1]) = Google
I (.../ result[2]/url[1]) = http://cudisol.ourproject.org
I (.../ result[2]/ title [1]) = CUDISOL
```

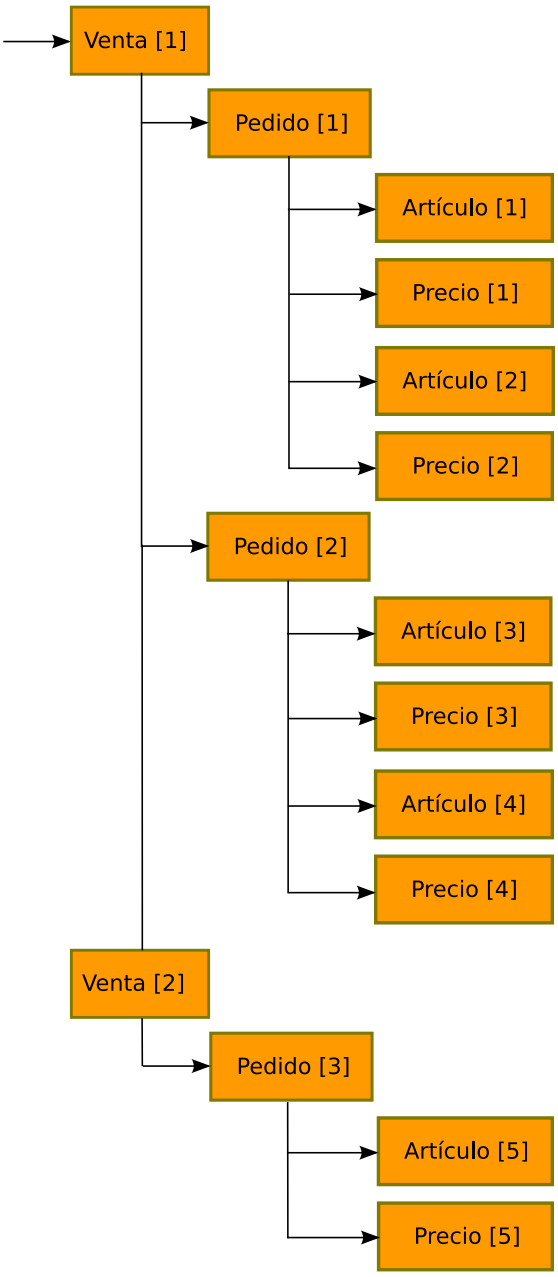


Figura 4.4: *Variable de ventas original.*

```
10 I (.../ result[2]/snippet[1]) = Cursos para la Difusion del Software Libre
11 I (.../ result[2]/from[1]) = MSN
12 I (.../ result[3]/url[1]) = http://jornadas.adala.org
13 I (.../ result[3]/ title [1]) = Jornadas Andaluzas de Software Libre
14 I (.../ result[3]/snippet[1]) = Organizadas por ADALA, CAGESOL y OSLUCA, Algeciras 2004
15 I (.../ result[3]/from[1]) = MSN
```

A primera vista, observando el registro de ejecución, podría parecer que cada elemento de *result* solo tiene una *url*, un *title*, un *snippet* y un *from*. Sin embargo, en la definición XML Schema de la variable se permite que aparezcan varios, por lo que la correspondencia debe tenerlo en cuenta. Esto es algo muy común en las definiciones de tipos para procesos WS-BPEL: no se incluyen todas las restricciones del tipo de dato porque se confía en que la composición hará un buen uso de él.

Otra posibilidad que se podría antojar dada las limitaciones del ejemplo concreto, sería hacer tres vectores de cadenas. Cada uno tendría todos los campos por cada elemento *result* (la `url`, el `title`, el `snippet` y el `from`). Sin embargo, en el caso general, *result* podría tener hijos de distinto tipo (por ejemplo, un número entero, un flotante, etc.), o cadenas con distintas restricciones incompatibles (como un hijo tipo cadena solo de vocales y otro de tipo cadena de números y consonantes), por lo que tampoco es válida dicha estrategia.

Por lo tanto, hay que considerar que los datos obtenidos se refieren a una variable (*result*), con dos dimensiones: una para cada resultado obtenido (*result[1]*, *result[2]* y *result[3]*), otra por cada hijo de estos (`url`, `title`, `snippet` y `from`).

A continuación se aplican ambas técnicas de correspondencia a los dos ejemplos comentados: la variable de ventas y la variable de respuesta de la composición de la metabúsqueda.

4.3.2. Correspondencia mediante división

La primera aproximación se basa en el funcionamiento de Kvasir, el *front-end* de Daikon para C++. En Daikon, un *front-end* es un pequeño programa que traduce las trazas de un lenguaje de programación concreto (y la información relevante sobre ellas) a su formato interno, para que pueda procesarlas directamente. La distribución estándar de Daikon incluye varios de ellos, y su manual de programador explica como desarrollar otros para soportar el lenguaje que se desee.

En concreto, Kvasir soluciona las limitaciones de Daikon a la hora de analizar matrices N-dimensionales (de dimensión mayor que uno) dividiéndolas en matrices de dimensión $(N-1)$ de manera iterativa a partir de la raíz, hasta que solo queden vectores unidimensionales. En la figura 4.5(b) se observa el resultado tras aplicarlo a la variable de ventas la figura 4.5(a). Como se puede ver, las ventas agrupadas inicialmente se han divido en dos variables que, a su vez, se han divido por pedidos. Una vez quedan únicamente vectores unidimensionales, se dividen en una variable por cada tipo de dato que contienen. Por lo tanto se tiene una primera variable con los artículos del primer pedido

de la primera venta, otra variable independiente con los precios del primer pedido de la primera venta, e igual con el resto de ventas y pedidos.

Al aplicarlo a la metabúsqueda se observa que la variable bidimensional (*result*) se ha dividido en tres, una por cada resultado (*result[1]*, *result[2]* y *result[3]*). Y cada resultado se ha dividido por cada tipo de elemento hijo, generando así 12 variables:*result[1].url[]*, *result[1].title[]*, *result[1].snippet[]* y *result[1].from[]*, *result[2].url[]*, etc.

Nótese que todos los caracteres hasta el último corchete son literales: *result[1].url[]*, *result[2].url[]* y *result[3].url[]* son tres variables (tipo vector unidimensional de cadenas URL) absolutamente independientes. Simplemente se les deja el corchete intermedio con su índice para que el usuario sepa los elementos de la variable XML que almacenan. Por ejemplo, si se obtuviera el invariante *result[1].from[1]=result[2].from[1]* se podría saber que el primer y segundo resultado provienen del mismo motor de búsqueda.

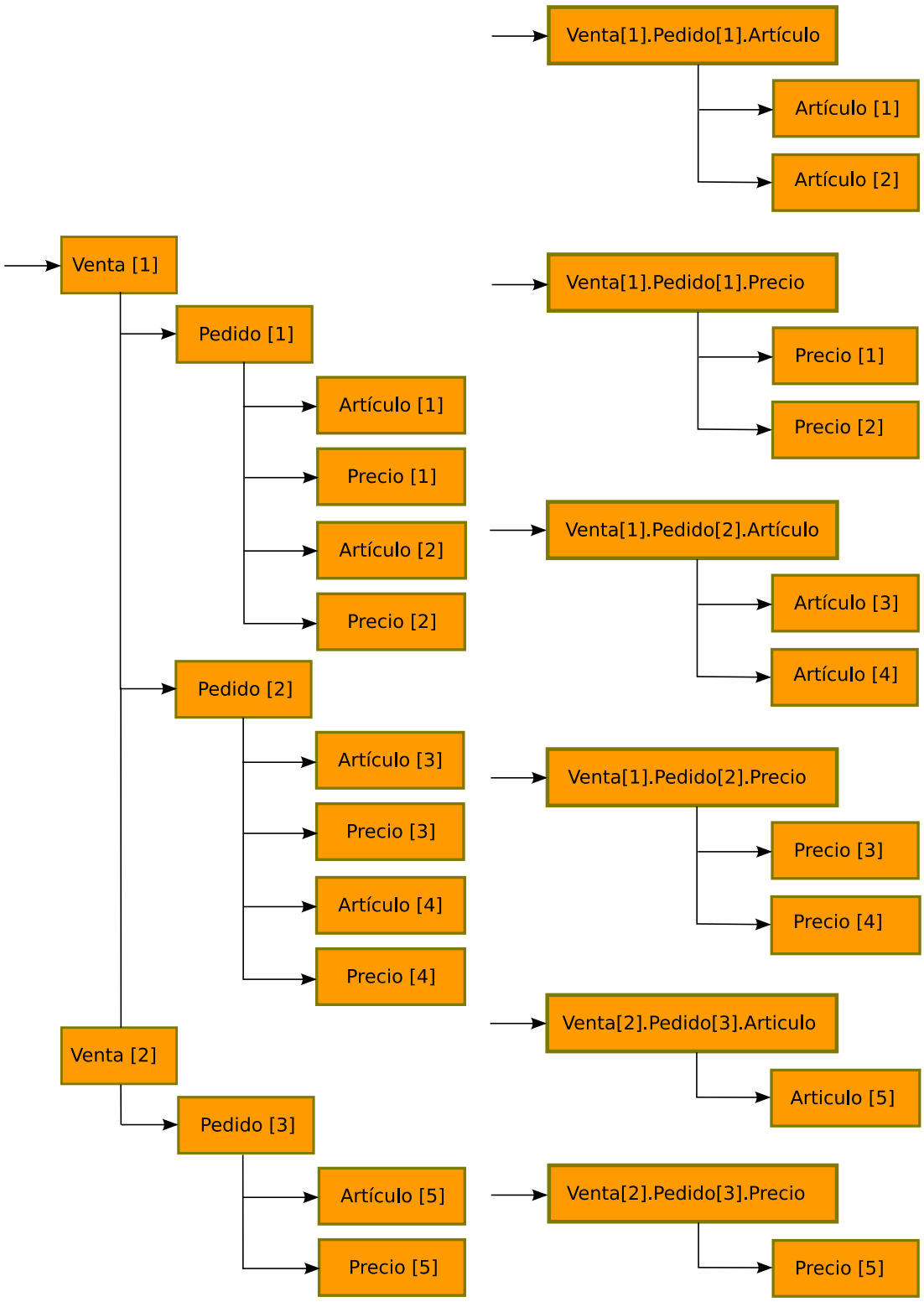
Listado 4.14: *Correspondencia por división en el registro de ejecución de la metabúsqueda.*

```
1  (...). noResult[1] = 3
2  (...). noFromGoogle[1] = 1
3  (...). noFromMSN[1] = 2
4
5  (...). result[1]/url = [ ''http://softwarelibre.uca.es'' ]
6  (...). result[1]/ title = [ ''OSLUCA'' ]
7  (...). result[1]/snippet = [ ''Oficina de Software Libre, Universidad de Cadiz'' ]
8  (...). result[1]/from = [ ''Google'' ]
9  (...). result[2]/url = [ ''http://cudisol.ourproject.org'' ]
10 (...). result[2]/ title = [ ''CUDISOL'' ]
11 (...). result[2]/snippet = [ ''Cursos para la Difusion del Software Libre'' ]
12 (...). result[2]/from = [ ''MSN'' ]
13 (...). result[3]/url = [ ''http://jornadas.adala.org'' ]
14 (...). result[3]/ title = [ ''Jornadas Andaluzas de Software Libre'' ]
15 (...). result[3]/snippet = [ ''Organizadas por ADALA, CAGESOL y OSLUCA, Algeciras 2004'' ]
16 (...). result[3]/from = [ ''MSN'' ]
```

Las ventajas y desventajas de esta técnica son:

- En principio es la manera más natural de obtener invariantes de elementos concretos de una estructura multidimensional, porque cada elemento se almacena en una variable independiente. Por ejemplo, en la metabúsqueda permite comprobar que siempre que Google de un resultado, este sea el primer resultado de la composición. Para ello se consultarían los invariantes relativos a *result[1].from[]* en la variable *outputVariable* en los puntos del programa que solo se ejecuten al obtener al menos un resultado de Google.

- Cada variable resultante corresponde a un conjunto de elementos contiguos en el árbol XML original. De este modo, en sucesivos pasos (como se verá en la sección 5.2) se podrán eliminar invariantes redundantes referentes a la longitud del vector usando las restricciones que puede imponer el XML Schema sobre tamaños mínimos y máximos de hijos de sus nodos.



(a) Variable de ventas original (b) Variable de ventas tras división

Figura 4.5: *Correspondencia de variable mediante división.*

- El número de variables se incrementa considerablemente, y cuantas más dimensiones y elementos haya en cada una, más rápido se produce. En el ejemplo del motor de metabúsqueda la variable bidimensional inicial da lugar a 12 unidimensionales. Si hubiera otro nivel más de anidamiento en el árbol con dos elementos (que podrían ser, por ejemplo, otros parámetros de búsqueda) se podrían obtener 24 variables. Si esto sucediera con muchas variables en la composición podría obtenerse una salida excesivamente grande en la que resultara difícil que una persona encontrara información de interés.

  Además, como se comentó en 4.2.5, las necesidades de espacio en memoria y tiempo de ejecución para Daikon podrían incrementarse considerablemente. En concreto, cuantas más variables haya, más grandes serán los ficheros con las trazas y en un orden aún mayor crecerán las combinaciones de invariantes a comprobar. Por lo tanto sería recomendable limitar el número de puntos de programa a instrumentalizar y/o la cantidad de variables a inspeccionar en ellos. Esto se puede efectuar manualmente o con ayuda las optimizaciones que se comentan en el capítulo 5.

- Las variables que se obtienen son *independientes* a todos los efectos, lo que impone limitaciones sobre el tipo de invariantes que se pueden obtener sobre ellas. Por ejemplo, Daikon no comprueba toda combinación de variables en todo punto del programa (por razones de complejidad computacional y rendimiento), limitándose a invariantes que relacionen una, dos o tres variables.

  Esto no implica que Daikon no sea capaz de generar ningún invariante relativo a una estructura multidimensional que se divida en más de tres variables. Daikon emplea por defecto una heurística que crea determinadas variables nuevas (en inglés las llama *derived variables*) en ciertos puntos del programa que considera de interés (puede ampliarse información en [Dai10]). Existen algunos invariantes que se pueden generar con variables derivadas nuevas, como por ejemplo *results.length*, que indica el número total de resultados, pero son casos muy concretos.

- Como limitación de este método, hay que destacar que no permite generar invariantes del tipo «todos los elementos son positivos» o «la suma de todos los elementos es X». Para obtenerlos sería necesario inferirlos manualmente o traducir usando otra técnica de reducción de la dimensión de árboles XML Schema.

4.3.3. Correspondencia mediante aplanado

WS-BPEL usa XPath [W3C07d] como lenguaje por defecto para describir los dos lados de asignaciones, condiciones booleanas para bucles, etc. XPath incorpora expresiones del tipo *pedido//artículo*, que devuelven la secuencia de todos los elementos *artículo* bajo cualquier nodo hijo *pedido* del nodo actual siguiendo el orden definido en el documento.

Esta es precisamente la idea que subyace bajo esta segunda forma de reducir la dimensionalidad de una matriz: reducirla a varios vectores unidimensionales de acuerdo a un determinado orden transversal. Este orden es el definido por el documento XML, por lo tanto los índices de los elementos de segundo nivel varían antes que los primeros: por ejemplo, se considera *pedido[1]/artículo[2]* un elemento anterior a *pedido[2]/artículo[1]*.

En la figura 4.6(b) se observa el resultado tras aplicarlo a la variable de la figura 4.6(a). Se puede ver que hay tantas variables como tipos de nodos hoja (en concreto dos: artículo y precio). La variable *venta.pedido.artículo* es un vector unidimensional con todos los artículos según su orden XML, mientras que *venta.pedido.precio* es similar pero con los precios. Al provenir los valores de distintas ventas y pedidos, en el nombre de las variables no aparece índices intermedios, como ocurría en la correspondencia por división (por ejemplo, *ventas[1].pedido[2]*).

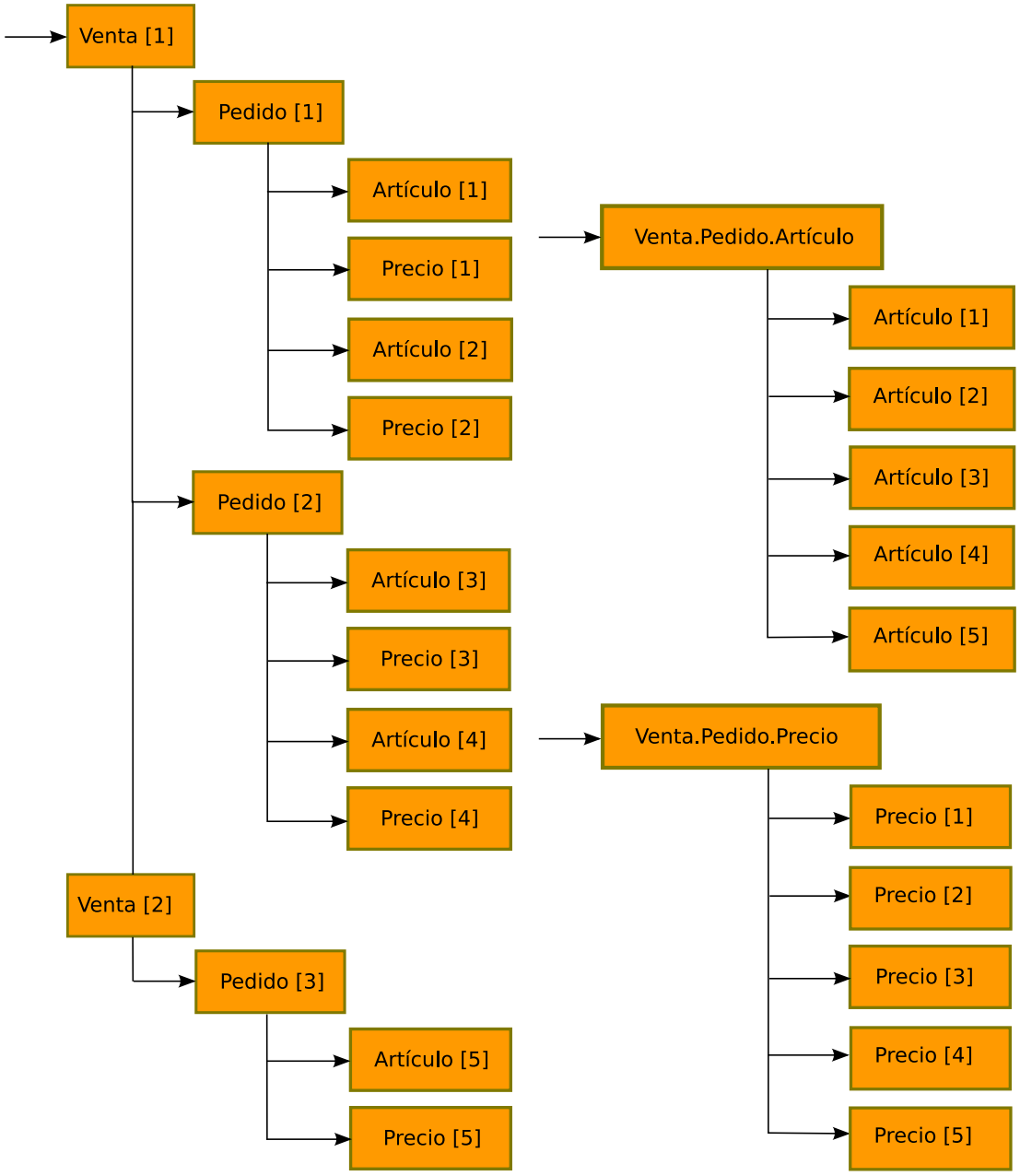
Su aplicación a la metabúsqueda se puede ver en el fragmento de código del listado 4.15 (o desglosando los elementos del vector en el listado 4.16). En ellos se observa que la variable bidimensional se aplana como 4 vectores unidimensionales. Cada uno de ellos tendría todos los nodos hoja de un tipo concreto, independientemente del padre al que pertenezcan.

Listado 4.15: *Correspondencia por aplanado en el registro de ejecución de la metabúsqueda.*

```
1  (...). noResult[1] = 3
2  (...). noFromGoogle[1] = 1
3  (...). noFromMSN[1] = 2
4
5  (...). result []. url = [ ''http: //[...] uca.es'' , ''http://cudisol [...] '' , ''http://jornadas [...] '' ]
6  (...). result []. title = [ ''OSLUCA'' , ''CUDISOL'' , ''Jornadas Andaluzas de Software Libre'' ]
7  (...). result []. snippet = [ ''Oficina [...] '' , ''Cursos [...] '' , ''Organizadas [...] '' ]
8  (...). result []. from = [ ''Google'' , ''MSN'' , ''MSN'' ]
```

Listado 4.16: *Vista alternativa del aplanado en el registro de ejecución de la metabúsqueda.*

```
1  (...). noResult[1]) = 3
2  (...). noFromGoogle[1]) = 1
3  (...). noFromMSN[1]) = 2
4
5  (...). result []. url[1] = ''http://softwarelibre.uca.es''
6  (...). result []. url[2] = ''http://cudisol.ourproject.org''
7  (...). result []. url[3] = ''http://jornadas.adala.org''
8
9  (...). result []. title [1] = ''OSLUCA''
10 (...). result []. title [2] = ''CUDISOL''
11 (...). result []. title [3] = ''Jornadas Andaluzas de Software Libre''
12
13 (...). result []. snippet[1] = ''Oficina de Software Libre, Universidad de Cadiz''
14 (...). result []. snippet[2] = ''Cursos para la Difusion del Software Libre''
15 (...). result []. snippet[3] = ''Organizadas por ADALA, CAGESOL y OSLUCA, Algeciras 2004''
16
17 (...). result []. from[1] = ''Google''
```



(a) Variable de ventas original (b) Variable de ventas tras aplanado

Figura 4.6: *Correspondencia de variable mediante aplanado.*

```
18  (...). result []. from[2] = ''MSN''
19  (...). result []. from[3] = ''MSN''
```

Las ventajas e inconvenientes de esta técnica contrastan claramente con los de la anterior:

- Si bien la correspondencia por división era la manera natural de obtener invariantes de elementos concretos de una estructura multidimensional, este método puede generar fácilmente invariantes relativos a todos los elementos de un determinado tipo en la estructura original. Por ejemplo, permitiría demostrar que todos los resultados provienen de Google o MSN.

- También permite obtener invariantes de algunos elementos de los vectores unidimensionales, pero solo aquellos que la heurística de Daikon seleccione. Por lo tanto, sería necesario cambiar la configuración por defecto de Daikon para asegurarnos que se chequean determinados invariantes que puedan interesar.

- Esta aproximación «rompe» la estructura del árbol original y, por lo tanto, limita mucho el uso de información XML Schema para eliminar invariantes redundantes (que se verá en el capítulo 5).

  Esto se puede ver, entre otras restricciones, en las longitudes de las variables. Las longitudes mínimas se pueden comprobar directamente, porque el número mínimo de apariciones de un elemento en una variable resultante de un aplanado es la suma de los mínimos que impone cada padre. Por ejemplo, si una variable tuviera una estructura con un campo que debe aparecer mínimo dos veces, y hubiera cuatro apariciones de dicha estructura, se podría afirmar que, al aplanar, el mínimo de apariciones del campo sería ocho. Así que, si esa información se derivara como invariante, se podría eliminar. Esto es posible para las longitudes mínimas porque XML Schema impone un mínimo por defecto a cada campo: si es optativo puede no aparecer, si es obligatorio debe aparecer una vez al menos.

  Sin embargo, esto no es posible para las repeticiones máximas, porque puede definirse un campo con un número indeterminado de apariciones. Es más, ese es el significado por defecto si no se indica un máximo concreto, algo muy común en los desarrollos WS-BPEL (que suelen confiar en que el programador haga buen uso de las variables en su código y no especifican límites).

- Sin embargo, esta técnica tiene la ventaja de que el número de variables se mantiene constante sobre el número de elementos y el nivel de anidamiento. Esto es muy importante, pues en la correspondencia por división estos dos parámetros hacían crecer significativamente la cantidad de variables.

  Dado que es más eficiente en tiempo y espacio quizás sea razonable, para el caso general, usar este método por defecto y que se use el otro sólo si el usuario desea obtener información concreta que quede fuera del alcance de este.

4.3.4. Combinación de ambos métodos

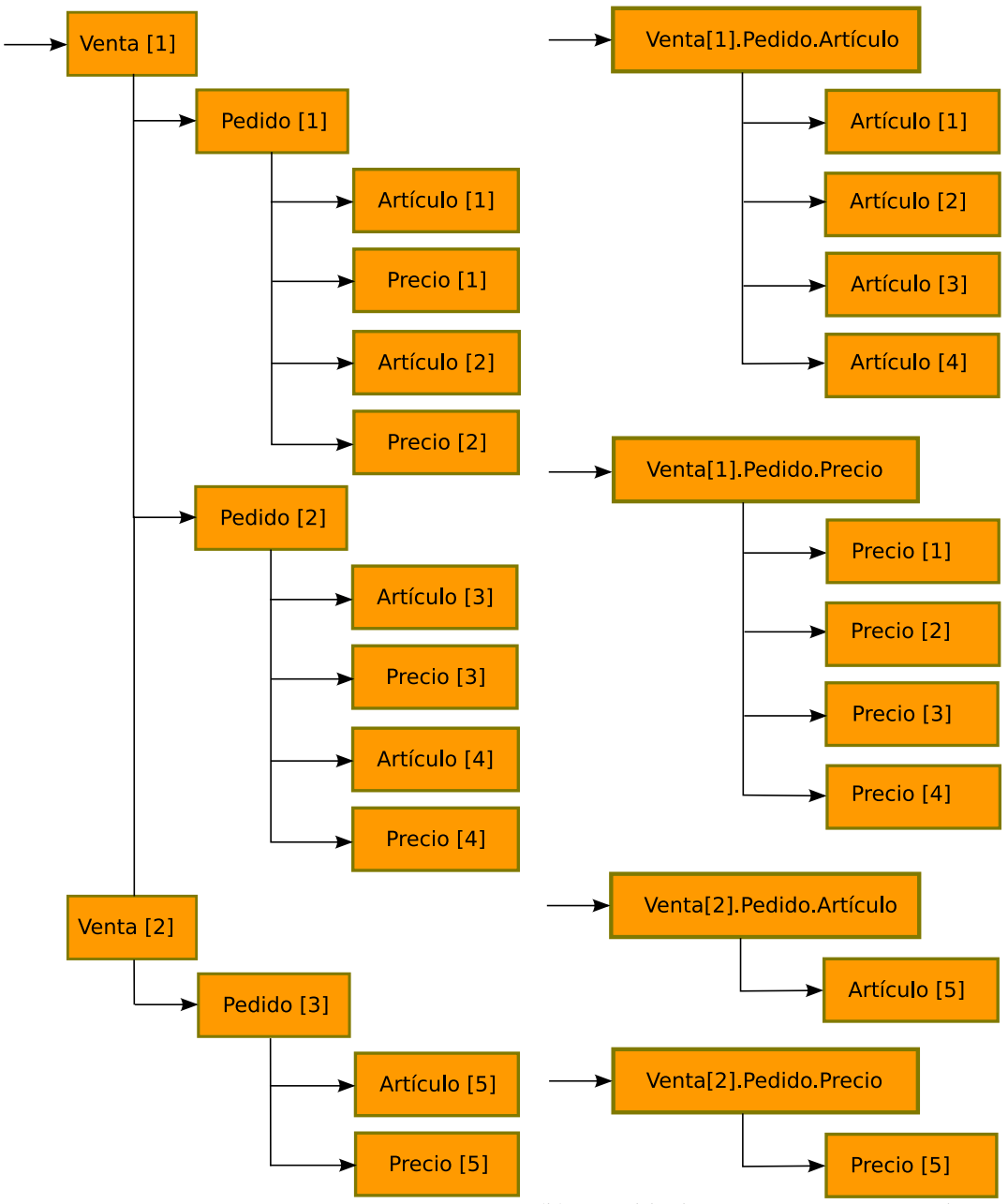
Aunque para la composición de la metabúsqueda y otras muchas las dos técnicas de correspondencia entre árboles XML y vectores unidimensionales son suficientes, por lo general, podría no ser siempre así. Por ejemplo, podrían existir composiciones que incluyeran variables tridimensionales como la de ventas de artículos usada de ejemplo en las secciones anteriores. En ella, hay una estructura del tipo *ventas[].pedido[].artículo[]* para la que podría interesar obtener invariantes de todo elemento *artículo* bajo *ventas[1]* independientemente de a cuál de sus *pedidos* en concreto pertenezca.

Con correspondencia mediante división sólo se generarían invariantes para los elementos *artículo* de una pareja de *ventas* y *pedido* concretos. Mientras que con aplanado se tendrían todos los *artículos* dentro de un mismo vector. Así que para obtener lo que se desea sería necesario combinar ambos: dividir el primer nivel del árbol (separando la variable original en *ventas[1]* y *ventas[2]*) y aplanar el segundo nivel (agrupando todos los nodos hoja de cada tipo en el mismo vector). En la figura 4.7(b) se observa el resultado tras aplicar esta correspondencia mixta a la variable de la figura 4.7(a). En ella se ven que el árbol inicial se dividió en dos variables (*venta[1]* y *venta[2]*), y dentro de cada una variable por cada tipo de nodos hoja (en concreto dos: artículo y precio). Esta aproximación mixta se observa claramente en los nombres de las variables: *venta[1].pedido.artículo* es un vector unidimensional con todos los artículos de la primera venta según su orden XML, mientras que *venta[2].pedido.artículo* es similar pero con los de la segunda. Al estar los valores aplanados por pedidos, el índice de estos se pierde en los nombres de variables.

Generalizando, se podrían aplicar distintas técnicas a cada dimensión de cada matriz. Por ejemplo, a una variable se le podría aplicar división para la primera dimensión y aplanado para la segunda, mientras que otra podría dividirse usando únicamente división. También se podría complementar con una selección manual de correspondencia más fina usando algún lenguaje de selección de información de árboles XML (como XPath). De este modo, por ejemplo, se podría seleccionar las cinco primeras apariciones de un campo en una variable XML Schema y aquellos campos que empezaran por una letra concreta para hacer una variable Daikon. Esta correspondencia permitiría un estudio mucho más fino, aunque requeriría intervención humana. Estos estudios quedan como línea de trabajo futuro.

4.3.5. Análisis de resultados

A continuación se comentan algunos invariantes obtenidos aplicando ambas correspondencias a la composición de la metabúsqueda. Hay que destacar que debido a su tamaño (tiene más de 500 instrucciones) sale un número bastante elevado de campos de variable registrados: más de 17.000 con división y más de 11.000 con aplanado. Esto provoca que se superen los 30.000 invariantes con correspondencia por división y los 18.000 con aplanado.



(a) Variable de ventas original

(b) Variable de ventas tras correspondencia mixta

Figura 4.7: *Correspondencia mixta de variable.*

Mediante una selección de invariantes obtenidos con correspondencia por división (listado 4.17) y con aplanado (listado 4.18) se muestran algunas de las diferencias entre ambos métodos. Los invariantes de las líneas 1 de cada listado se dan al comenzar a procesar la solicitud de búsqueda. Mientras que los de las líneas 3 se cumplen al terminar de procesar los resultados de Google. Por último, el resto de invariantes se infieren a la salida de la instrucción <sequence> principal de la composición (justo antes de responder al cliente).

Listado 4.17: *Selección de invariantes obtenidos mediante correspondencia por división.*

```
1  (...). MetaSearchProcessRequest[1].query[] elements == "Philipp␣Lahm"
2
3  maxGoogle one of { 0, 1, 3 }
4
5  (...). result [1]. from[] one of { [Google], [MSN] }
6  (...). result [2]. from[] one of { [Google], [MSN] }
7  (...). result [3]. from[] elements one of { "Google", "MSN" }
8  (...). result [4]. from[] elements == "MSN"
9  (...). result [5]. from[] elements == "Google"
10 (...). result [6]. from[] elements == "MSN"
11
12 size ((...). result [3]. from[]) one of { 0, 1 }
13 size ((...). result [4]. from[]) one of { 0, 1 }
14 size ((...). result [5]. from[]) one of { 0, 1 }
15 size ((...). result [6]. from[]) one of { 0, 1 }
```

Listado 4.18: *Selección de invariantes obtenidos mediante correspondencia por aplanado.*

```
1  (...). MetaSearchProcessRequest.query[] elements == "Philipp␣Lahm"
2
3  maxGoogle one of { 0, 1, 3 }
4
5  (...). result .from[] elements one of { "Google", "MSN" }
```

Analizando los invariantes de las líneas 1 vemos que ambos informan de que el término que se ha solicitado buscar en todas las peticiones es el mismo, *Philipp Lahm* (desarrollador de BPELUnit). De hecho, los invariantes son prácticamente idénticos con ambas técnicas: esta situación se suele dar cuando no hay repeticiones de nodos hoja en una variable. Después, en las líneas 3, se observan dos invariantes idénticos. Son invariantes de una variable unidimensional, que no ha necesitado que se le aplique correspondencia alguna.

Donde sí se observan diferencias más interesantes es en aquellos invariantes obtenidos a la salida del <sequence> principal de la composición. Los resultados de ambos métodos son similares, pero no son en absoluto iguales. Con correspondencia por división se comprueba que en el único caso de prueba que devuelve 6 resultados, los últimos que provienen de Google y MSN se alternan tal y como era de esperar. De hecho, los invariantes de las líneas 12 a 15 obtenidos con división (que no tienen equivalente en aplanado) indican que existen casos de prueba que solo tienen dos resultados, pues afirman que la longitud de los elementos del tercero al sexto puede ser cero (es decir, que

pueden estár vacíos), y no existen otros invariantes que lo indiquen para los elementos primero y segundo.

Sin embargo, hay que destacar que solo observando este punto del programa no se puede concluir nada sobre la alternancia de Google y MSN como fuentes de los primeros 3 resultados. Para ello habría que observar los invariantes que se cumplen en los puntos del programa que se sabe solo se ejecutan cuando hay al menos un resultado de Google o MSN. También se observa que no hay ningún caso de prueba que no devuelva ningún elemento, pues si fuera así la lista vacía estaría dentro de los valores de *result[1].from[]*: este es un buen ejemplo de cómo Takuan permite encontrar deficiencias en un conjunto de casos de prueba.

En resumen, la decisión de usar una u otra correspondencia dependerá de dos factores: primero, de si se buscan invariantes sobre elementos individuales o sobre una secuencia de ellos, y segundo de las restricciones de espacio y tiempo impuestas sobre ellos. Por su propia naturaleza, el correspondencia por aplanado necesita menos recursos que la división, dado que la cantidad de variables no depende de la dimensión y cardinalidad de la entrada. Además, la correspondencia por aplanado produce bastante menor cantidad de invariantes que la división, y estos suelen ser más fuertes y abstractos, pero menos detallados. Por lo tanto, para analizar una composición WS-BPEL en la que no se busque ningún invariante concreto, se podría usar una primera aproximación con aplanado. Se examinaría su salida y después se seleccionarían determinados puntos de programa y variables para obtener información más detallada con una segunda ejecución aplicando correspondencia por división.

4.3.6. Rendimiento

En este apartado se analiza cómo se comporta Takuan con composiciones que incluyen gran cantidad de puntos de programa y variables complejas. Para ello se ha realizado una prueba con la composición de la metabúsqueda ejecutando solo 7 casos de prueba. Se han generado invariantes para todas las combinaciones de las siguientes opciones:

Correspondencia por división y aplanado.

Variables registradas todas las variables o bien solo aquellas que aparecen en la respuesta de la composición. En el primer caso Daikon genera todos los invariantes que puede sobre cualquier conjunto de variables. Mientras que en el segundo caso se simula el conocimiento de un humano que desea obtener sólo invariantes relacionados con determinadas variables.

Puntos del programa a analizar analizando todos los puntos de programa o solo los tres primeros niveles de profundidad según la definición de actividades del proceso WS-BPEL. Con la primera opción se registrarán los valores de las variables seleccionadas en todas las instrucciones de bloque (<sequence> o <flow>) de

Cuadro 4.1: *Tiempo de ejecución por correspondencia, punto de programa y selección de variables.*

Correspondencia	Puntos de prog.	Variables	Tiempo (mm:ss)
División	64 (todos)	17.404 (todas)	7:18[a]
		4.720 (selec.)	1:19
	12 (3 niveles)	2.240 (todas)	0:43
		704 (selec.)	0:22
Aplanado	64 (todos)	11.412 (todas)	3:46
		3.888 (selec.)	1:01
	12 (3 niveles)	1.560 (todas)	0:28
		624 (selec.)	0:19

[a] Fue necesario poner el tamaño máximo de la pila del JVM a 800 MiB para evitar una actividad excesiva del recolector de basura. Esto provocó cierta distorsión debido a un menor intercambio de memoria.

la composición. Por contra, la segunda solo considera los tres primeros niveles anidados de ellas. Es decir, se registra toda instrucción <sequence> o <flow>, las que estas contengan y las que estas últimas a su vez incluyan. Pero en caso de que recursivamente aparezcan más niveles, ya no se registran. De esta forma se simula el conocimiento de un desarrollador que elige una serie de puntos del programa donde cree que se obtendrán los invariantes que le interesan.

Los resultados de la ejecución de la composición con cada posible combinación de estos tres aspectos se encuentran en el cuadro 4.1.

Como era de esperar, hay una diferencia significativa en el tiempo de ejecución entre correspondencia por división y por aplanado cuando se analizan todas las variables de la composición. Por ejemplo, con división, los 7 casos de prueba tardaron más de 7 minutos en ejecutarse consumiendo más de 800 MiB de RAM. La razón principal es la gran cantidad de combinaciones de pares y tripletas de variables que Daikon chequea buscando invariantes en cada punto del programa. Sin embargo, al hacer una selección más fina de la entrada, ambos métodos tardan un tiempo de orden similar, aunque el aplanado sigue siendo un poco más rápido.

Igualmente queda claro que al limitar el número de niveles (lo que reduce el número de puntos de programa donde se inspeccionan variables) el tiempo de ejecución es significativamente menor. Sin embargo, la selección de puntos del programa tiene que realizarla un humano con conocimiento del programa, para evitar descartar invariantes de interés en los puntos no instrumentalizados.

Por contra, al ver el número de invariantes se puede concluir fácilmente que son cifras demasiado altas para considerar que una persona las pueda revisar manualmente.

Este análisis cualitativo de la salida permite afirmar que sería deseable implementar algunas optimizaciones ya apuntadas en el apartado 4.2.5. De este modo se podrían reducir considerablemente la cantidad de invariantes, mitigando además las consecuencias de una selección manual que incluya demasiados puntos del programa y/o variables.

Estas optimizaciones se comentan en el capítulo 5 y consisten básicamente en descartar variables no usadas en determinados puntos del programa y comparar variables únicamente del mismo tipo abstracto.

CAPÍTULO 5

Optimizaciones en Takuan

La información recopilada en el análisis de los invariantes de Takuan permite detectar aspectos a mejorar en el sistema. En concreto, aparecen invariantes sin sentido debido a que existe mucha información de comparabilidad en la composición que no se aprovecha, y la estructura de variables compartidas entre ámbitos en WS-BPEL hace que algunas variables estén disponibles en muchos puntos del programa donde no se usan. Además, el sistema pierde tiempo en inferir invariantes que no proporcionan información al usuario, pues ya estaban contenidos en la definición de tipos de datos XML Schema. En este capítulo se muestra cómo Takuan subsana estas deficiencias [Pal09c].

5.1. Comparabilidad de variables en WS-BPEL

En primer lugar se va a usar la información de comparabilidad que hay implícita en la composición para indicar a Daikon qué variables deben intentar relacionarse en un invariante. Como se observó en el apartado 4.2.3, Daikon acepta un *índice de comparabilidad* (del inglés *comparability index*) para saber si dos variables del mismo tipo pertenecen también a un mismo tipo abstracto (y, por lo tanto, pueden aparecer juntas en un invariante). Por ejemplo, una cantidad monetaria, la edad de una persona y los años para devolver un préstamo pueden ser todos de tipo entero. Pero solo la edad y los años pertenecen al mismo tipo (años naturales), por lo que no parece de interés generar invariantes que los relacione con la cantidad. Mediante este mecanismo se le puede indicar qué variables tiene sentido relacionar en un invariante, ahorrando de este modo tiempo de cómputo y eliminando información no interesante de la salida.

Para ello, se etiqueta cada variable con su índice de comparabilidad: un identificador (normalmente un número entero) que agrupa las variables del mismo tipo de dato que pertenecen a un mismo tipo abstracto, como una cantidad monetaria o años naturales. De este modo, Daikon sólo relacionará en invariantes variables que tengan el

mismo tipo abstracto, reduciendo en el caso promedio la explosión combinatoria que se produciría en el tiempo de ejecución de Takuan.

5.1.1. Justificación

Aunque existen herramientas que calculan índices de comparabilidad para lenguajes como Java o C++, no se pueden aplicar directamente a WS-BPEL. Esto es debido a que la definición de los campos de los procesos WS-BPEL la interpreta el motor, por lo que algunas técnicas de bajo nivel como la instrumentalización a nivel de bytes o el acceso a memoria que funcionan en Java o C/C++ no son aplicables.

La aproximación usada por Lackwit en [O'C97] para comprobar estáticamente qué campos de qué variables están relacionados en instrucciones de una composición WS-BPEL es difícil, dado que las expresiones XPath que acceden a ellos pueden navegar por los contenidos de las variables (que pueden tener estructura arbórea) de diversas maneras [W3C07d]. Por lo tanto, Takuan implementa una aproximación dinámica, de manera similar a como hicieron los autores de la herramienta de análisis de comparabilidad para Java DynComp [Ern00b].

Además, hay que tener en cuenta que no todo par de campos (de la misma o distinta variable) accedidos en una expresión tienen que ser comparables, sino que depende de la relación entre el uso de ambos. Por ejemplo, en la expresión $X = A$ *and* $Y = B + C$, se puede afirmar que X y A, al igual que Y, B y C están relacionados. Pero no así X y B.

5.1.2. Implementación

Se extendieron los pasos de instrumentalización y ejecución de Takuan (incluyendo modificaciones del propio motor ActiveBPEL) de modo que los registros incluyeran todas las variables que realmente se acceden en la ejecución de cada caso de prueba. Para ello se amplió la extensión de XPath encargada del registro *reg:inspect* para que durante la ejecución no solo registre los valores de las variables, sino también su ámbito.

Posteriormente, se definieron un conjunto de reglas que dividen recursivamente una expresión en *ámbitos de comparabilidad* (*comparability scopes* en inglés). De este modo, Takuan considera que dos variables están relacionadas si pertenecen a un mismo ámbito (teniendo en cuenta que una variable puede pertenecer a más de uno). Inicialmente existe un único ámbito al que pertenecen todos los elementos de cada expresión, y se divide al encontrar:

- Un argumento en una llamada a función. Por ejemplo, en la expresión $Y = X +$ *saldo(NUM-CLIENTE, NUM-CUENTA)* se puede deducir que Y y X pertenecen a un mismo tipo, mientras que *NUM-CLIENTE* pertenece a otro y *NUM-CUENTA* a un tercero.

  Sin embargo, existen excepciones en algunas funciones internas de XPath que implican una relación semántica entre sus parámetros. Por ejemplo, en las funciones

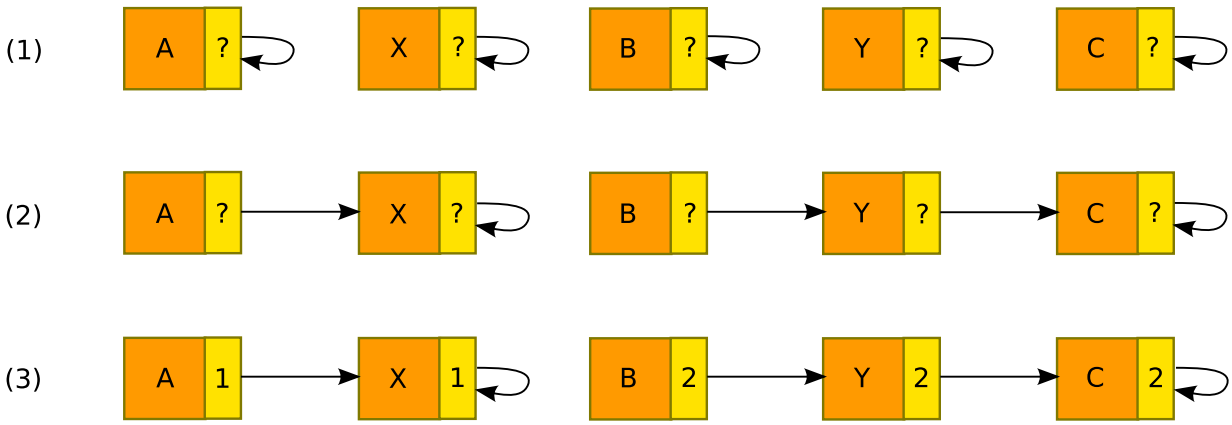


Figura 5.1: *Ejemplo de etiquetado de variables relacionadas semánticamente.*

que calculan valores mínimos o máximos de una lista todos los elementos de la lista son comparables. En concreto, en $max(X,Y) > 8$ se crearía un único tipo para X e Y.

- Un operador lógico, como *and*, *or* o *not*.
- Predicados de filtrado de conjuntos de nodos, como $total > 10$ en $compra[total > 10]$, que se dividiría en un tipo para *compra* y otro para *total* y 10.

En la figura 5.1 se muestra de manera conceptual el proceso para el ejemplo $X = A$ *and* $Y = B + C$. En cada punto del programa:

1. Se crea un grafo no dirigido sin aristas y un vértice por cada campo de una variable que se encuentre (como se observa en el paso 1 de la figura).
2. Se conectan los vértices que representan campos del mismo ámbito para cada expresión XPath evaluada en ese punto del programa, como se observa en el paso 2 de la figura.
3. Los campos de variables de cada componente no conectada se etiquetan con un índice de comparabilidad distinto (véase el paso 3 de la figura).

Sin embargo, todavía esta información es incompleta, porque recoge la comparabilidad en cada evaluación de cada instrucción XPath en un determinado caso de prueba. Se necesita generalizar esta información con la información del resto de casos, porque puede ser que dos o más de ellos tengan comparabilidades relacionadas. Por ejemplo, sea la expresión $precio[X] > A$ y dos casos de prueba, uno en el que al evaluar la expresión X valga 1 y otro en el que X tome el valor 8. Habría que incluir en el mismo ámbito A, $precio[1]$ y $precio[8]$, pero no el resto de miembros del vector.

Los índices de comparabilidad que se obtienen de este modo se utilizan, en primer lugar, para que Daikon restrinja su búsqueda a invariantes que relacionen variables de

un mismo tipo abstracto. Pero además, esta información se puede usar para implementar una optimización más potente: descartar, en cada punto del programa, todas las variables (o campos de ellas) no usadas en él. En el apartado 5.3 se estudiará de manera separada el impacto de estas dos optimizaciones.

5.2. Restricciones XML Schema

Tras aplicar Takuan a varias composiciones, se observó otra ineficiencia: una cierta cantidad de los invariantes obtenidos no proporcionaban información de interés para el usuario. En concreto, bastantes de ellos solo confirmaban propiedades ya incluidas en la declaración de los tipos de datos XML Schema usados en la composición.

5.2.1. Justificación

XML Schema es un lenguaje muy rico que permite especificar propiedades tales como longitudes mínimas o máximas de vectores y cadenas o rangos de valores válidos para variables numéricas entre otras. Por lo tanto, si en un ejemplo, la declaración del XML Schema indica que debe de producirse al menos un artículo, se puede considerar el invariante *numero(articulos)* $>= 1$, redundante y descartarlo. Es más, si la variable *articulos* tiene un contenido multidimensional y se usa correspondencia por aplanado, esta podría ser traducida posteriormente a n variables unidimensionales, por lo que se estaría evitando generar no uno, sino n invariantes irrelevantes.

Además, esta técnica puede extenderse a otros lenguajes de programación que incluyan restricciones de tipos similares. Por ejemplo, vectores con tamaño estático en C/C++, cadenas de tamaño fijo en FORTRAN, o campos *VARCHAR(N)* en SQL. Por lo tanto, es importante implementar esta técnica de modo que sea lo más independiente del lenguaje posible.

5.2.2. Implementación

Hay tres momentos en que se puede reducir la cantidad de invariantes que Daikon genera: antes de invocarlo (eliminándolos de su entrada), durante su ejecución (evitando comprobarlos) o una vez se tiene la lista de invariantes a su salida.

Evidentemente lo ideal es descartar los invariantes redundantes cuanto antes. Así se evita una pérdida de rendimiento que se puede extender a varias fases de la generación de invariantes. Sin embargo, a diferencia de la generación de índices de comparabilidad, la información de XML Schema no permite descartar variables antes de invocar a Daikon. Y hacer un filtrado en la salida no aporta ninguna ganancia temporal en el rendimiento.

Por lo tanto, la opción más adecuada es extender Daikon. La modificación realizada permite que acepte información adicional sobre cada variable con restricciones obteni-

das externamente, y usarla para evitar comprobar invariantes que se deriven de ella durante la ejecución. Así se consigue que la solución no sea únicamente para WS-BPEL.

En el caso de WS-BPEL, esta información es la contenida en la declaración de tipos XML Schema. En concreto: longitudes máximas y mínimas de vectores y cadenas, valores máximos y mínimos de variables numéricas y listas de valores admitidos en tipos enumerados.

5.3. Resultados

En el listado 5.1, que extiende el listado 4.6 del préstamo bancario versión sequence visto anteriormente en la página 51, se puede observar cómo se amplía el fichero de declaraciones para incluir información que permita realizar las dos optimizaciones comentadas anteriormente. Nótese que en este fichero se han incluido otras variables registradas además de *processOutput.output.accept* para observar las relaciones que se pueden establecer entre ellas.

Listado 5.1: *Extracto simplificado de fichero de declaraciones* `.decl` *con tipos abstractos.*

```
1  DECLARE
2  LoanApproval.SmallAmountLowRisk:::ENTER
3  processOutput.output.accept[]
4  xsd:boolean[] # minvalue=0, maxvalue=1, minlength=1, maxlength=1, validvalues=["0" "1"]
5  boolean[]
6  1
7  processInput.input.amount[]
8   xsd:float []  # minlength=1, maxlength=1
9   float []
10 3
11 assessorOutput.output.risk[]
12 tns:RiskType[] # minlength=1, maxlength=1, validvalues=["high" "low"]
13 java.lang.String[]
14 2
15 assessorInput.input.amount[]
16  xsd:float []  # minlength=1, maxlength=1
17  float []
18 3
19 approverOutput.output.accept[]
20 xsd:boolean[] # minvalue=0, maxvalue=1, minlength=1, maxlength=1, validvalues=["0" "1"]
21 boolean[]
22 1
23 approverInput.input.amount[]
24  xsd:float []  # minlength=1, maxlength=1
25  float []
26 3
```

En concreto, se observa cómo en las líneas 6, 10, 14, 18, 22 y 26 aparecen números enteros que indican el tipo abstracto de cada variable: el tipo abstracto 1 es para la respuesta de la composición, el tipo 2 para el riesgo del préstamo y el 3 para la cantidad

Cuadro 5.1: *Estadísticas de rendimiento para cada combinación de optimizaciones.*

Corresp.	Optimiz.[a]	P. prog.[b]	Variables	Memoria[c]	Tiempo[d]	Invariantes
División	Nada	64	17.404	656,74	409,98	30.399
	X			646,57	400,15	21.793
	C	48	14.148	561,63	416,72	27.089
	CX			579,81	401,69	18.358
	CF		1.398	25,25	72,96	2.135
	CXF			24,54	75,03	1.559
Aplanado	Nada	64	11.412	291,11	162,00	18.658
	X			280,96	173,40	18.654
	C	48	9.036	261,01	179,39	16.718
	CX			264,54	163,90	16.714
	CF		710	11,18	52,29	942
	CXF			12,29	55,61	940

[a] X: supresión de invariantes en el XML Schema, C: índices de comparabilidad, F: filtrado de variables no usadas.

[b] Tras eliminar durante el análisis de comparabilidad los puntos del programa sin expresiones XPath.

[c] Uso máximo que hace Daikon de la pila del JVM, en MiB.

[d] Tiempo consumido por la fase de análisis de Takuan (incluyendo el preprocesado), en segundos.

que se solicita prestada. Además, cada variable lleva, después de su tipo original una almohadilla y una serie de restricciones XML Schema conocidas. Por ejemplo, en la línea 4 se indica que las cantidades son escalares (con longitud mínima y máxima uno), y que la respuesta solo puede ser 0 o 1.

Se ha estudiado el impacto de cada una de estas dos técnicas en la composición de metabúsqueda usada anteriormente y detallada en el apéndice A.2.1. Esta composición busca webs relacionadas con un término indicado por el usuario usando dos SW, y ofrece los resultados de estos intercalados al usuario. Se usaron los mismos 7 casos de prueba que en el apartado 4.3.6, aplicándoles cada una de las técnicas por separado y en combinación, considerando el descarte de variables no usadas como una variante de la generación de índices de comparabilidad.

En el cuadro 5.1 se resume el impacto de cada posible combinación respecto a diversos factores. En las filas están las distintas opciones de funcionamiento de Takuan, mientras que en las columnas los resultados. Por ejemplo, la mitad superior de las filas muestran las pruebas realizadas con correspondencia de variables mediante división, y la mitad inferior usando aplanado. En cada caso, las filas muestran, de arriba a abajo, los

resultados de la ejecución de la composición de metabúsqueda sin ninguna optimización, con supresión de invariantes en el XML Schema, con índices de comparabilidad, combinando estas dos técnicas, combinando el uso de índices de comparabilidad con filtrado de variables no usadas y aplicando las tres técnicas.

Para cada caso, se incluyen cuatro columnas con el número de puntos de programa y las variables registradas en ellos (lo que indica el tamaño de la entrada de Daikon), los recursos consumidos durante el análisis (en memoria principal y tiempo) y el tamaño del conjunto de invariantes obtenidos a la salida. En los siguientes apartados se analizan los resultados de cada caso.

Las pruebas se realizaron sobre la máquina usada en la sección 4.3.6. El consumo máximo de memoria que hace el script Perl se mantiene alrededor de los 193MiB en todas las entradas del cuadro.

5.3.1. Análisis del uso de índices de comparabilidad

Al analizar los resultados de la generación de índices de comparabilidad no debe olvidarse que esta aproximación es algo más agresiva que otras que podrían usarse, dado que elimina los puntos del programa donde no se usan variables. Esto se debe a que se supone que dichos puntos no van a contener invariantes con información útil, sirviendo simplemente para controlar el flujo de la composición. Esto explica la reducción de 64 a 48 puntos de programa cuando se activa el uso de índices de comparabilidad, y la consiguiente variación en la cantidad de variables.

Por otro lado también se observa una disminución en el uso de memoria, mientras que el tiempo de ejecución se incrementa ligeramente. Quizás la cantidad de variables no se decrementa tanto como cabía esperar, pero un análisis más detallado nos hace ver que han cambiado su naturaleza. Anteriormente se obtenían invariantes que relacionaban variables sin conexión semántica en la composición, mientras que ahora se obtienen invariantes disjuntos para cada tipo definido por cada índice. Esto hace que se obtenga una lista de invariantes más informativa y fácil de entender.

Por ejemplo, en la metabúsqueda, si los resultados obtenidos de MSN no incluyen marca temporal (*timestamps*), la versión sin información de comparabilidad detectaba que tanto *result.day* como *result.hour* eran vacíos, e infería el invariante *result.day = result.hour*. Sin embargo, al aplicar esta técnica este último invariante sin sentido se eliminaría en todos los puntos de programa donde apareciera.

5.3.2. Análisis del uso del filtrado de variables no usadas

Al generar la información de comparabilidad se pueden eliminar aquellos campos de variables que no se usen en cada punto del programa. Esto es especialmente útil en lenguajes como WS-BPEL donde no existe el concepto de función, y que, por lo tanto tienen que implementar la colaboración entre actividades mediante variables compartidas. Esto implica que en la mayoría de las ocasiones las variables, a pesar de estar accesibles

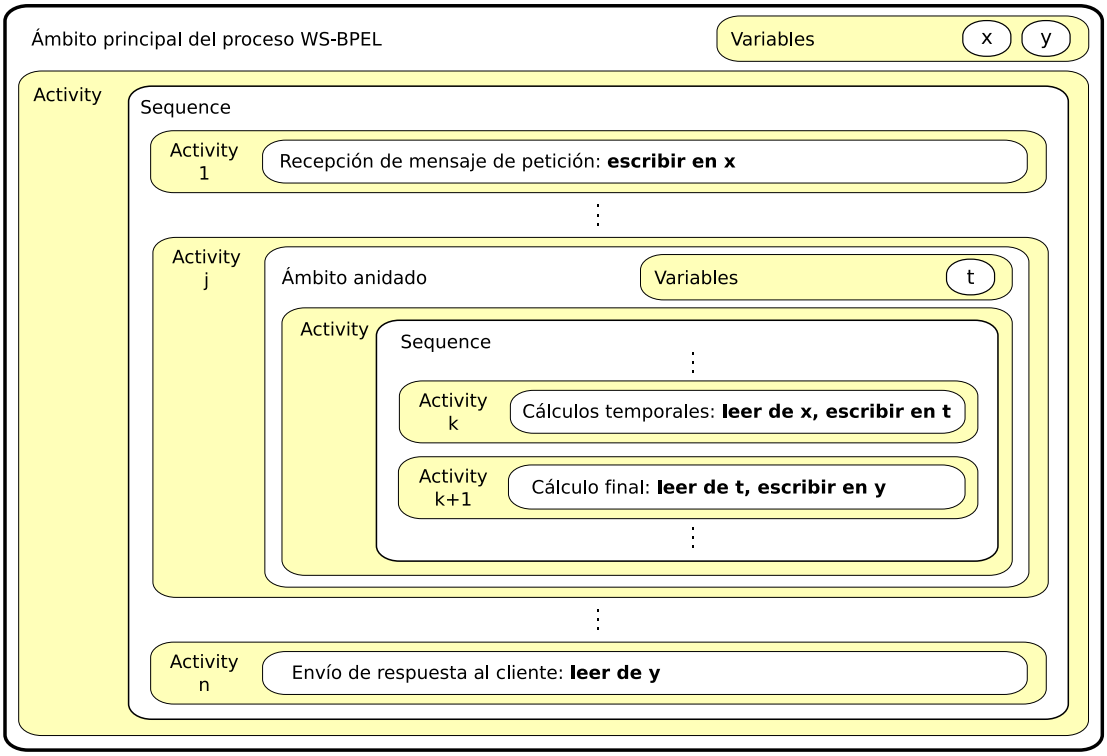


Figura 5.2: *Comunicación entre actividades mediante variables compartidas.*

desde muchos puntos del programa, solo se usan en un número bastante reducido de ellos (normalmente dentro de un determinado ámbito). Por lo tanto, si dos puntos del programa necesitan acceder a una determinada variable global (o a un campo de ella), solo debería registrarse su valor en ellos dos, no en otros puntos.

Se puede ver un ejemplo en la figura 5.2, que muestra los ámbitos que genera el código del listado 5.2. En él, la variable *y* está disponible en todas las instrucciones del programa. Pero se observa que registrar sus valores en puntos donde no se lea o modifique muy probablemente solo aumente el tiempo de procesamiento de Takuan y produzca invariantes de poca utilidad.

Listado 5.2: *Fragmento simplificado de código WS-BPEL con variables compartidas.*

```
1  <sequence>
2    <receive  [...]  variable="x"/>
3     [...]
4    <scope>
5      <sequence>
6        <assign>
7          <copy>
8            <from>[...] x  [...] </from>
9            <to> [...]  t  [...] </to>
10         </copy>
11       </assign>
```

```
12          <assign>
13            <copy>
14              <from>[...]  t   [...] </from>
15              <to> [...]  y   [...] </to>
16            </copy>
17          </assign>
18        </sequence>
19      </scope>
20       [...]
21      <reply  [...]  variable="y"/>
22  </sequence>
```

Esta es la razón por la que el análisis de comparabilidad incluido en Takuan puede eliminar (si así se desea) los campos de variables que no se usan en cada punto del programa. Como se observa en el cuadro 5.1, esto provoca una drástica reducción de la cantidad de variables, un menor tiempo y espacio necesarios para procesarlas y un conjunto de invariantes menor a la salida. Por ejemplo, usando correspondencia por división, esta técnica reduce el número de variables de 14.148 a 1.398 (menos del 10 %), y el de invariantes de 27.089 a 2.135 (por debajo del 8 %).

Sin embargo, esta aproximación también tiene sus desventajas. Principalmente puede producirse cierta pérdida de información, sobre todo en el caso de que dos o más actividades colaboren produciendo un invariante de alto nivel que las relacione y que pudiera ser de interés para el usuario. Por ejemplo, si solo se inspeccionaran variables locales dentro de un bucle, no se generaría sus pre-condiciones y post-condiciones. Una posible solución consistiría en que el usuario pudiera indicar a Takuan que considerara un punto del programa y todos sus puntos anidados como una sola entidad, mejora que queda como trabajo futuro.

5.3.3. Análisis del uso de restricciones del XML Schema

En cuanto al uso de restricciones incluidas en la especificación de tipos XML Schema para eliminar invariantes poco informativos, en primer lugar se puede afirmar que su uso no modifica el tamaño de la entrada de Daikon. Tampoco varían demasiado el tiempo de ejecución ni la memoria consumida. Pero sí que destaca la reducción en la cantidad de invariantes obtenidos, especialmente usando correspondencia por división: más de 9.000 invariantes han desaparecido. La mayoría de ellos eran comparaciones entre longitudes o límites de vectores incluidos en la declaración de tipos del XML Schema.

Por contra, al usar aplanado esta técnica no es tan efectiva, dado que las restricciones sobre las longitudes de los vectores se debilitan o incluso desaparecen al aplanar las variables. Para verlo más claro se puede analizar la variable de ventas descrita en la figura 4.4 (página 59). Supongamos que el tipo de dato de dicha variable tiene la restricción de que cada pedido puede tener un máximo de dos artículos (y, por lo tanto, dos precios).

Al aplicar correspondencia por división (figura 4.5 de la página 62), la variable original daría lugar en Takuan a n variables con artículos y otras n con precios. Cada una

de estas $2n$ variables podría contener dos elementos. Por lo tanto, se podrían eliminar hasta $2n$ invariantes de la salida de Daikon. Mientras que con aplanado (figura 4.6, página 65), solo aparecerían 2 variables, de las que solo se podría afirmar que el máximo de elementos en cada una de ellas es n. Es más, esta es una restricción mucho más débil, y que solo podría aplicarse a dos invariantes de la salida (que no siempre se tendrían que obtener, por ejemplo si algún caso de prueba no llega al tope de artículos).

Por lo tanto, se puede concluir que los pocos invariantes que se han eliminado de la composición de metabúsqueda con aplanado se deben a las restricciones de longitud mínima. De todas formas no parece que en general se deba descartar su uso con aplanado, dado que en casos en que se usen declaraciones de tipos XML Schema más estrictas para los aspectos implementados (longitudes máximas y mínimas de vectores y cadenas, valores máximos y mínimos de variables numéricas y listas de valores admitidos en tipos enumerados), sí que podría ser interesante.

CAPÍTULO 6

Experimentos

Tras demostrar que Takuan es un sistema maduro para generar invariantes de composiciones WS-BPEL, en este capítulo se hacen dos experimentos con él. En primer lugar se analiza la estabilidad y convergencia de su salida. Para ello, se demuestra que cuando aumenta el número de casos de prueba aleatorios los invariantes proporcionados se estabilizan. Después de este experimento cuantitativo se realiza otro cualitativo, comprobando que los conjuntos de casos de prueba que ofrecen mejor cobertura del flujo de ejecución de una composición proporcionan mejores resultados.

6.1. Estabilidad de los invariantes

Este primer experimento tiene por objetivo demostrar que la generación de invariantes con Takuan converge a medida que aumenta el número de casos de prueba aleatorios, y que existe un umbral a partir del cual aumentar el número de casos de prueba no produce ganancia en la cantidad ni calidad de los invariantes obtenidos [Gar08b].

Para ello se usan las composiciones del préstamo bancario sequence, el mercado de compraventa y dos versiones de la metabúsqueda, generando casos de prueba aleatorios para cada una. En cada caso se parte de un conjunto inicial pequeño, y se aumenta hasta que se estabiliza la salida de Takuan. Esta prueba está basada en la realizada en [Ern01].

6.1.1. Estructura del experimento

Para las composiciones del préstamo versión sequence y el mercado de compraventa 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. Se tomaron los invariantes inferidos con el conjunto más amplio como referencia para compararlos con los obtenidos usando los demás conjuntos. Para las metabúsquedas se hizo igual, pero con conjuntos de diez, cincuenta, cien, quinientos y mil casos de prueba.

Generación de casos de prueba aleatorios

Para comprobar la influencia del tamaño del conjunto de casos de prueba en la generación de invariantes se realizaron pruebas con las siguientes composiciones: el préstamo bancario, explicada en el apéndice A.1.1, la composición del mercado de compraventa del apéndice A.3 y dos versiones del motor de metabúsqueda documentadas en los apéndices A.2.1 y A.2.2. La composición del préstamo básicamente recibe peticiones de préstamo y las acepta o deniega según informen sobre ellas un servicio externo de asesoramiento de riesgo y otro de aprobación. En el apéndice A.1.1 se puede ampliar información sobre esta composición.

La segunda composición es un mercado de compraventa que recibe peticiones de venta y compra de un producto. Si le llega una petición de venta se crea una instancia de la composición que queda a la espera de que le llegue una oferta (y viceversa). Una vez la composición tiene una petición de venta y una oferta de compra comprueba si la cantidad ofrecida para comprar es mayor o igual que la pedida para la venta. Si es así, la composición envía mensajes a los dos socios para comunicarles que se puede realizar la transacción, y en otro caso les informa negativamente. Los detalles de la composición pueden consultarse en el apéndice A.3.

Las composiciones de metabúsqueda implementan un sistema de consulta de información en Internet usando dos servicios simples de búsqueda (uno de Google y otro de MSN). Para ello los invoca a ambos y ofrece al usuario sus resultados intercalados empezando por el primer resultado de Google si lo hubiera. La única diferencia entre las dos versiones es que una usa una instrucción <while> para iterar en los resultados y la otra una <forEach>. Se puede ampliar información sobre estas composiciones en los apéndices A.2.1 y A.2.2.

Como no se disponía de servicios reales, en todos los 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 euros en intervalos de 100 unidades. Las respuestas del servicio 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 (un takuan o rábano japonés) con cantidades comprendidas entre 0 y 999 euros. Aleatoriamente se les estableció un retraso de 0, 1 o 2 segundos independiente para cada uno de los mensajes enviados desde los socios, de modo que se crearan instancias de la composición tanto por la recepción de mensajes de ventas como de compras.

Por último, para la metabúsqueda, se crearon casos de prueba que buscaban webs relacionadas con una palabra aleatoria en inglés americano, y pidiendo un número alea-

torio entre 1 y 10 resultados de cada motor. Cada servicio simulado proporcionaba una cantidad aleatoria de webs entre 0 y el número máximo que había pedido el cliente. En el caso del servicio de Google, de cada resultado se proporcionaba una URL de la forma "http://" seguida de una palabra aleatoria, un título de entre 1 y 4 palabras aleatorias y una descripción de entre 1 y 10 palabras concatenadas. En cuanto a la respuesta del servicio MSN, se usaron de 1 a 4 palabras aleatorias para el título, una descripción de entre 1 y 10 de ellas, y otra palabra con el prefijo "http://" como URL. Las palabras aleatorias se tomaron del diccionario de inglés americano que se incluye en la distribución Ubuntu Jaunty 9.04 GNU/Linux en el fichero `/usr/share/dict/american-english`.

Métricas

A la hora de comparar los resultados hay que aclarar que Daikon clasifica los invariantes en *interesantes* y *no interesantes* según una heurística interna. Estos segundos son aquellos que hacen referencia a valores concretos de variables con listas de valores o intervalos, y que son obtenidos a partir de un número no significativo de casos de prueba que ejecutan dicho punto del programa. Si dicha cantidad no supera un determinado umbral, Daikon considera que pueden ser invariantes falsos que aparecen debido a limitaciones de la muestra proporcionada. Por ejemplo, si en la composición del préstamo se solicitan préstamos por valor de 1.000, 1.400, 2.800 y 3.000 euros es posible que Daikon informe que el valor de la cantidad solicitada está siempre entre 1.000 y 3.000. Pero su heurística probablemente considere que añadiendo más casos de prueba (por ejemplo, uno que solicitara 3.100 euros) se falsificaría dicho invariante.

La comparación de cada ejecución de Takuan con la ejecución de referencia contiene los siguientes datos:

- Número total de invariantes obtenidos por la ejecución.
- Número total de diferencias entre los conjuntos de invariantes interesantes.
- Número total de diferencias entre los conjuntos de invariantes completos (incluyendo los etiquetados por Daikon como no interesantes).
- Número total de puntos del programa donde se generan invariantes.
- Número total de puntos del programa donde se generan invariantes interesantes distintos.
- Número total de puntos del programa donde se generan invariantes distintos (incluyendo también lo etiquetados como no interesantes).
- El tiempo total empleado, que es el tiempo desde que se invoca Takuan hasta que se obtiene su respuesta final.
- El tiempo dedicado a ejecutar código del programa, que es el *tiempo de CPU de usuario*.

- El tiempo dedicado a llamadas al sistema, que es el *tiempo de CPU de sistema*.

Conviene aclarar que 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.

6.1.2. Resultados

Estas pruebas se realizaron en un equipo Intel Pentium 4 a 1.6 GHz con 512 MB de memoria RAM. En dicha arquitectura se ejecutó Takuan usando una máquina virtual 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 el equipo no es demasiado potente y la ejecución sobre una máquina virtual alarga un poco el tiempo de ejecución, la intención no es estudiar los tiempos de ejecución de Takuan de manera absoluta, pues han sido tratados detalladamente en el capítulo 4. Por contra, el objetivo de recopilar información de los tiempos es su interés comparativo.

Los resultados de las pruebas en la composición del préstamo bancario se pueden observar en el cuadro 6.1 en la página siguiente, y los del mercado de compraventa están en el cuadro 6.2. La primera composición necesitó 200 casos de prueba aleatorios para converger, pero la segunda lo hizo antes, con 100.

Los resultados de las pruebas en la composición de la metabúsqueda versión while pueden observarse en el cuadro 6.3 en la página 88 (usando correspondencia tipo división) y en el 6.4 (para correspondencia por aplanado). Para la versión forEach con división es el cuadro 6.5 y para esta con aplanado el 6.6 en la página 89. Se observa cómo los resultados de ambas composiciones son muy similares si se agrupan por técnica de correspondencia: las dos versiones convergen con unos 50-100 casos de prueba con división y sobre 500 para aplanado.

Cada columna de la comparativa muestra la información relativa a un conjunto de casos de prueba. En la parte superior se indica 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. Seguidamente se muestra el total de puntos de programa y la cantidad de ellos afectados por las diferencias anteriores. Por último, se muestran los tiempo totales, de usuario y de sistema (en los casos en que las cifras son altas, los tiempos se han redondeado al minuto).

La ejecución de Takuan se hizo con sus opciones por defecto para el préstamo y la compraventa. Estas inspeccionan todos los campos y propiedades de todas las variables que se usan en cada elemento <sequence> de la composición, y muestran el resultado del análisis sin aplicarle el simplificador lógico Simplify. En concreto, se registraron 120

Cuadro 6.1: *Evolución de la composición del préstamo bancario sequence.*

Número de casos de prueba	5	10	20	50	100	200
Total de invar.	106	164	160	163	167	167
Dif. en invar. interes.	116	45	18	4	0	-
Dif. en invar. (incl. no interes.)	124	62	38	14	10	-
Total puntos de prog. (TPP)	8	10	10	10	10	10
TPP con dif. en invar. interes.	10	7	7	4	0	-
TPP con dif. en invar. (incl. no interes.)	10	8	8	6	4	-
Tiempo total	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 6.2: *Evolución de la composición del mercado de compraventa.*

Número de casos de prueba	5	10	20	50	100
Total de invar.	8	8	6	6	6
Dif. en invar. interes.	6	2	0	0	-
Dif. en invar. (incl. no interes.)	14	14	8	8	-
Total puntos de prog. (TPP)	3	3	3	3	3
TPP con dif. en invar. interes.	2	2	0	0	-
TPP con dif. en invar. (incl. no interes.)	2	2	2	2	-
Tiempo total	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

elementos en el préstamo y 8 en la compraventa. Mientras que para las metabúsquedas se usaron todas las optimizaciones disponibles (pues el número de invariantes en caso contrario era muy alto).

6.1.3. Análisis de resultados

En la columna del conjunto de cinco casos de prueba del préstamo, se ve que estos generan un total de 106 invariantes. Sin embargo, en la tercera fila de datos se observa 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 la 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.

Cuadro 6.3: *Evolución de la composición de la metabúsqueda while con división.*

Número de casos de prueba	10	50	100	500	1000
Total de invar.	543	573	570	575	575
Dif. en invar. interes.	85	14	0	0	-
Dif. en invar. (incl. no interes.)	142	61	43	0	-
Total puntos de prog. (TPP)	42	42	42	42	42
TPP con dif. en invar. interes.	10	1	0	0	-
TPP con dif. en invar. (incl. no interes.)	19	16	15	0	-
Tiempo total	6m	12m	24m	149m	311m
Tiempo de CPU de usuario	2m	6m	12m	59m	140m
Tiempo de CPU de sistema	7s	15s	29s	2m29s	5m34s

Cuadro 6.4: *Evolución de la composición de la metabúsqueda while con aplanado.*

Número de casos de prueba	10	50	100	500	1000
Total de invar.	1106	1404	1730	1844	1844
Dif. en invar. interes.	2647	1285	488	0	-
Dif. en invar. (incl. no interes.)	2972	1480	631	12	-
Total puntos de prog. (TPP)	42	42	42	42	42
TPP con dif. en invar. interes.	17	14	13	0	-
TPP con dif. en invar. (incl. no interes.)	21	21	21	4	-
Tiempo total	12m	21m	40m	149m	293m
Tiempo de CPU de usuario	3m	9m	17m	70m	170m
Tiempo de CPU de sistema	9s	18s	34s	2m32s	4m45s

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. Las trazas de ejecución resultantes difícilmente podrán reflejar adecuadamente la complejidad interna de la composición (los distintos valores que hacen ejecutar las ramas de una instrucción condicional, etc.). Y, por lo tanto, se generan muchos invariantes falsos.

Sin embargo, a medida que el número de casos aumenta, la diferencia respecto al resultado del conjunto de referencia disminuye, y los resultados van convergiendo hacia un conjunto estable. Esto se cumple tanto para 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 casos más a los diez existentes en la prueba anterior), lo que parece debido a que presenta una complejidad interna menor. Por su parte, en la composición del préstamo esta convergencia da un fuerte incremento también con veinte casos de prueba,

Cuadro 6.5: *Evolución de la composición de la metabúsqueda forEach con división.*

Número de casos de prueba	10	50	100	500	1000
Total de invar.	487	509	509	510	510
Dif. en invar. interes.	61	14	0	0	-
Dif. en invar. (incl. no interes.)	127	60	42	0	-
Total puntos de prog. (TPP)	42	42	42	42	42
TPP con dif. en invar. interes.	9	1	0	0	-
TPP con dif. en invar. (incl. no interes.)	21	17	16	0	-
Tiempo total	6m	13m	22m	102m	222m
Tiempo de CPU de usuario	2m	5m	11m	53m	130m
Tiempo de CPU de sistema	6s	14s	27s	2m6s	5m9s

Cuadro 6.6: *Evolución de la composición de la metabúsqueda forEach con aplanado.*

Número de casos de prueba	10	50	100	500	1000
Total de invar.	1004	1323	1648	1769	1769
Dif. en invar. interes.	2624	1273	487	0	-
Dif. en invar. (incl. no interes.)	2905	1440	608	12	-
Total puntos de prog. (TPP)	42	42	42	42	
TPP con dif. en invar. interes.	14	12	12	0	-
TPP con dif. en invar. (incl. no interes.)	21	20	20	4	-
Tiempo total	10m	18m	34m	123m	259m
Tiempo de CPU de usuario	3m	8m	16m	63m	149m
Tiempo de CPU de sistema	9s	16s	32s	2m15s	5m55s

pero sigue aumentando paulatinamente hasta los cien. Lo indicado para las diferencias en invariantes es aplicable a los puntos del programa en los que se han observado.

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 retrasos en el envío de mensajes desde los socios 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.

En cuanto a las metabúsquedas, presentan un comportamiento muy similar si se observa según la correspondencia de variables usada. Parece lógico que el hecho de usar

una instrucción <while> o <forEach> para iterar no lleve a comportamientos demasiado diferentes en un mismo programa. Sin embargo, la técnica de correspondencia sí que influye significativamente. El mapeo por aplanado crea muchas menos variables, pero de estructura más compleja, lo que hace que se generen muchos más invariantes, que necesitan más casos de prueba para estabilizarse. Por lo demás, el comportamiento es similar a las demás composiciones: primero convergen los invariantes interesantes (que son un subconjunto) y después el global. También, el número de puntos de programa disminuye según lo hacen los invariantes.

Sobre los tiempos en las metabúsquedas, son bastante altos. Esto es debido principalmente a las limitaciones de la máquina en que se ejecutaron, pues se observa una gran diferencia entre la suma de los dos tiempos de CPU y el total (que está dedicado a intercambio y otras tareas del sistema operativo). A nivel comparativo también se observa como, en los conjuntos de casos más pequeños la máquina tiene recursos suficientes, y el crecimiento de los tiempos es inferior a los conjuntos mayores.

Conclusiones

A partir de los datos obtenidos se puede afirmar que, como suele suceder en las 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 ejecutarlos. En composiciones relativamente simples como el préstamo o la compraventa, los tiempos son pequeños. Pero como se apuntó en el capítulo 4, este crece significativamente en las composiciones de metabúsquedas (que son bastante más complejas). Este crecimiento está causado principalmente por el aumento del número de instrucciones y la cantidad de variables a inspeccionar. Por lo tanto, podría reducirse si el usuario indicara a Takuan que inspeccionara sólo aquellas variables que le interesan en cada punto de programa.

Además en ambos experimentos se observa un determinado umbral a partir del cual la mejora en la salida es marginal (o incluso nula), por lo que no merece la pena añadir más casos de prueba a partir de él. En la composición del mercado de compraventa dicho umbral se puede establecer en 20 casos de prueba, mientras que para el préstamo está entre 50 y 100, y para las metabúsquedas que usan división unos 50 o 100 y alrededor de 500 para aplanado.

Resumiendo, se puede 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, hay que tener en cuenta el coste temporal que tienen dichas pruebas, pues, además, existe un umbral de eficiencia a partir del cual los nuevos casos mejoran poco o nada la salida.

6.2. Cobertura de los casos de prueba

En el apartado 6.1 se ha demostrado que Takuan, al igual que la práctica totalidad de herramientas de prueba, suele funcionar mejor cuanto mayor sea el conjunto de casos de prueba. Sin embargo, en el capítulo 4 se vio que la ejecución de un número elevado de casos de prueba y el posterior análisis de sus trazas para generar invariantes puede ser costosa en tiempo.

Por ello, a continuación se estudia la relación entre los resultados obtenidos y la calidad de los conjuntos de casos de prueba [Pal09a], medida según su cobertura [Ram07, Zhu97].

6.2.1. Estructura del experimento

Este estudio sigue una aproximación basada en la llevada a cabo en [Ern01], comparando los resultados del conjunto de casos de prueba más completo con otros más limitados. La composición que se usa es una modificación del préstamo sequence denominada *préstamo ramas*, y cuya estructura se detalla en el apéndice A.1.2. La composición converge con casos de prueba aleatorios de manera similar a la composición del préstamo sequence. Sin embargo, como permite definir varios conjuntos de casos de prueba con distintas coberturas, se presta a realizar un estudio cualitativo.

Cobertura de los conjuntos de casos de prueba

Para las pruebas se crearon cinco conjuntos de quince casos de prueba cada uno. El primero de ellos es un conjunto de casos creados aleatoriamente según una distribución uniforme. Como la composición tiene dos condicionales con ramas vacías se crearon dos conjuntos que, ofreciendo cobertura de sentencias, dejaran sin ejecutar una rama vacía distinta cada uno (por lo que ninguno ofrecía cobertura de ramas). Después se creó un conjunto más de casos de prueba que ofrecía cobertura de sentencias y de ramas pero no de caminos. Y por último se generó un conjunto que ofrecía cobertura de caminos.

La razón de incluir conjuntos de 15 casos cuando con menos bastaba para ofrecer la cobertura deseada es que Daikon, por defecto, necesita que varias trazas verifiquen una propiedad para deducirla como invariante. De esta forma evita muchos invariantes que podrían surgir por limitaciones de los conjuntos de prueba. Por la misma razón, los casos que ofrecen determinadas coberturas se crearon cuidando que ejecutaran el mismo número de veces las combinaciones de ramas que cubren. Con 15 casos en la cobertura más exhaustiva (de caminos), cada camino se ejecuta 3 veces.

Métricas

Para medir la utilidad de cada conjunto de casos se tomaron los resultados del conjunto más completo (el que ofrece cobertura de caminos) como referencia con la que

comparar los demás. Por cada conjunto se anotaron el número total de invariantes y las diferencias entre su salida y la del conjunto de referencia.

Las métricas tomadas para comparar son las mismas que en 6.2 (así como la máquina empleada). La única diferencia es que no se muestran los tiempos empleados. Esto es debido a que como el número de casos de prueba de todos los conjuntos era el mismo (quince), no se produjeron diferencias significativas en sus tiempos de ejecución: tardaron todos entre 29 y 41 segundos en total.

6.2.2. Resultados

Los resultados de las distintas ejecuciones se pueden observar en el cuadro 6.7.

Cuadro 6.7: *Resultados de la composición del préstamo ramas en las pruebas.*

Cobertura	ALEAT.	INST. 1	INST. 2	RAM.	CAM.
Total de invar.	67	63	63	64	66
Dif. en invar. interes.	7	5	3	2	-
Dif. en invar. (incl. no interes.)	13	7	5	4	-
Total puntos de progr. (TPP)	7	7	7	7	7
TPP con dif. en invar. interes.	4	1	3	1	-
TPP con dif. en invar. (incl. no interes.)	4	2	4	2	-

En dicho cuadro, cada fila muestra una de las mismas métricas comentadas anteriormente para cada conjunto: total de invariantes, diferencias en los invariantes respecto a la salida del conjunto de referencia (primero solo los marcados como interesantes y después todos), total de puntos de programa donde se detectan invariantes y el número de ellos con diferencias (en primer lugar solo los interesantes y a continuación todos).

Las columnas indican los distintos conjuntos de casos de prueba usados: *Aleat.* es el conjunto aleatorio, *Inst. 1* es el que realiza cobertura de instrucciones sin pedir cantidades superiores a 10.000 euros, mientras que Inst. 2 también ofrece cobertura de instrucciones, pero ejercitando otras ramas (ningún préstamo tiene riesgo alto). Por último, *Ram.* es el conjunto que realiza cobertura de ramas, y *Cam.* de caminos.

6.2.3. Análisis de resultados

En la columna de los casos de prueba aleatorios (Aleat.) se ve que estos generan un total de 67 invariantes. Sin embargo, en las dos filas siguientes se observa que hay 7 diferencias en invariantes interesantes y 6 en no interesantes (13 en total) respecto a la cobertura de caminos. Y en las tres últimas filas se indica que dichas diferencias se encuentran en 4 de los 7 puntos del programa analizados.

En las dos siguientes columnas se ve que los dos conjuntos que realizan cobertura de instrucciones mejoran los resultados del conjunto aleatorio significativamente. Mientras

que el conjunto aleatorio tenía casi un 20 % de diferencias en invariantes respecto al total, estos aproximadamente las reducen a la mitad. También destaca que la diferencia entre el conjunto que ofrece la cobertura por ramas (Ram.) y la cobertura de caminos (Cam.) es muy pequeña: solo dos diferencias en invariantes interesantes.

En general se observa que, a medida que se usan conjuntos de casos de prueba más completos, mejoran los invariantes obtenidos (tanto los interesantes como los no interesantes). Esto parece coherente, pues las trazas de ejecución de los conjuntos más completos reflejan mejor la complejidad interna de la composición (los distintos valores que hacen ejecutar distintas ramas de una instrucción condicional, etc.), de modo que mejoran los invariantes generados con conjuntos más limitados. Además, si en una composición no es posible obtener cobertura de caminos (por tener bucles) la de ramas también proporciona resultados bastante buenos.

Conclusiones

En los resultados obtenidos, se observa una relación clara entre la calidad de un conjunto de casos de prueba (en el sentido de la cobertura que ofrece) y la de los invariantes que Takuan genera. De modo que, como era de esperar, usar conjuntos de casos de prueba que realicen coberturas lo más completas posible es una forma segura de mejorar y confirmar los invariantes generados por Takuan. Además, el uso de un conjunto u otro no repercute significativamente en su tiempo de ejecución.

Dado que el tiempo de ejecución de los distintos conjuntos de casos de prueba ha sido de similar magnitud, si se dispone de un método poco costoso para generar casos de prueba con determinadas coberturas, puede ser una ayuda interesante para la generación de invariantes.

A partir de dicha observación, se puede tomar la cobertura que ofrecen como indicativo de su fiabilidad. Incluso, ya se usen casos de prueba aleatorios o reales para obtener invariantes de una composición, se puede considerar añadir casos adicionales a dichos conjuntos para que cumplan criterios de cobertura más amplios y generen invariantes más fiables. Por ello, Takuan incorpora en su salida (en fase beta) un informe de la cobertura de instrucciones y ramas que consiguen los casos de prueba de una ejecución. Además, también indica las instrucciones y ramas concretas no ejecutadas, lo que puede ayudar a encontrar casos que mejoren la cobertura del conjunto.

Queda como trabajo futuro perfilar más características del conjunto de casos de prueba que afecten a la calidad de los invariantes obtenidos. Se podrían realizar estudios sobre la cobertura del flujo de datos, ejecuciones de bucles, etc. Para ello, un elemento clave es disponer de un conjunto con un número significativo de composiciones variadas sobre el que realizar los experimentos.

CAPÍTULO 7

Conclusiones y trabajos futuros

En este capítulo se recopilan las conclusiones del trabajo desarrollado a lo largo de la tesis, y se presentan las líneas de trabajo futuro que se obtienen de él.

7.1. Resumen de los resultados

En los últimos años, las arquitecturas orientadas a servicios están cambiando la filosofía de desarrollo de software en muchos entornos. El uso de servicios web facilita significativamente la interoperabilidad entre sistemas, permitiendo programar un sistema de gran tamaño usando otros más simples de manera sencilla. El principal lenguaje para componer servicios es WS-BPEL 2.0, que ha sido estandarizado por OASIS con la participación de las grandes empresas del sector informático. WS-BPEL incorpora instrucciones presentes en otros lenguajes de programación, como asignaciones o bucles, pero también instrucciones para la invocación de SW, compensaciones de errores en SW, etc. Esto hace que sea necesario adaptar las técnicas clásicas de prueba de software a este lenguaje.

El estudio del estado del arte de la prueba de software en WS-BPEL permite afirmar que existen pocas herramientas automáticas, es decir, que operen directamente a partir de los ficheros de definición de la composición sin ser necesaria intervención humana y que se basen en la ejecución real de código WS-BPEL. Es más, algunas de ellas solo funcionan con BPEL4WS 1.1. Aunque los resultados son prácticamente tan válidos como los logrados en WS-BPEL 2.0, sería necesario actualizarlas para dar soporte al estándar y asegurar la compatibilidad. También es destacable que si no se distribuyen con licencias libres ni ofrecen interfaces claras para su uso masivo de manera programable, difícilmente podrán integrarse con otros sistemas, limitando su adopción. Por último, a medio-largo plazo parece necesaria la definición de metodologías de desarrollo específicas para WS-BPEL que proporcionen soporte completo, incluyendo la prueba.

La generación dinámica de invariantes es una técnica dinámica (basada en la ejecución de código) que se ha utilizado con éxito en la prueba y mejora de programas escritos en lenguajes imperativos. El uso de invariantes obtenidos a partir de conjuntos de casos de prueba fiables incluyen la verificación de programas, depuración de errores, detección de errores al modificar un programa y su documentación. Adicionalmente, también permite detectar aspectos a mejorar en un conjunto de casos de prueba comparando los invariantes que se generen dinámicamente a partir de él con los indicados en la especificación del programa.

Esta tesis se ha planteado como objetivo el estudio de la validez de la generación dinámica de invariantes (también conocida como generación de invariantes potenciales) para la prueba de composiciones de servicios web en WS-BPEL. Para ello, en primer lugar, se comprobó la viabilidad de la generación dinámica de invariantes para WS-BPEL, diseñando una arquitectura basada en el generador dinámico de invariantes Daikon, que ha demostrado su utilidad para varios lenguajes imperativos. Daikon es software libre, y se ha integrado con código propio y otros dos sistemas libres: el motor de ejecución compatible WS-BPEL 2.0 ActiveBPEL y la biblioteca de prueba unitaria para WS-BPEL BPELUnit, que incorpora un mecanismo de simulación de servicios web que puede ser de utilidad para composiciones cuyos servicios socios no estén disponibles o para probarla bajo determinadas situaciones. El sistema resultante se ha denominado Takuan, y es el único generador dinámico de invariantes para WS-BPEL hasta la fecha.

Takuan recibe en su entrada los ficheros que definen una composición WS-BPEL (que pueden hacer uso de extensiones para indicar las variables a inspeccionar y los puntos de programa a instrumentalizar), un conjunto de casos de prueba en formato BPELUnit y sus opciones de funcionamiento. A partir de dicha entrada, Takuan genera una lista de invariantes que se cumplen en los puntos de programa indicados. Para ello, implementa dos correspondencias entre datos XML Schema con estructura arbórea y los vectores unidimensionales que Daikon maneja (división y aplanado). Estas correspondencias han demostrado ser complementarias, permitiendo obtener distintos invariantes según se desee con un rendimiento distinto cada una.

Tras implementar Takuan se realizaron pruebas para comprobar la utilidad de los invariantes que genera, pruebas que mostraron su capacidad para encontrar limitaciones en un conjunto de casos de prueba y para resaltar fallos en una composición. Sin embargo, en ellas se observaron dos mejoras específicas de WS-BPEL que permitiría mejorar el rendimiento de Takuan: la eliminación de invariantes incluidos en las restricciones XML Schema de los datos usados y el uso de información de comparabilidad para reducir la cantidad de invariantes no útiles que genera (incluyendo el filtrado de variables no usadas). Tras implementarlas y evaluarlas, se obtuvo una mejora sustancial tanto en el tiempo de ejecución de Takuan como en la cantidad de invariantes que produce, eliminando muchos invariantes no informativos y redundantes. Posteriormente, se demostró la estabilidad de los invariantes generados, permitiendo asegurar su correcto funcionamiento con un conjunto soporte adecuado. Los resultados obtenidos permiten afirmar la validez de la generación dinámica de invariantes para apoyar la prueba de composiciones WS-BPEL.

7.2. Trabajos futuros

A continuación se listan los trabajos futuros que la tesis ofrece:

- Como se comentó en el apartado 4.3.5, se han implementado dos técnicas de correspondencia de árboles XML a vectores unidimensionales que Daikon puede manejar: aplanado y división. Ambas son complementarias, presentando cada una de ellas ventajas e inconvenientes. Sin embargo, podrían aplicarse distintas técnicas a cada dimensión de cada matriz. Por ejemplo, a una variable se le podría aplicar división para la primera dimensión y aplanado para la segunda, mientras que otra podría dividirse usando únicamente división. También se podría implementar una selección manual de correspondencia más fina usando algún lenguaje de selección de información de árboles XML (como XPath). De este modo, por ejemplo, se podría seleccionar las cinco primeras apariciones de un campo en una variable XML Schema y aquellos campos que empezaran por una letra concreta para hacer una variable Daikon. Esta correspondencia permitiría un estudio mucho más fino, aunque requeriría intervención humana y probablemente aumentaría el tiempo de ejecución de Takuan.

- Sobre las optimizaciones implementadas comentadas en el apartado 5.3.1, el filtrado de variables no usadas tiene una desventaja: puede producir cierta pérdida de información. Esto se puede dar, sobre todo, en el caso de que dos o más actividades colaboren produciendo un invariante de alto nivel que las relacione y que pudiera ser de interés para el usuario. Por ejemplo, si solo se inspeccionaran variables locales dentro de un bucle, no se generaría sus pre-condiciones y post-condiciones. La solución pasaría por habilitar diversos tipos de instrumentalización de puntos de programa e inspección de variables. La implementación de esta opción no parece excesivamente difícil, pero para su evaluación sería necesario disponer de un número representativo de composiciones WS-BPEL reales con asertos.

- Los experimentos realizados sobre el tamaño del conjunto de casos de prueba usado en la entrada de Takuan y su cobertura han permitido asegurar la estabilidad del sistema. Sin embargo, sería deseable realizar un estudio sobre los elementos de la composición que inciden en los conjuntos necesarios para estabilizar los invariantes: bucles, estructuras paralelas, etc. Para ello sería interesante tener un conjunto con un número significativo de composiciones WS-BPEL reales con que hacer el estudio.

- Existen diversas tecnologías incipientes que están usando WS-BPEL como base. Entre ellas destaca BPMN (*Business Process Modeling Notation*, una notación gráfica estandarizada para procesos de negocio), que tiene una amplia adopción y ha respaldado a WS-BPEL implementando transformación automática de modelos gráficos BPMN a código WS-BPEL ejecutable [Ini10]. Igualmente también han

aparecido recientemente otras como BPEL4People (que incluye actividades humanas en WS-BPEL) y BPELScript (que facilita la creación de composiciones con un lenguaje de *scripting*). Sería interesante, una vez las tecnologías tengan cierta aceptación en el mundo de las AOS, estudiar la adaptación de Takuan para operar con ellas.

- Sería deseable realizar un estudio cualitativo de invariantes que genera Takuan, que permitiría saber qué tipo de invariantes detectan más fallos en WS-BPEL, si sería deseable modificar el conjunto de invariantes que Daikon comprueba por defecto (pues está pensado para lenguajes tradicionales), etc. El principal obstáculo es que sería necesario un conjunto significativo de composiciones con asertos en su código, algo que actualmente no se ha encontrado en la bibliografía ni en la red.
- Por último, se podría comparar la capacidad de detectar fallos de Takuan con la de otras herramientas para la prueba de WS-BPEL, como podría ser la prueba mutación con GAmera.

APÉNDICE **A**

Composiciones WS-BPEL

En este apéndice se comentan las distintas composiciones WS-BPEL usadas en el documento. Se puede conseguir información más detallada de cada una de ellas, así como su código fuente en [Gru10].

A.1. Préstamo bancario

Esta composición recibe peticiones de préstamos bancarios por parte de clientes. Está detallada en el estándar WS-BPEL [OAS07b]. Cada petición incluye la cantidad que se solicita y cierta información personal del cliente. La composición da como respuesta si el préstamo se aprueba o rechaza.

Para ello, se basa en la cantidad solicitada, el riesgo que un servicio asesor externo le asigna al cliente y la valoración de la operación por parte de un servicio aprobador. En concreto, en el caso de que la cantidad sea inferior a 10.000 euros se consulta el riesgo y, si es bajo, el préstamo se aprueba. Pero si falla alguna de estas dos condiciones, la composición invoca al servicio externo de aprobación, cuya respuesta será la que se le dé al cliente.

En ambas versiones 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*, *approvalInput* y *assessorInput*, y la variable *approval* por *processOutput* y *approvalOutput*. Igualmente, se ha renombrado la variable *risk* como *assessorOutput* para homogeneizar los nombres. En el cuadro A.1 en la página siguiente se resumen los cambios.

A.1.1. Versión sequence

La versión *sequence* de la composición del préstamo bancario usa actividades en vez de secuencias (como se hace en el ejemplo del estándar) para implementar su lógica

Cuadro A.1: *Cambios de variables en la composición del préstamo bancario sequence.*

Variables originales	Variables nuevas
loanInfo	processInput, approvalInput, assessorInput
risk	assessorOutput
approval	processOutput, approvalOutput

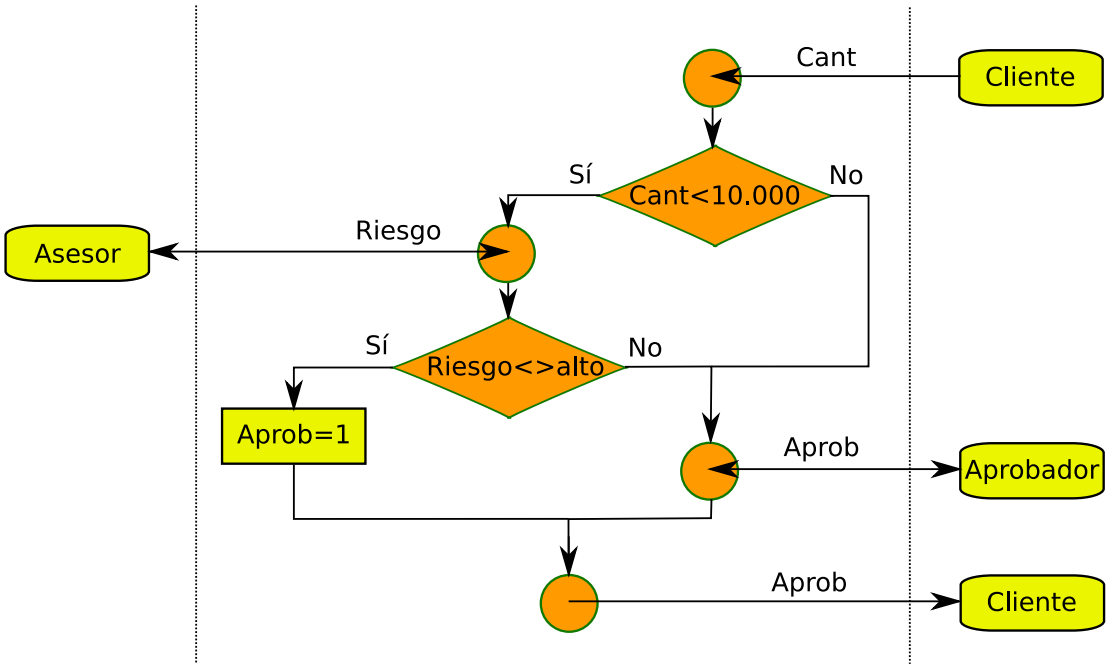


Figura A.1: *Esquema de la composición préstamo sequence.*

interna y facilitar la generación de invariantes. En la figura A.1 se observa un esquema simplificado del flujo de ejecución de la composición.

A.1.2. Versión ramas

Esta versión de la composición del préstamo bancario es una modificación de la versión del estándar que permite definir conjuntos de casos de prueba con diversas coberturas del código. En concreto, se pueden encontrar conjuntos que ofrecen cobertura de sentencias dejando sin ejecutar alguna de las dos ramas condicionales vacías que tiene la composición, conjuntos que proporcionan cobertura de sentencias y de ramas pero no de caminos, y otros con cobertura de caminos. En la figura A.2 en la página siguiente se observa de manera esquemática.

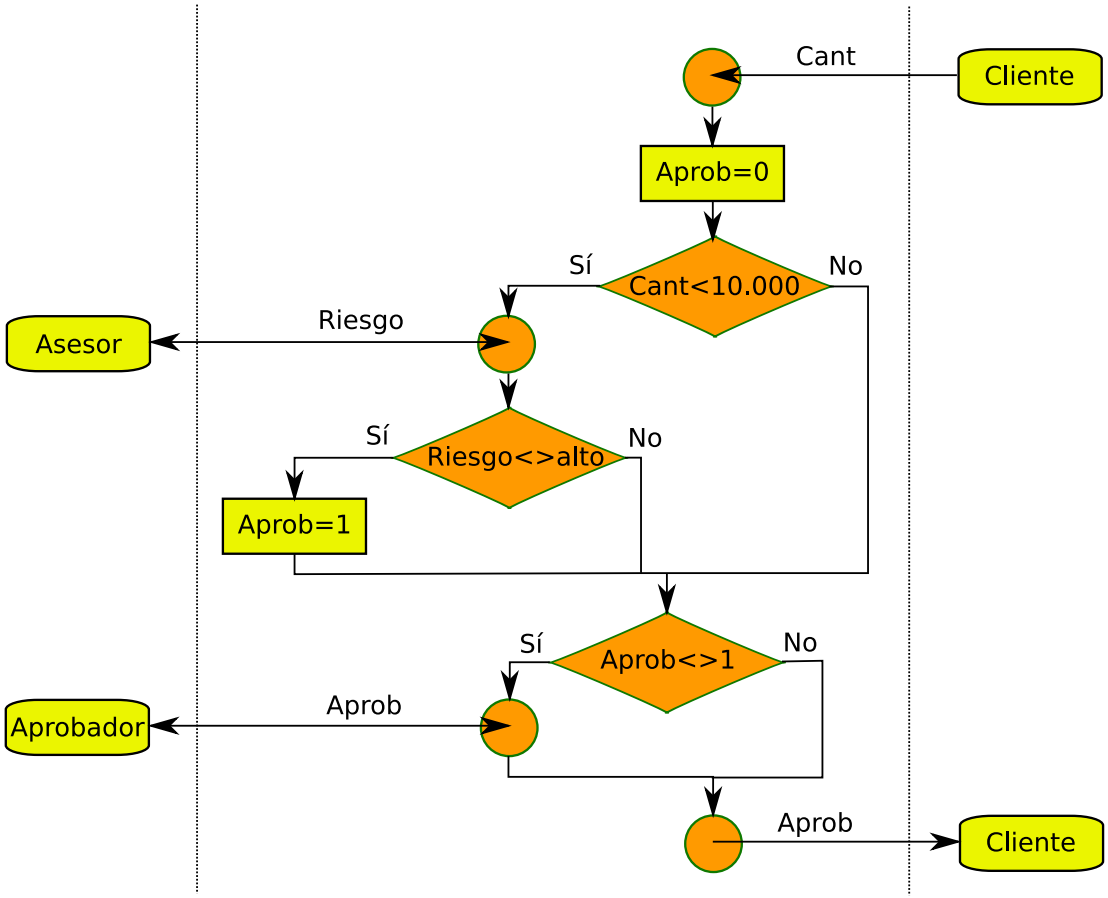


Figura A.2: *Esquema de la composición préstamo ramas.*

Para ello, la composición comprueba en primer lugar si la cantidad solicitada es inferior a 10.000 euros. Si es así llama al asesor, y si el riesgo que este estima no es alto se marca el préstamo como concedido y se envía al cliente. En caso contrario, si el préstamo no se ha concedido (porque la cantidad fuera mayor que 10.000 o el riesgo alto), se invoca al aprobador y se envía su respuesta como resultado de la composición.

A.2. Metabúsqueda

Esta composición implementa un motor de *metabúsqueda* de webs en Internet detallado en [May06a]. Para ello, hace uso de dos servicios simples de búsqueda (uno de Google y otro de MSN). Invoca a ambos en paralelo y ofrece al usuario sus resultados intercalados empezando por el primer resultado de Google si lo hubiera. Internamente hace uso de más de 70 instrucciones, incluyendo bucles, concurrencia y variables no escalares.

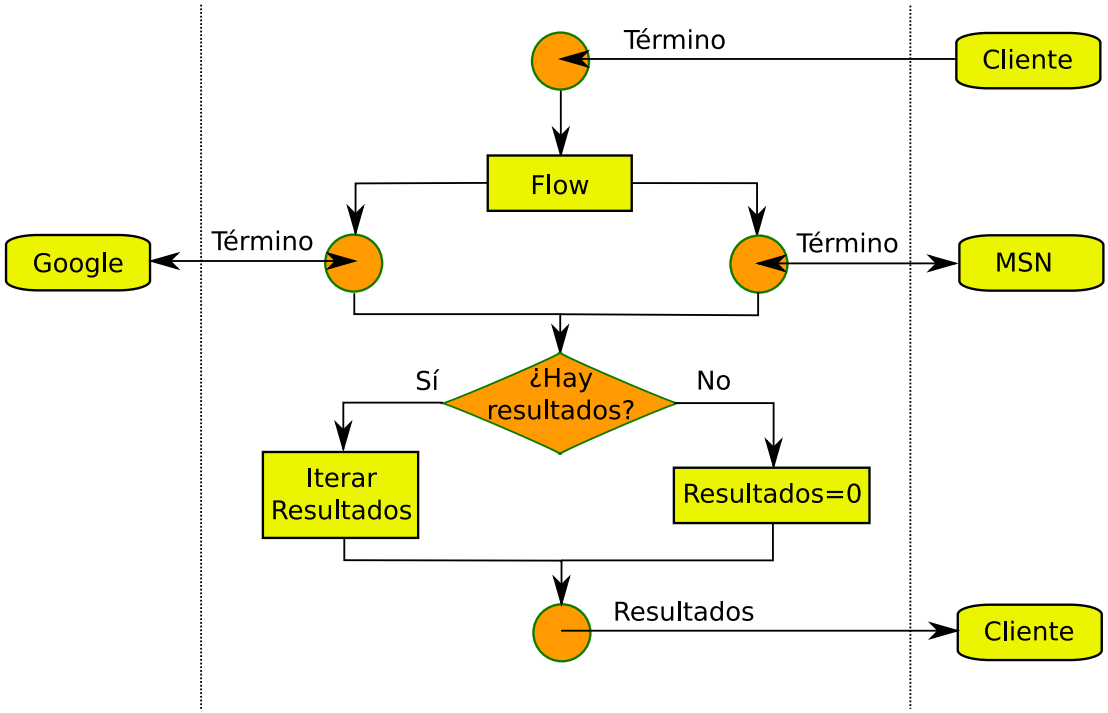


Figura A.3: *Esquema simplificado de la composición de la metabúsqueda.*

De esta composición existen dos variantes: versión *while* y *forEach*. Ambas son similares, siguiendo el esquema de la figura A.3 (que es una adaptación de la figura 94 de [May06a]). La única diferencia es la instrucción usada para recorrer los resultados de cada servicio.

A.2.1. Versión while

Esta versión hace uso de un bucle *while* (mientras) para iterar los resultados de los dos servicios y construir la variable resultado que envía al cliente.

A.2.2. Versión forEach

Esta versión recorre los resultados de los servicios usando una instrucción *forEach* (que hace una iteración por cada elemento de una variable XML Schema que incluya repeticiones), construyendo así la variable resultado que se envía al cliente.

A.3. Mercado de compraventa

Esta composición se ofrece como ejemplo en la web de ActiveVOS [Act08], e implementa un mercado de compraventa de productos (en inglés *MarketPlace*). Básicamente

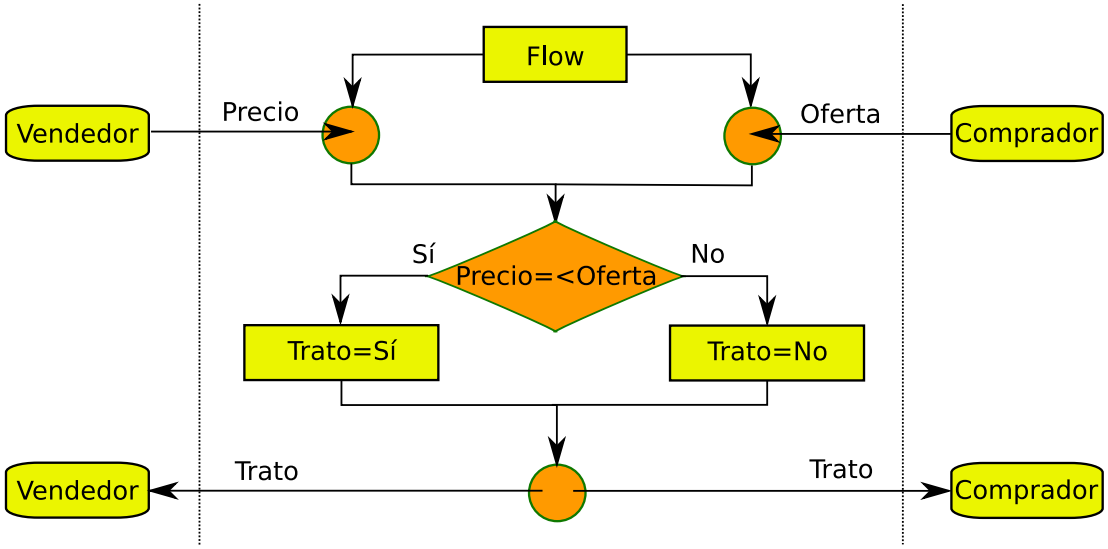


Figura A.4: *Esquema de la composición mercado de la compraventa.*

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 el precio pedido para la venta, se informa que se puede realizar la transacción, y en otro caso indica que no se ha llegado a un acuerdo. La figura A.4 muestra un esquema simplificado de su flujo de ejecución.

APÉNDICE **B**

Instalación y uso de Takuan

En este apéndice se comentan las diversas formas de instalar y usar Takuan. La información relativa a la extensión para NetBeans está adaptada, en su mayor parte de [Á10], mientras que la de IdigInBPEL proviene de [San10]. Ambos textos están liberados bajo licencia compatible con la de esta tesis, y se cuenta con el visto bueno de sus autores para su uso en ella.

Takuan es software libre, y su código fuente se puede descargar gratuitamente de su web oficial [Grub]. En dicha web también se proporciona información de las publicaciones relacionadas con Takuan, un área de soporte al usuario y otros recursos para facilitar su explotación.

B.1. Instalación

La instalación de Takuan puede realizarse compilando, instalando e integrando cada uno de los componentes que lo forman por separado. Sin embargo, se recomienda hacerlo usando el guión (del inglés *script*) de instalación proporcionado en su web oficial, o descargando una imagen de máquina virtual con el sistema ya instalado de dicha web. Nótese que si desea usar IdigInBPEL, el guión de instalación de este se encarga también de la instalación de Takuan, por lo que puede saltarse este apartado e ir directamente a la sección B.2.3.

Cuando se instala, Takuan está accesible mediante un *servlet* (programa ligero que se ejecuta en un servidor) en *Apache Tomcat* (en concreto, en la URI `http://localhost:8180/takuan`), que puede ser invocado desde otra máquina si se desea y que puede servir para probar su funcionamiento.

B.1.1. Instalación mediante guión

La instalación mediante guión es la forma recomendada de instalación, pues asegura que se dispondrá de la última versión liberada y también permite la actualización del sistema fácilmente. Aunque el código propio de Takuan no sufre modificaciones con demasiada frecuencia, sí suele hacerlo el de algunos de los sistemas libres que han sido modificados e integrados en él: la biblioteca de prueba unitaria BPELUnit, el motor WS-BPEL 2.0 ActiveBPEL y el generador de invariantes Daikon.

Dicho guión se ha usado para instalar Takuan en varias distribuciones GNU/Linux basadas en Debian y openSUSE publicadas entre los años 2008 y 2010, aunque probablemente funcione con la mayoría de distribuciones de dichas fechas o más recientes. Para descargar el guión de la web oficial de Takuan y ejecutarlo se pueden usar las siguientes órdenes:

```
$ wget http://neptuno.uca.es/redmine/repositories/entry/sources-fm/\
trunk/scripts/install.sh?format=raw
$ bash install.sh takuan
```

Durante el proceso de instalación el guión informará de cualquier problema en las dependencias que Takuan necesita, y solicitará la clave del usuario administrador (*root*) del sistema cuando sea necesario instalar el software que descargue. En caso de que el guión detecte componentes obsoletos preguntará si se desean actualizar (se debe responder con *y* en caso afirmativo). Al finalizar, pedirá que reinicie la sesión actual para que el entorno incluya las modificaciones realizadas en el fichero `.profile` del directorio de entrada del usuario.

En las distribuciones Ubuntu 9.04 y 8.10 es necesario instalar el *Sun Java 6 JDK* antes de ejecutar el guión. Puede hacerse mediante las siguientes órdenes:

```
$ sudo aptitude install sun-java6-jdk
$ sudo update-java-alternatives -s java-6-sun
```

B.1.2. Instalación mediante imagen de máquina virtual

La instalación de Takuan mediante una imagen de máquina virtual se recomienda para el caso de que no se tengan permisos de administrador en un sistema GNU/Linux o se prefiera usar desde otro sistema operativo.

La máquina virtual es una distribución Ubuntu Server GNU/Linux con Takuan instalado y configurado para su uso inmediato. La imagen se proporciona en el formato estandarizado independiente del sistema de virtualización *Open Virtualization Format* (OVF) 1.0. No obstante, hay que señalar que esta imagen no se actualiza siempre que se producen modificaciones en algún componente de Takuan. Por lo que si se necesita alguna opción reciente se debería confirmar si la versión que incluye la máquina la incorpora.

Para su instalación hay que descargar la imagen de la web oficial de Takuan:

```
$ wget http://neptuno.uca.es/files/virtual/takuan-vm.zip
```

Tras descomprimirla, es necesario instalarla en el sistema de virtualización que se desee usar. En el caso del sistema libre VirtualBox, los pasos son los siguientes:

1. Importe la descripción incluida en la imagen OVF (`File` → `Import virtualized service`).
2. Active el soporte para PAE (botón derecho sobre la máquina virtual `Configuration` → `System` → `Processor` → `Enable PAE/NX`).
3. Arranque la máquina virtual. Le aparecerá una *shell* con el usuario `takuan` (clave `takuan`). Si fuera necesario tener permiso de administrador del sistema, el usuario `admin` (clave `admin`) tiene capacidad para ejecutar la orden `sudo`.

B.2. Uso de Takuan

Takuan se puede usar de tres formas: desde consola, a través de una extensión para el entorno de desarrollo NetBeans y mediante la aplicación gráfica de escritorio en fase *beta* IdigInBPEL.

B.2.1. Uso desde consola

El uso desde consola es el menos amigable, pues necesita conocer la estructura de directorios y ficheros de Takuan. Sin embargo, es el que permite más potencia, sobre todo a la hora de programar pruebas masivas.

Para empezar, asegúrese de que el motor ActiveBPEL está activado. Para ello, escriba *ActiveBPEL.sh start*. Si fuera necesario pararlo, puede hacerlo con *ActiveBPEL.sh stop*. Y en caso de desear el modo de depuración remota JDB use *ActiveBPEL.sh start -debug*.

Configuración de la ejecución de Takuan

En la instalación de Takuan se incluyen varios directorios con composiciones de ejemplo en `/home/takuan/takuan/samples`. Cada uno de ellos contiene todos los ficheros para desplegar una composición WS-BPEL. Entre otros, cada directorio incluye un fichero en formato XML denominado `build.xml`, que tiene las opciones de funcionamiento de Takuan (en el listado B.1 se observa un ejemplo de su contenido). Los parámetros de configuración que soporta son:

- *bprfile* ruta del fichero BPR. En el ejemplo se observa en la línea 1.
- *bptsfile* ruta del fichero BPTS con la especificación del conjunto de casos de prueba. En la línea 2 del código de ejemplo se puede ver.

- *main.bpel* ruta del fichero BPEL con la definición de la composición, línea 3 del ejemplo.
- *analyzer.flags* opciones del analizador (separadas por espacios). Las opciones que acepta Takuan son:
  - Correspondencia XML. Por defecto es "matrix flattening" (aplanado). Para usar "matrix slicing" (división) debe indicarse *--index-flattening*.
  - Las optimizaciones de comparabilidad están activadas por defecto. Para desactivarlas, hay que indicar *--disable-comparability*.
  - Por defecto, las variables que no se usan en un punto del programa se descartan. Para cambiar este comportamiento se debe indicar *--disable-filter-unused*.
  - Si no se indica lo contrario, la salida de Daikon no se procesa por el simplificador Simplify. Para pasarla use *--simplify*.
  - Para cambiar el guión de Daikon que se usa, puede indicarse con *--daikon-script=<guión>*.
  - Por defecto, Takuan almacena su salida en el directorio de trabajo. Si se prefiere que lo haga en otro se puede cambiar con *--output-dir=<directorio>*.
  - Si desea generar métricas y estadísticas del proceso, está el parámetro *--metrics*.
  - Para usar varias líneas de ejecución en las operaciones en las que sea posible, es necesario indicar su número con *--ncpu=<número>*.

Nótese que el uso de correspondencia por división en una composición cuyas variables son escalares (no árboles XML Schema) producirá, en el proceso de correspondencia, un vector unidimensional de un solo elemento (el valor escalar) por cada variable. Y, por lo tanto, Takuan inferirá, además de los invariantes que produciría usando correspondencia por aplanado, otros que no aportarán información, del tipo $size(var1) = 1$, $size(var1) = size(var2)$, etc. Por lo tanto, no se recomienda usarlo en dicho caso.

Listado B.1: *Fichero `build.xml` con parámetros de funcionamiento de Takuan.*

```
1 <property name="bprfile" value="fichero.bpr"/>
2 <property name="bptsfile" value="fichero.bpts"/>
3 <property name="main.bpel" value="fichero.bpel"/>
4 <property name="analyzer.flags" value="opciones"/>
```

La versión de Daikon que usa Takuan elimina automáticamente las restricciones incluidas en el XML Schema. Este comportamiento no se puede deshabilitar sin modificar su código fuente.

Adaptación de la composición WS-BPEL

Aunque Takuan puede trabajar sobre una composición WS-BPEL sin modificaciones, existen extensiones XML propias que permiten configurar su procesado. En concre-

to, especificando qué puntos de programa se quieren instrumentalizar y qué variables inspeccionar.

Para hacer uso de estas extensiones, primero es necesario definir el espacio de nombres de Takuan en el documento. Además, si se quiere que ActiveBPEL pueda ejecutar directamente la composición modificada, se debe añadir un bloque que indique al motor de ejecución que ignore las extensiones propias de XPath que usa. Esto se hace con el código de las líneas 4 a 7 del listado B.2.

Listado B.2: *Declaración del espacio de nombres de Takuan.*

```
1 <bpel:process
2   xmlns:bpel="http://docs.oasis−open.org/wsbpel/2.0/process/executable"
3   xmlns:uca="http://www.uca.es/xpath/2007/11">
4    <bpel:extensions>
5       <bpel:extension mustUnderstand="no"
6         namespace="http://www.uca.es/xpath/2007/11"/>
7    </bpel:extensions>
8 </bpel:process>
```

En lo referente a las variables, hay tres parámetros generales personalizables. Todos ellos son atributos del elemento raíz <process> de la composición WS-BPEL:

uca:instrumentVariablesByDefault acepta los valores "yes" y "no". Especifica si se inspeccionarán todas las variables por defecto o no.

uca:instrumentSequencesByDefault acepta los valores "yes" y "no". Indica si se instrumentalizan todas las secuencias por defecto, o solo las marcadas.

uca:maxInstrumentationDepth acepta valores enteros mayores o iguales a 0. Controla la profundidad máxima del árbol XML de instrucciones WS-BPEL que se instrumentalizará (0 significa sin límite).

En el listado B.3 se puede ver un ejemplo que instrumentaliza todas las variables por defecto.

Listado B.3: *Parámetros generales de registro de variables en Takuan.*

```
1 <process name="..." targetNamespace="..." xmlns="..."   ...
2   uca:instrumentVariablesByDefault="yes">
```

Además, cada variable puede incluir el atributo *uca:inspeccionar*, que especifica si debe o no inspeccionarse. El valor "yes" indica que sí se inspeccionará, y "no" que no se hará. En caso de que no se defina, se usará el comportamiento configurado por el atributo *uca:instrumentVariablesByDefault* del elemento raíz. En el listado B.4 se muestra un ejemplo con los tres posibles valores:

Listado B.4: *Valores de atributos de Takuan para variables.*

```
1 <variables>
2   <variable name="approverInput" [...]  uca:inspeccionar="yes"/>
```

```
3    <variable name="approverOutput" [...] uca:inspeccionar="no"/>
4    <variable name="assessorOutput" messageType="ns2:AssessorOperationReply"/>
5  </variables>
```

Por último, en cada punto de programa se puede especificar si el <sequence> se quiere instrumentalizar o no. Esto se realiza con el atributo *uca:instrument*, cuyo uso es similar al equivalente para variables: "yes" para instrumentalizar la secuencia (incluso cuando su profundidad sea mayor que el límite configurado por defecto), "no" para no instrumentalizarla, y si no se indica nada se considerará el comportamiento configurado con el atributo *uca:instrumentSequencesByDefault* del elemento raíz. Nótese que si el valor por defecto es "yes", pero la profundidad de anidamiento del <sequence> es mayor que el límite especificado por *uca:maxInstrumentationDepth*, no se instrumentalizará. En el listado B.5 se muestra un ejemplo de un <sequence> que se indica explícitamente que no se instrumentalice.

Listado B.5: *Atributo de instrumentalización en instrucción WS-BPEL.*

```
1  <sequence name="..." uca:instrument="no">
```

Casos de prueba

El formato de los conjuntos de casos de prueba que Takuan acepta es el formato BPTS, definido por BPELUnit. A continuación se comentará brevemente su estructura. Para conseguir información más completa y actualizada puede consultarse [Eng10].

Los ficheros con extensión *bpts* son ficheros XML. Como todos los documentos XML, incluye un elemento raíz donde se especifican los espacios de nombres. Nótese que el prefijo *tes* está asociado al espacio de nombres de BPELUnit.

Tras un bloque de información con información del proceso que se desplegará (nombre, URL, etc.) se define el conjunto de casos de prueba con la etiqueta <tes:testCases>, que define cada caso de prueba con un <tes:testCase>.

En el ejemplo que muestra el listado B.6 se observa que el cliente invocará al servicio mediante una instrucción <send> que esperará una respuesta síncrona con <receive>. Después se indican los datos que esta primera envía (en este caso el artículo que se desea vender y su precio mínimo de venta). Tras ella, se observa la <receive> vacía, aunque podría contener alguna condición que se quisiera comprobar referente a la respuesta de la composición. Posteriormente se define el comportamiento del otro servicio socio (*buyer*) y termina el caso de prueba.

Listado B.6: *Extracto de fichero de casos de prueba* `.bpts`.

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  < tes:testSuite  xmlns:trs=" ... "  ...  >
4  </ tes:testSuite >
5
6  <tes:name>MarketPlace</tes:name>
7  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
```

```
8  <tes:deployment> ... </tes:deployment>
9
10 <tes:testCases>
11  <tes:testCase  name="case">
12   <tes:clientTrack>
13    <tes:sendReceive service="tns:marketplaceSeller"
14                          port="seller"  operation="submit">
15
16     <tes:send>
17      <tes:data>
18       <tns:inventoryItem>Takuan</tns:inventoryItem>
19       <tns:askingPrice>231</tns:askingPrice>
20      </tes:data>
21     </tes:send>
22
23     <tes:receive  fault="false">
24     </tes:receive>
25    </tes:sendReceive>
26   </tes:clientTrack>
27
28 <tes:partnerTrack name="Buyer">
29   ...
30 </tes:partnerTrack>
31
32 </tes:testCase>
```

La estructura de este fichero puede ser mucho más rica, permitiendo indicar retrasos en el envío de mensajes, condiciones a comprobar sobre los mensajes de los socios, etc. Dicha información puede consultarse en [Eng10].

Ejecución de Takuan y resultados

Para facilitar la ejecución de Takuan, se dispone de un guión (escrito en *Apache ant*) que se encarga de llamar a los distintos componentes de manera secuencial. De modo que al ejecutar *ant* en el directorio donde está una composición se lanzará Takuan, y se podrá ver en la consola los mensajes que genera la ejecución de sus distintos componentes. Cuando termine, se obtendrán los resultados en un subdirectorio denominado `daikon-out-(fecha)-(hora)`. En él se encontrará el fichero `build.xml` con las opciones de la ejecución y dos directorios: `process-logs` y `results`.

El primero almacena los registros de ejecución que genera cada caso de prueba en ActiveBPEL, mientras que `results` tiene los ficheros con las trazas resultantes de filtrar los registros de ejecución, y el resto de ficheros que genera y usa Takuan. Los invariantes están en formato plano en el fichero `procesoInspeccionado.out`, y en formato interno de Daikon en `procesoInspeccionado.inv.gz`.

En el listado B.7 se observa un ejemplo de invariantes obtenidos en la composición del mercado de compraventa. Los puntos de programa se indican con *:::*, y los inva-

riantes están en una lógica definida por Daikon, por lo que se recomienda consultar su documentación y usar sus herramientas para trabajar con ellos [Dai10].

Listado B.7: *Invariantes obtenidos en el fichero .out.*

```
1  Daikon version 4.3.1, released August 2, 2007; http://pag.csail.mit.edu/daikon.
2  Reading declaration files . (read 1 decls file)
3  Processing trace data; reading 20 dtrace files:
4
5  Invoking Simplify to identify redundant invariants .....0,4 s
6  ==============================================================================
7  marketplace._process1_MarketplaceSequence_MarketplaceSwitch_else1_sequence1:::EXIT
8  negotiationOutcome.outcome == "Deal␣Failed"
9  ==============================================================================
10 marketplace._process1_MarketplaceSequence_MarketplaceSwitch_if−condition1_sequence1:::ENTER
11 buyerInfo.offer > sellerInfo.askingPrice
12 ==============================================================================
13 marketplace._process1_MarketplaceSequence_MarketplaceSwitch_if−condition1_sequence1:::EXIT
14 buyerInfo.offer == orig(buyerInfo.offer)
15 sellerInfo.askingPrice == orig(sellerInfo.askingPrice)
16 negotiationOutcome.outcome == "Deal␣Successful"
17 buyerInfo.offer > sellerInfo.askingPrice
18 Exiting Daikon.
```

Los ficheros generados por Takuan con información relevante a la cobertura de los casos de prueba son los siguientes (nótese que la generación de estos ficheros está en fase de desarrollo no estable):

coverage.log contiene información sobre la cobertura de sentencias y de ramas. Puede verse un ejemplo de la composición del préstamo versión sequence en el listado B.8, que tiene una cabecera, el listado de sentencias ejecutadas (indicando con una fracción el número de casos que la han ejecutado sobre el total) y las sentencias que ningún caso ha ejecutado.

Posteriormente, en el mismo fichero, se muestra la cobertura de ramas. Tiene un formato similar al primer bloque: encabezado, lista de ramas ejecutadas indicando cantidad de casos que lo han hecho y ramas no ejecutadas.

Listado B.8: *Fichero de cobertura* `coverage.log`*.*

```
1  Instruction coverage
2  ====================
3
4  Executed 45/50 (90%)
5  −−−−−−−−−−−−−−−−−−−−−−
6  (1/1) /process/sequence/if/else/sequence/while/sequence/assign/copy[0]
7  (1/1) /process/sequence/assign[3]
8  (1/1) /process/sequence/reply[@name='Response']
9  (1/1) /process/sequence
10 Never executed 5/50 (10%)
11 −−−−−−−−−−−−−−−−−−−−−−−−
12 /process/sequence/if/if−condition/sequence/assign[2]
```

```
13 /process/sequence/if/if-condition/sequence
14 /process/sequence/if/if-condition/sequence/assign[3]
15
16 Branches coverage
17 ==================
18 Branches executed 1 / 2 (50 %)
19 ------------------------------
20 /process/sequence/if/else
21
22 Branches not executed 1 / 2 (50 %)
23 ----------------------------------
24 /process/sequence/if/if-condition
```

pathCoverageAll.log es uno de los tres ficheros que almacena información sobre cobertura de caminos. En concreto, lista todos los caminos detectados en la composición. En el listado B.9 se informa de que hay dos caminos de ejecución posibles en el préstamo sequence, siendo el número máximo de instrucciones posible 46, y el menor, 36. Debido a su longitud, se han eliminado de dicho listado la serie de instrucciones que forman los caminos. Estos se indican de igual manera que en el listado B.10.

Listado B.9: *Fichero `pathCoverageAll.log` con caminos detectados.*

```
1 Maximum length of an execution path: 46
2 Possible execution paths: 2
3 Minimum length of an execution path: 36
```

pathCoverageNotExecuted.log es el segundo de los tres ficheros que almacena información sobre cobertura de caminos. Contiene los caminos que no ha ejecutado ningún caso de prueba. Se observa un ejemplo en el listado B.10, que informa de un camino no ejecutado y lista las instrucciones que lo forman en la composición del préstamo sequence.

Listado B.10: *Fichero `pathCoverageNotExecuted.log` con caminos no ejecutados.*

```
1 Number of not executed paths: 1
2
3 Execution path no.1
4 36 sentences
5 /process
6 /process/sequence
7 /process/sequence/receive
8 /process/sequence/assign
9 /process/sequence/assign/copy[0]
10 /process/sequence/assign/copy[1]
11 /process/sequence/assign/copy[2]
12 /process/sequence/assign/copy[3]
13 /process/sequence/assign/copy[4]
14 /process/sequence/assign[2]
15 /process/sequence/assign[2]/copy[0]
```

```
16 /process/sequence/if
17 /process/sequence/if/if−condition
18 /process/sequence/if/if−condition/sequence
```

pathCoverageExecuted.log es el tercer y último fichero que almacena información sobre cobertura de caminos. Almacena los caminos que se han recorrido en algún momento por algún caso de prueba, para ello tiene una estructura similar al fichero `pathCoverageNotExecuted.log`, pero indicando los caminos que sí han sido ejecutados.

Se ha mostrado el contenido de los ficheros con información de cobertura en texto plano. También existe la posibilidad de generarlos en formato XML, para facilitar su procesado. Puede encontrarse más información sobre esto en [Á10].

B.2.2. Uso desde NetBeans

La extensión para NetBeans permite el uso de Takuan desde un asistente con interfaz gráfica (tipo *Wizard*). Este método es adecuado para hacer pequeñas pruebas mientras se está desarrollando una composición con NetBeans.

Nótese que esta extensión no instala Takuan, ni necesita ejecutarse en la misma máquina donde esté instalado Takuan. El asistente prepara el fichero BPEL junto a sus dependencias, lo envía para su ejecución a la instancia de Takuan que se indique (puede ser en el mismo equipo, en una máquina virtual u otro equipo), y recibe su respuesta para mostrarla al usuario.

Instalación

Para poder hacer uso de la extensión, primero necesita tener instalado un entorno NetBeans con soporte para WS-BPEL. Las instrucciones para ello dependen de la versión concreta del entorno y se puede consultar su web oficial [Com]. Una vez se haya instalado, el siguiente paso es descargar el paquete con la extensión:

```
$ wget http://neptuno.uca.es/files/netbeans/es-uca-takuan.nbm
```

Una vez se dispone del paquete, hay que abrir NetBeans y entrar en el menú `Tools` → `Plugins`. Dentro de la pestaña `Downloaded` se selecciona `Add plugins`. En el diálogo de selección de fichero se elige el fichero `.nbm` que se ha descargado.

Después se selecciona el paquete "Takuan_NetBeans" y se pulsa el botón `Install`. De este modo se iniciará el asistente de importación de NetBeans. Simplemente hay que seguir sus pasos para terminar la instalación.

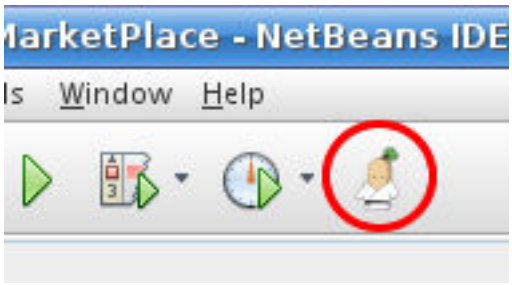


Figura B.1: *Botón para lanzar el asistente de Takuan.*

Uso

A continuación se describe el uso habitual de la extensión, que consta de los siguientes pasos secuenciales:

1. Para empezar, hay que crear un proyecto desde cero siguiendo las instrucciones de NetBeans [Com], o bien crear un proyecto vacío y copiar todos los ficheros de una composición ya existente (incluyendo los ficheros WSDL, XSD, etc.) al directorio `src` del proyecto.

2. En segundo lugar, hay que adaptar la composición WS-BPEL. Se realiza de la forma descrita en el apartado B.2.1. Para ello se puede emplear el editor de código WS-BPEL que incluye NetBeans.

3. Para poder ejecutar el asistente de Takuan, debe marcar el proyecto WS-BPEL como `Main Project`, tener el fichero BPEL seleccionado o activo en el editor y pulsar sobre el botón representado en la figura B.1. Al pulsarlo aparecerá la ventana de bienvenida al asistente.

4. El asistente solicitará que se configuren los parámetros relacionados con la ejecución e instrumentalización (figura B.2 en la página siguiente):

   Instrument variables by default: si se marca, todas las variables se inspeccionarán por defecto. Si se desmarca, solo se instrumentalizarán aquellas para las que se indique explícitamente.

   Maximum depth: profundidad máxima de <sequence> de la composición WS-BPEL que se instrumentalizará.

   Coverage output format: formato de salida de la información de cobertura del conjunto de casos de prueba. Puede escogerse XML (que facilita su posterior procesado automático), o texto plano (más legible por humanos).

   Mapping scheme: el tipo de correspondencia entre datos XML Schema y Daikon.

   Filter unused variables: filtrado de las variables que no se usan en un punto del programa.

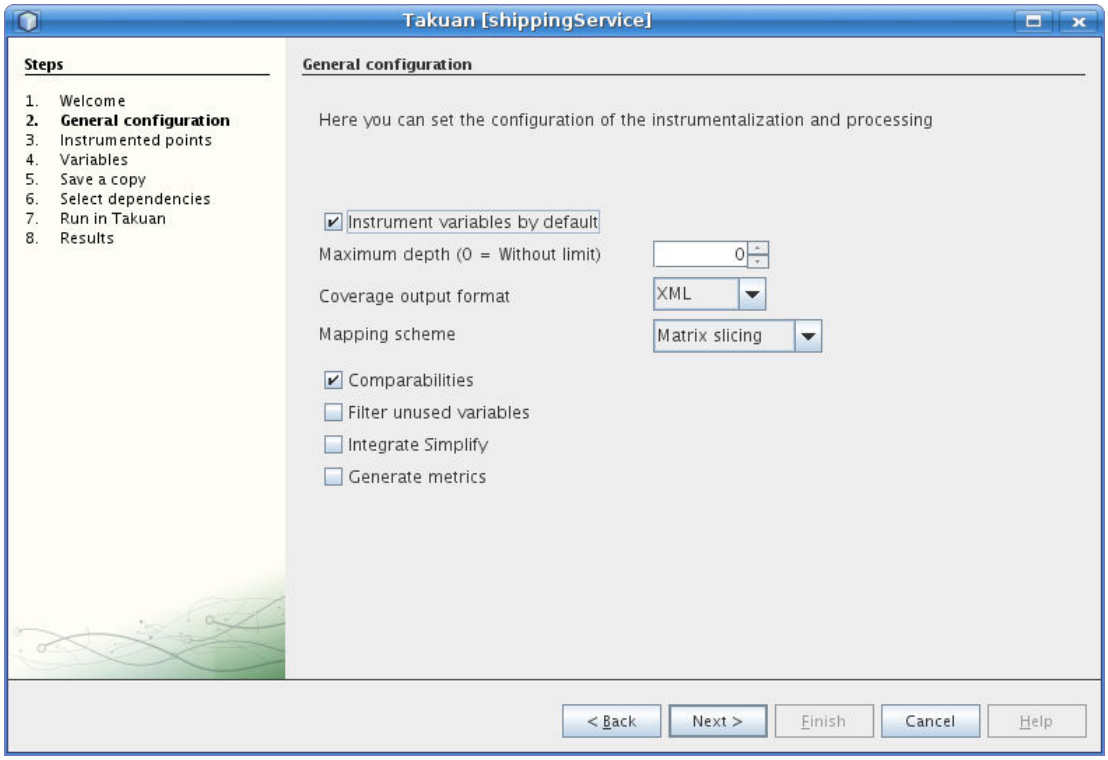


Figura B.2: *Parámetros de ejecución de Takuan.*

Integrate Simplify: ejecución de Simplify sobre el conjunto de invariantes resultantes.

Generate metrics: indica si generar o no métricas del proceso.

5. En el siguiente diálogo se debe especificar cuál es la acción por defecto para todos los puntos del programa: instrumentalizarlo o no. Después, se especificarán aquellos que tengan el comportamiento opuesto al indicado anteriormente. Es decir, si por defecto se instrumentalizan todos los puntos, se indicarán aquellos que no y viceversa.

   Una vez seleccionados los puntos de programa a instrumentalizar, se hará lo mismo con las variables. Los valores admitidos son tres:

   <**not set**>**:** se aplicará el comportamiento por defecto configurado en el primer paso del asistente.

   Yes: se inspeccionará la variable.

   No: no se inspeccionará la variable.

6. Como los cambios realizados sobre el fichero BPEL (parámetros, puntos a instrumentalizar, variables a inspeccionar, etc.) se perderán al realizar la ejecución, el

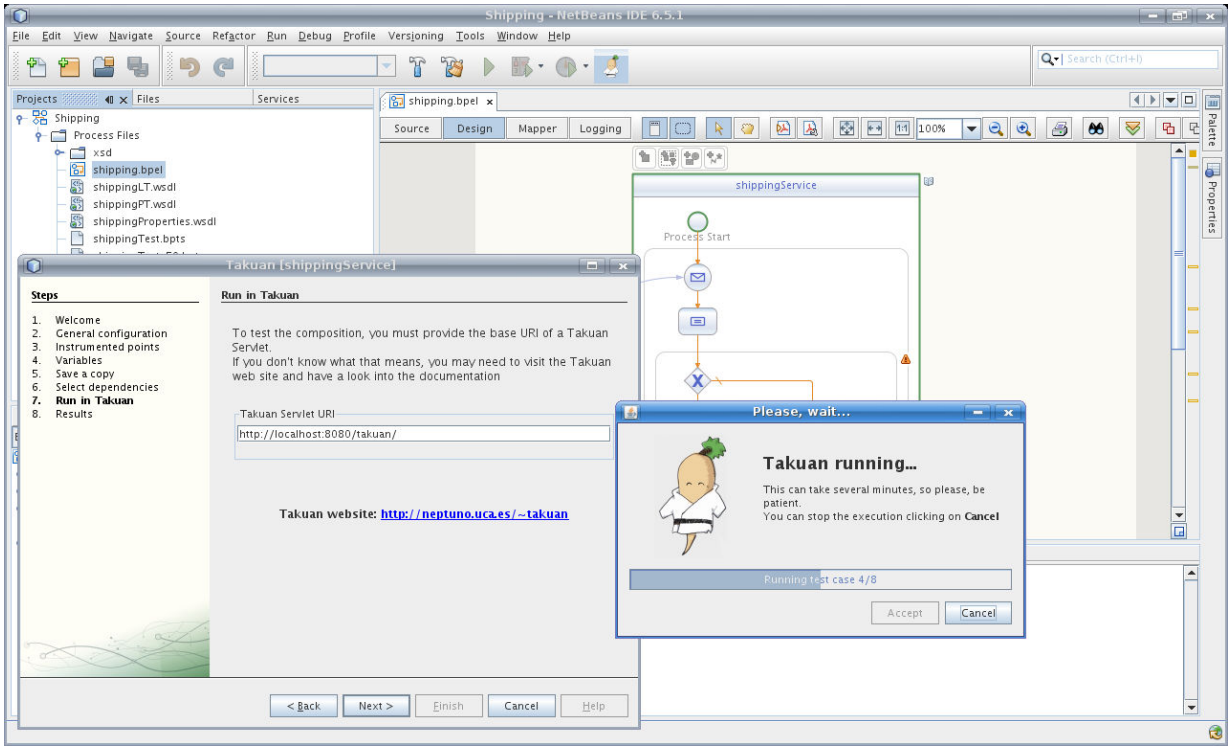


Figura B.3: *Proceso de ejecución de Takuan desde NetBeans.*

asistente permite guardar una copia del fichero BPEL modificado para facilitar que se repita la ejecución posteriormente.

7. En el siguiente diálogo se especificarán los ficheros de los que depende la composición WS-BPEL y el conjunto de casos de prueba que se usará. Mediante los botones + y − pueden añadirse o eliminarse dependencias, incluyendo ficheros externos al proyecto.

8. La última opción es la URI del servlet de Takuan al que el asistente se conectará. Una vez introducido el valor, se comenzará con la ejecución, y se mostrará el diálogo de progreso (figura B.3).

9. Se obtienen dos resultados: el conjunto de invariantes generados por Takuan e información sobre la cobertura del conjunto de casos de prueba.

   Las opciones que permite el asistente son visualizar los invariantes, añadir al proyecto de NetBeans los ficheros de invariantes e información de cobertura, y guardar dichos ficheros en disco.

   La opción de visualización está desactivada para la información de cobertura al ser un archivo comprimido con varios ficheros. Sin embargo, si se añade al proyecto, el archivo se descomprimirá y se podrá ver su contenido.

B.2.3. Uso desde IdigInBPEL

IdigInBPEL[1] es una aplicación libre de escritorio en fase *beta* que permite trabajar con Takuan de manera cómoda sin necesidad de programas auxiliares. Permite, por ejemplo, programar lotes de ejecuciones, gestionar una biblioteca de resultados de ejecuciones anteriores, optimización de la ejecución de lotes de pruebas que aumenten otros anteriormente ejecutados, etc. El sistema está realizado en Python y GTK, y se puede conseguir en [San10].

Instalación

Se recomienda realizar su instalación mediante un guión creado al efecto, que se encarga de descargar la aplicación, configurarla y establecer el entorno de Takuan. Este también sirve para actualizar una versión existente y por defecto instala tanto Takuan como IdigInBPEL. Si se deseara instalar solo uno de los componentes se le puede proporcionar el parámetro *takuan* para instalar Takuan o *idg* para instalar IdigInBPEL.

Se puede descargar y ejecutar en un sistema GNU/Linux con las siguientes órdenes:

```
$ wget https://forja.rediris.es/plugins/scmsvn/viewcvs.php/*checkout*\
/trunk/install.py?root=cusl4-idigin
$ ./install.py
```

En caso de que se haya instalado Takuan en un directorio distinto al que usa por defecto (directorio `takuan` en el directorio de entrada del usuario), debe irse a la copia de trabajo y editar el fichero de configuración `idiginbpel/trunk/home/config.xml` donde se almacenan las rutas o indicarlo en la entrada `File` → `Options` del panel de opciones.

Uso

La ejecución de *IdiginBPEL* se realiza mediante proyectos. Un proyecto incluye una composición, un conjunto de casos de prueba y trazas de ejecución.

Para crear un proyecto basta con pulsar en `New Project`. Esto llevará a la pantalla de creación de proyectos, donde se introducirá su nombre y el fichero BPEL de la composición con la que se trabajará. Durante la creación del proyecto dicho fichero será analizado e importado, y sus dependencias buscadas recursivamente e importadas igualmente.

Una vez definido el proyecto, la generación de invariantes se realiza en cinco pasos, representados por las cinco pestañas de la ventana de la figura B.4 en la página siguiente. Cada paso usa información de los anteriores, y el sistema permite volver atrás si

[1] IdigInBPEL es el acrónimo inglés de *Improved Dynamic Invariant Generation In BPEL*, en español *Generación Dinámica de Invariantes en BPEL Mejorada*. Pero tiene un doble sentido, pues *I dig in BPEL* significa en ingles *Trabajo ansiosamente en BPEL* o *Escarbo con determinación en BPEL*, haciendo referencia a la búsqueda de propiedades subyacentes en el código WS-BPEL que Takuan realiza.

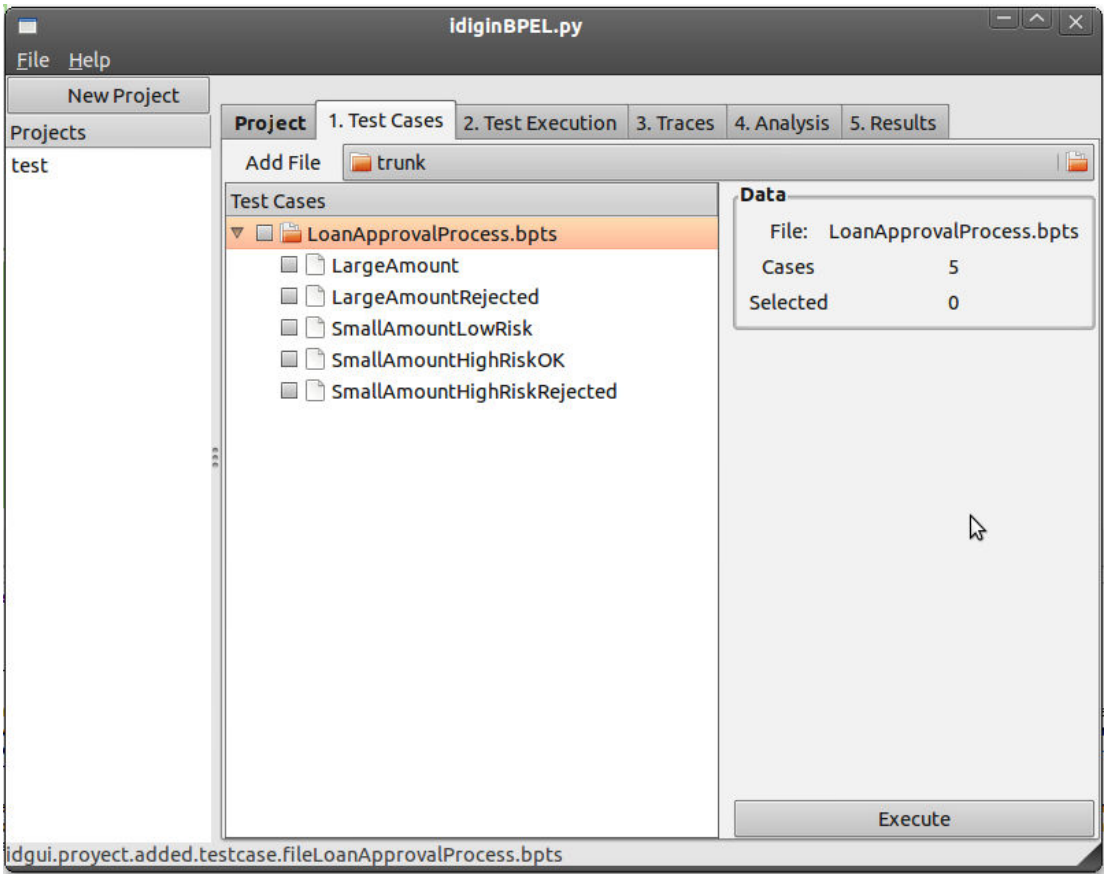


Figura B.4: *Ventana de gestión de casos de prueba de IdigInBPEL.*

se desea cambiar alguna información. Existe un botón `Instrument` para realizar la instrumentalización del fichero según las opciones que incluya. Estas opciones se pueden incorporar a mano o con el asistente para NetBeans. Para futuras versiones está planificado que pueda hacerse desde el sistema.

1. Una vez que el proyecto esté preparado, deben añadirse casos de prueba mediante la inclusión en el proyecto de ficheros BPTS en la ventana de gestión de casos (figura B.4).

2. Con el proyecto configurado y los casos de prueba disponibles, se pueden seleccionar aquellos que se desean ejecutar en el servidor ActiveBPEL. En la ventana de gestión de casos de prueba se seleccionan los casos de prueba a ejecutar y comienzan la acción empleando el botón `Execute`.

3. Durante la ejecución los distintos casos irán enviándose al servidor, siendo procesados y generando una traza que es recogida por el programa. El progreso de la ejecución puede visualizarse gráficamente en el árbol de la izquierda de la venta-

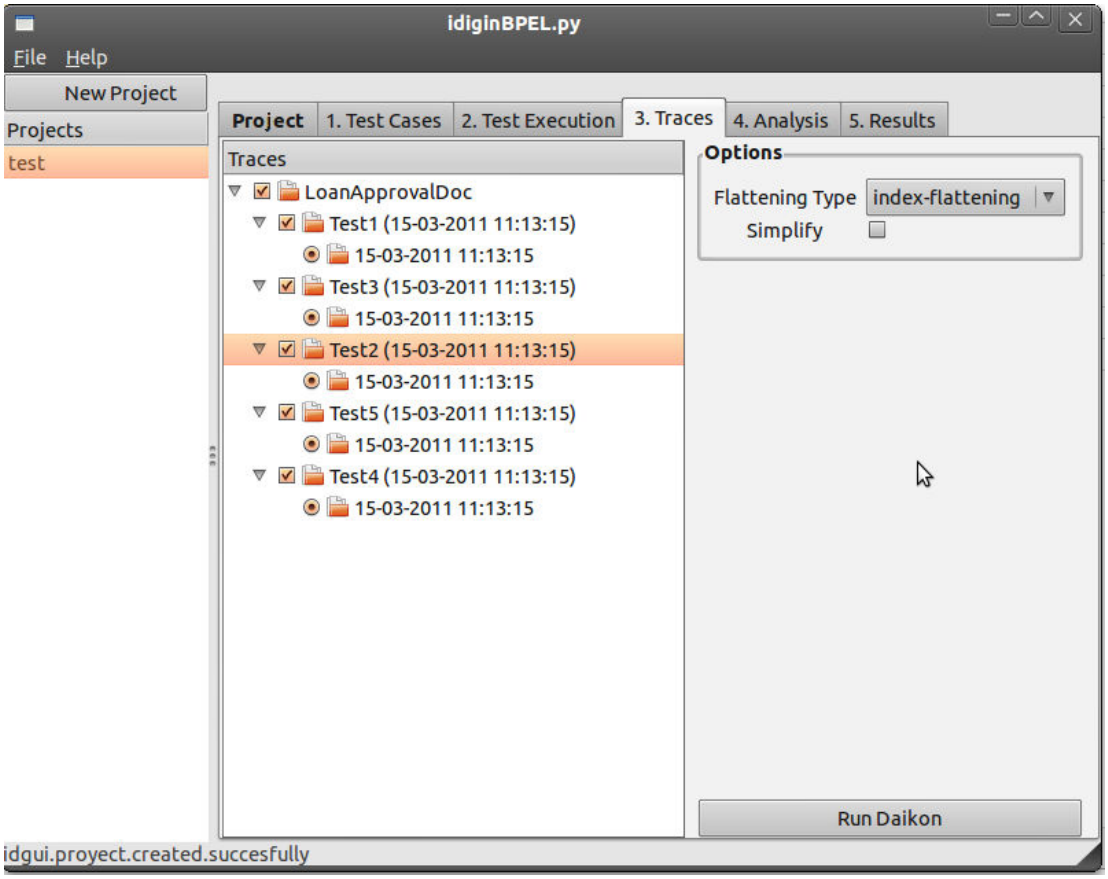


Figura B.5: *Ventana de selección de trazas en IdigInBPEL.*

na, donde se indica el caso que se ejecuta actualmente, los que no han sido ejecutados todavía y el resultado (positivo o negativo) de los que ya lo han sido.

4. Cuando se ha realizado al menos una ejecución, se dispondrá de un conjunto de trazas, sobre las que realizar un análisis. En la ventana (figura B.5) se visualizan las trazas disponibles, junto a los casos a los que corresponden. Se muestran ordenadas por fecha de creación.

5. En la ventana de trazas se puede seleccionar, del conjunto de trazas disponibles, aquellas que van a ser enviadas al análisis. Al pulsar sobre el botón `Run Daikon` las trazas son enviadas a Daikon, el generador de invariantes de Takuan, para que realice su labor.

6. Tras la ejecución completa del análisis pueden visualizarse los invariantes obtenidas por Daikon en el campo de texto que aparecerá automáticamente. El proceso de obtención y selección de trazas y generación de invariantes puede repetirse tantas veces como sea necesario para obtener diferentes resultados.

BIBLIOGRAFÍA

[Á10] Alejandro Álvarez Ayllón. Reingeniería y ampliación del generador dinámico de invariantes potenciales para composiciones de servicios web en WS-BPEL Takuan. http://dx.doi.org/10498/8940, 2010. Proyecto fin de carrera de Ingeniería en Informática.

[Abo06] Faisal Abouzaid. A mapping from Pi-Calculus into BPEL. En *Proceeding of the 2006 conference on Leading the Web in Concurrent Engineering*, páginas 235–242. IOS Press, Amsterdam, Países Bajos, 2006. ISBN 1-58603-651-3.

[Act08] Active Endpoints. ActiveBPEL WS-BPEL and BPEL4WS engine. http://sourceforge.net/projects/activebpel, 2008.

[Act09] Active Endpoints. ActiveVOS home site. http://www.activevos.com, 2009.

[Akk03] Rama Akkiraju, Richard Goodwin, Prashant Doshi y Sascha Roeder. A method for semantically enhancing the service discovery capabilities of UDDI. En *Proceedings of the Workshop on Information Integration on the Web (IJCAI)*, páginas 9–10. Citeseer, 2003.

[Ask04] Sid Askary. WS-BPEL issues list, number 42: Need for formalism. http://www.oasis-open.org/committees/download.php/20228/WS_BPEL_issues_list.html\#Issue42, 2004.

[Ass10] European Regional Information Society Association. Peardrop project. http://www.peardrop.eu/about/Pages/index.aspx, 2010.

[Atk08] Colin Atkinson, Daniel Brenner, Giovanni Falcone y Monika Juhasz. Specifying high-assurance services. *Computer*, 41(8):páginas 64–71, 2008. ISSN 0018-9162. doi:http://dx.doi.org/10.1109/MC.2008.308.

[Bai05] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai y Yinong Chen. WSDL-Based automatic test case generation for web services testing. *Service-Oriented System*

Engineering, IEEE International Workshop on, 0:páginas 215–220, 2005. doi:http://doi.ieeecomputersociety.org/10.1109/SOSE.2005.43.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, EE.UU., 2002. ISBN 0321146530.

[Ben09] Lina Bentakouk, Pascal Poizat y Fatiha Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. En *TESTCOM '09/FATES '09: Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, páginas 16–32. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-05030-5. doi:http://dx.doi.org/10.1007/978-3-642-05031-2_2.

[Ber05] Antonia Bertolino y Eda Marchetti. A brief essay on software testing. En Richard H. Thayer y Mark Christensen, editores, *Software Engineering, The Development Process*. Wiley-IEEE Computer Society Press, third edición, 2005. ISBN 0471684171.

[Bjø97] Nikolaj Bjørner, Anca Browne y Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):páginas 49–87, 1997.

[Bla09] Raquel Blanco, José García Fanjul y Javier Tuya. A first approach to test case generation for BPEL compositions of web services using scatter search. En *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, páginas 131–140. IEEE Computer Society, Washington, DC, EE.UU., 2009. ISBN 978-0-7695-3671-2. doi:http://dx.doi.org/10.1109/ICSTW.2009.24.

[Boz10] Mustafa Bozkurt, Mark Harman y Youssef Hassoun. Testing web services: A survey. Informe Técnico TR-10-01, Department of Computer Science, King's College London, 2010.

[Bre06] Van Breugel y Maria Koshkina. Models and verification of BPEL, 2006.

[Bro08] William A. Brown, Robert G. Laird, Clive Gee y Tilak Mitra. *SOA Governance: Achieving and Sustaining Business and IT Agility*. IBM Press, 2008. ISBN 0137147465, 9780137147465.

[Bru05] Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, Gianpiero Esposito y Valentina Mazza. Using test cases as contract to ensure service compliance across releases. En *In Service-Oriented Computing - ICSOC 2005, Third International Conference*, páginas 87–100. 2005.

[Buc07] Antonio Bucchiarone, Hernán Melgratti y Francesco Severoni. Testing service composition. En *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07)*. 2007.

[Cao10] Tien-Dung Cao, Patrick Felix y Richard Castanet. WSOTF: An automatic testing tool for web services composition. *Internet and Web Applications and Services, International Conference on*, 0:páginas 7–12, 2010. doi:http://doi.ieeecomputersociety.org/10.1109/ICIW.2010.9.

[Cha07] W. Chan, S. Cheung y K. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)*, 4(2):páginas 61–81, 2007.

[Cla99] Edmund M. Clarke, Orna Grumberg y Doron A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0262032708.

[Col03] Michael Colón, Sriram Sankaranarayanan y Henny Sipma. Linear invariant generation using non-linear constraint solving. En Warren A. Hunt Jr. y Fabio Somenzi, editores, *Computer Aided Verification (CAV)*, tomo 2725 de *Lecture Notes in Computer Science*, páginas 420–432. Springer, 2003. ISBN 3-540-40524-0.

[Com] NetBeans Community. Web oficial del proyecto NetBeans SOA. `http://soa.netbeans.org`.

[Com06] OASIS Web Services Security (WSS) Technical Committee. WS-Security 1.1 standard. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`, 2006. OASIS.

[Com09] OASIS Web Services Transaction (WS-TX) Technical Committee. WS-Trasaction 1.2 standard. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx`, 2009. OASIS.

[Cor07] IBM Corporation. Adapting legacy systems for SOA. `http://www.ibm.com/developerworks/webservices/library/ws-soa-adaptleg`, 2007.

[Cur03] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai y Sanjiva Weerawarana. The next step in Web Services. *Communications of the ACM*, 46(10):páginas 29–34, 2003.

[Dai07] Guilan Dai, Xiaoying Bai, Yongbo Wang y Fengjun Dai. Contract-based testing for web services. En *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, páginas 517–526. IEEE Computer Society, Washington, DC, EE.UU., 2007. ISBN 0-7695-2870-8. doi:http://dx.doi.org/10.1109/COMPSAC.2007.100.

[Dai10] Daikon development team. Daikon manual. `http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html`, 2010.

[Dav81] Martin D. Davis y Elaine J. Weyuker. Pseudo-oracles for non-testable programs. En Beth Levy, editor, *Proceedings of the ACM '81 conference*, páginas 254–257. Association for Computing Machinery, 1981.

[Det05] David Detlefs, Greg Nelson y James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):páginas 365–473, 2005.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):páginas 859–866, 1972. Turing Award lecture.

[Dom07] Juan José Domínguez Jiménez, Antonia Estero Botaro, Inmaculada Medina Bulo, Manuel Palomo Duarte y Francisco Palomo Lozano. El reto de los servicios web para el software libre. En *Proceedings of the FLOSS International Conference 2007*, páginas 117–132. Servicio de Publicaciones de la Universidad de Cádiz, Jerez de la Frontera, 2007. ISBN 978-84-9828-124-8. doi:http://flossic.org/Contenidos/actas/reto.pdf.

[Dom09] Juan José Domínguez Jiménez, Antonia Estero Botaro, Antonio García Domínguez e Inmaculada Medina Bulo. GAmera: An automatic mutant generation system for WS-BPEL compositions. En *Proceedings of the 7th European Conference on Web Services (ECOWS'09)*, páginas 97–106. Eindhoven, Países Bajos, 2009.

[dt10a] Apache ODE development team. Apache ODE WS-BPEL engine unit test compositions. `http://svn.apache.org/viewvc/ode/trunk/bpel-test/src/test/resources/bpel/2.0`, 2010.

[dt10b] BPELUnit development team. Repositorio oficial de BPELUnit. `http://github.com/bpelunit/bpelunit`, 2010.

[Dum05] M. Dumas, A. H. M. Ter Hofstede, N. Russell, H. M. W. Verbeek y P. Wohed. Life after BPEL. En *WS-FM 2005, volume 3670 of Lecture Notes in Computer Science*, páginas 35–50. Springer-Verlag, 2005.

[Eng10] Fachgebiet Software Engineering. Web oficial de BPELUnit. `http://www.se.uni-hannover.de/forschung/soa/bpelunit`, 2010.

[Erl04] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, Upper Saddle River, NJ, EE.UU., 2004. ISBN 0131428985.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, EE.UU., 2005. ISBN 0131858580.

[Ern00a] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, 2000.

[Ern00b] Michael D. Ernst, Adam Czeisler, William G. Griswold y David Notkin. Quickly detecting relevant program invariants. En *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, páginas 449–458. Limerick, Ireland, 2000.

[Ern01] Michael D. Ernst, Jake Cockrell, William G. Griswold y David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):páginas 99–123, 2001.

[Ern07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz y Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):páginas 35–45, 2007.

[Est08] Antonia Estero Botaro, Francisco Palomo Lozano e Inmaculada Medina Bulo. Mutation operators for WS-BPEL 2.0. En *ICSSEA 2008: Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications*. 2008.

[Est10] Antonia Estero Botaro, Francisco Palomo Lozano e Inmaculada Medina Bulo. Quantitative evaluation of mutation operators for WS-BPEL compositions. En *ICSTW '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, páginas 142–150. IEEE Computer Society, Washington, DC, EE.UU., 2010. ISBN 978-0-7695-4050-4. doi:http://dx.doi.org/10.1109/ICSTW.2010.36.

[Evi] Eviware. Soapui. `http://www.soapui.org`.

[Fah05] D. Fahland, W. Reisig y D. Fahland W. Reisig. ASM-based semantics for BPEL: The negative control flow. En *Proc. 12th International Workshop on Abstract State Machines*, páginas 131–151. 2005.

[Fan06] José García Fanjul, Javier Tuya y Claudio de la Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. En *Proceedings of the International Workshop on Web Services Modeling and Testing*, páginas 83–94. Palermo, Italia, 2006.

[Gar08a] Antonio García Domínguez, Manuel Palomo Duarte e Inmaculada Medina Bulo. 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*, páginas 1–6. SISTEDES, Gijón, España, 2008.

[Gar08b] Antonio García Domínguez, Manuel Palomo Duarte e Inmaculada Medina Bulo. Implementación de un framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. En José Manuel López Cobo, Antonio Vallecillo y Antonio Ruiz Cortés, editores, *Actas de las IV Jornadas Científico-Técnicas en Servicios Web y SOA*, páginas 91–96. Sevilla, España, 2008.

[Gar09] Antonio García Domínguez, Inmaculada Medina Bulo y Mariano Marcos Bárcena. Hacia la integración de técnicas de pruebas en metodologías dirigidas por

modelos para SOA. En *Actas de las V Jornadas Científico-Técnicas en Servicios Web y SOA*. Madrid, 2009.

[Gho08] Sakti Ghosh, Ali Arsanjani y Abdul Allam. SOMA: a method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):páginas 377–396, 2008.

[Gol04] Nicolas Gold, Claire Knight, Andrew Mohan y Malcolm Munro. Understanding service-oriented software. *IEEE Software*, 21(2):páginas 71–77, 2004. ISSN 0740-7459. doi:http://dx.doi.org/10.1109/MS.2004.1270766.

[Gra00] Todd L. Graves, Alan F. Karr, J. S. Marron y Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26:páginas 653–661, 2000. ISSN 0098-5589. doi:10.1109/32.859533.

[Gro05] Web Services Choreography Working Group. WS-CDL version 1.0. `http://www.w3.org/TR/ws-cdl-10`, 2005.

[Grua] Grupos SPI&FM y UCASE. Web oficial de GAmera. `http://neptuno.uca.es/~gamera`. Registro Territorial de la Propiedad Intelectual de la Junta de Andalucía, código CA-77-10.

[Grub] Grupos SPI&FM y UCASE. Web oficial de Takuan. `http://neptuno.uca.es/~takuan`. Registro Territorial de la Propiedad Intelectual de la Junta de Andalucía, código CA-76-10.

[Gru10] Grupos SPI&FM y UCASE. Repositorio de composiciones WS-BPEL. `http://neptuno.uca.es/redmine/projects/show/wsbpel-comp-repo`, 2010.

[Gup03] Neelam Gupta. Generating test data for dynamically discovering likely program invariants. En *ICSE 2003 Workshop on Dynamic Analysis (WODA 2003)*. 2003.

[Gut08] Walter J. Gutjahr y Mark Harman. Search-based software engineering. *Computers & Operations Research*, 35(10):páginas 3049–3051, 2008.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):páginas 11–19, 1990. ISSN 0740-7459. doi:http://dx.doi.org/10.1109/52.57887.

[Hal08] Tim Hallwyl. Evaluating the BPEL standard specification. `ftp://ftp.diku.dk/diku/semantics/papers/D-609.pdf`, 2008. Master thesis.

[Hal10] Tim Hallwyl, Fritz Henglein y Thomas Hildebrandt. A standard-driven implementation of WS-BPEL 2.0. En *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, páginas 2472–2476. ACM, New York, NY, EE.UU., 2010. ISBN 978-1-60558-639-7. doi:http://doi.acm.org/10.1145/1774088.1774599.

[Hec04] Reiko Heckel y Marc Lohmann. Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, 82:página 2003, 2004.

[Hef07] Randy Heffner y Larry Fulton. Topic overview: Service-oriented architecture. Forrester Research, Inc., 2007.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:páginas 279–295, 1997.

[IEE04] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EE.UU., 2004.

[Ini10] Business Process Management Initiative. BPMN FAQ. `http://www.bpmn.org/Documents/FAQ.htm`, 2010.

[Jia10] Yue Jia y Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.

[Kov07] Máté Kovács, Dániel Varró y László Gönczy. Formal modeling of BPEL workflows including fault and compensation handling. En *EFTS '07: Proceedings of the 2007 workshop on Engineering fault tolerant systems*, página 1. ACM, New York, NY, EE.UU., 2007. ISBN 978-1-59593-725-4. doi:http://doi.acm.org/10.1145/1316550.1316551.

[Kov08] Máté Kovács, László Gönczy y Dániel Varró. Formal analysis of bpel workflows with compensation by model checking. *International Journal of Computer Systems and Engineering*, 23(5), 2008.

[Lev90] Nancy G. Leveson, Stephen S. Cha, John C. Knight y Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16:páginas 432–443, 1990. ISSN 0098-5589. doi:10.1109/32.54295.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Professional, 1995.

[Liu07] Hehui Liu, Zhongjie Li, Jun Zhu y Huafang Tan. Business process regression testing. En *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, páginas 157–168. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 978-3-540-74973-8. doi:http://dx.doi.org/10.1007/978-3-540-74974-5_13.

[Loh06] Niels Lohmann, Peter Massuthe, Christian Stahl y Daniela Weinberg. Analyzing interacting BPEL processes. En *Proceedings of the 4th International Conference on Business Process Management (BPM2006), volume 4102 of Lecture Notes in Computer Science*, páginas 17–32. Springer-Verlag, 2006.

[Loh08a] Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. En *WS-FM'07: Proceedings of the 4th international conference on web services and formal*

methods, páginas 77–91. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 3-540-79229-5, 978-3-540-79229-1.

[Loh08b] Niels Lohmann, Oliver Kopp, Frank Leymann y Wolfgang Reisig. Analyzing BPEL4Chor: verification and participant synthesis. En *WS-FM'07: Proceedings of the 4th international conference on Web services and formal methods*, páginas 46–60. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 3-540-79229-5, 978-3-540-79229-1.

[Loh09] Niels Lohmann, Eric Verbeek, Chun Ouyang y Christian Stahl. Comparing and evaluating Petri net semantics for BPEL. *International Journal of Business Process Integration and Management*, 4(1):páginas 60–73, 2009.

[Lüb07] Daniel Lübke. Unit testing BPEL compositions. En Luciano Baresi y Elisabetta Di Nitto, editores, *Test and Analysis of Web Services*, páginas 149–171. Springer, 2007. ISBN 978-3-540-72912-9.

[Luc07] Roberto Lucchi y Manuel Mazzara. A Pi-Calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):páginas 96–118, 2007.

[Mar06] Eric A. Marks y Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology*. John Wiley & Sons, Inc., New York, NY, EE.UU., 2006. ISBN 0471768944.

[Mar07a] David Martin, Mark Burstein, Drew McDermott, Sheila McIlraith, Massimo Paolucci, Katia Sycara, Deborah McGuinness, Evren Sirin y Naveen Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10:páginas 243–277, 2007. ISSN 1386-145X. 10.1007/s11280-007-0033-x.

[Mar07b] Evan Martin, Suranjana Basu y Tao Xie. Automated testing and response analysis of web services. En *In Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*, páginas 647–654. 2007.

[May06a] Philip Mayer. Design and implementation of a framework for testing BPEL compositions. `http://www.pst.informatik.uni-muenchen.de/~mayer/papers/2006_03_Masters_Thesis.pdf`, 2006. Master thesis.

[May06b] Philip Mayer y Daniel Lübke. 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*, páginas 33–42. ACM, New York, NY, EE.UU., 2006. ISBN 1-59593-458-8. doi:http://doi.acm.org/10.1145/1145718.1145723.

[Mcm04] Phil Mcminn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:páginas 105–156, 2004.

[Med09] Inmaculada Medina Bulo, Antonio García Domínguez, Francisco Aguayo González, Lorenzo Sevilla Hurtado y Mariano Marcos Bárcena. Propuesta metodológica para la implementación de una arquitectura orientada a servicios en entornos de sistemas de fabricación distribuida. En *Actas del III Congreso Internacional de la Sociedad de Ingeniería de Fabricación*, páginas 346–353. Alcoy, España, 2009.

[Mei09a] Lijun Mei, W. K. Chan y T. H. Tse. Data flow testing of service choreography. En *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, páginas 151–160. ACM, New York, NY, EE.UU., 2009. ISBN 978-1-60558-001-2. doi:http://doi.acm.org/10.1145/1595696.1595720.

[Mei09b] Lijun Mei, Zhenyu Zhang, W. K. Chan y T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. En *WWW '09: Proceedings of the 18th international conference on World wide web*, páginas 901–910. ACM, New York, NY, EE.UU., 2009. ISBN 978-1-60558-487-4. doi:http://doi.acm.org/10.1145/1526709.1526830.

[Mel] Matteo Melideo. Web oficial del proyecto SeCSE. `http://www.secse-project.eu`.

[Mor08] Shoichi Morimoto. A survey of formal verification for business process modeling. En *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part II*, páginas 514–522. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69386-4. doi:http://dx.doi.org/10.1007/978-3-540-69387-1_58.

[Mye04] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, second edición, 2004. ISBN 0471469122.

[Nac07] Francesco Nachira, Paolo Dini, Andrea Nicolai, Marion Le Louarn y Lorena Rivera Leon, editores. *Digital Business Ecosystems book*. Luxembourg: European Commission, 2007.

[New04] Eric Newcomer y Greg Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004. ISBN 0321180860.

[OAS98] OASIS. About OASIS. `http://www.oasis-open.org/who`, 1998.

[OAS03] 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`, 2003.

[OAS05a] OASIS. OASIS web service implementation methodology (public review draft). `http://www.oasis-open.org/committees/download.php/13420/fwsi-im-1.0-guidlines-doc-wd-publicReviewDraft.htm#_Toc105485370`, 2005.

[OAS05b] OASIS. UDDI standard 3.0. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm, 2005.

[OAS06] OASIS. OASIS reference model for service oriented architecture 1.0. http://www.oasis-open.org/committees/soa-rm, 2006.

[OAS07a] OASIS. WS-BPEL 2.0 primer. http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html, 2007.

[OAS07b] OASIS. WS-BPEL 2.0 standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[OAS09] OASIS. OASIS reference architecture foundation for service oriented architecture 1.0. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-cd-02.pdf, 2009.

[O'C97] Robert O'Callahan y Daniel Jackson. Lackwit: A program understanding tool based on type inference. En *Proceedings of the 1997 International Conference on Software Engineering*, páginas 338–348. ACM Press, 1997.

[odt10] Pfe otasw development team. Web oficial de Pfc-otasw. http://code.google.com/p/pfe-otasw, 2010.

[Off99] A Jefferson Offutt, Yiwei Xiong y Shaoying Liu. Criteria for generating specification-based tests. En *Engineering of Complex Computer Systems, 1999. ICECCS'99. Fifth IEEE International Conference on*, páginas 119–129. IEEE, 1999.

[Ora07] Oracle. Testing BPEL processes. Oracle Application Server 10g Documentation, 2007.

[Ora10] Oracle. Oracle BPEL Process Manager. http://www.oracle.com/technetwork/middleware/bpel/overview, 2010.

[Org] Web Services Interoperability Organization. Web oficial de WS-I. http://www.ws-i.org.

[Org10] Web Services Interoperability Organization. Deliverables from the basic profile working group. http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile, 2010.

[Pal08a] Marcos Palacios, José García Fanjul, Javier Tuya y Claudio de la Riva. Estado del arte en la investigación de métodos y herramientas de pruebas para procesos de negocio WS-BPEL. En José Manuel López Cobo, Antonio Vallecillo y Antonio Ruiz-Cortés, editores, *Actas de las IV Jornadas Científico-Técnicas en Servicios Web y SOA*, páginas 132–137. Sevilla, España, 2008.

[Pal08b] Manuel Palomo Duarte, Antonio García Domínguez e Inmaculada Medina Bulo. An architecture for dynamic invariant generation in WS-BPEL web service compositions. En *Proceedings of ICE-B 2008 - International Conference on e-Business*. INSTICC Press, Oporto, Portugal, 2008.

[Pal08c] Manuel Palomo Duarte, Antonio García Domínguez e Inmaculada Medina Bulo. Improving Takuan to analyze a meta-search engine WS-BPEL composition. En *SOSE '08: Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, páginas 109–114. IEEE Computer Society, Washington, DC, EE.UU., 2008. ISBN 978-0-7695-3499-2. doi:http://dx.doi.org/10.1109/SOSE.2008.18.

[Pal08d] Manuel Palomo Duarte, Antonio García Domínguez e Inmaculada Medina Bulo. Takuan: A dynamic invariant generation system for WS-BPEL compositions. En *ECOWS '08: Proceedings of the 2008 Sixth European Conference on Web Services*, páginas 63–72. IEEE Computer Society, Washington, DC, EE.UU., 2008. ISBN 978-0-7695-3399-5. doi:http://dx.doi.org/10.1109/ECOWS.2008.17.

[Pal09a] Manuel Palomo Duarte, Alejandro Álvarez Ayllón, Antonio García Domínguez e Inmaculada Medina Bulo. La cobertura de los casos de prueba en la generación dinámica de invariantes en composiciones WS-BPEL. En *Actas de Talleres de Ingeniería del Software y Bases de Datos*, páginas 1–7. SISTEDES, San Sebastián, España, 2009.

[Pal09b] Manuel Palomo Duarte, Antonio García, Alejandro Álvarez e Inmaculada Medina Bulo. Takuan: generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. En Antonio Vallecillo y Goiuria Sagardui, editores, *JISBD*, páginas 367–370. 2009. ISBN 978-84-692-4211-7.

[Pal09c] Manuel Palomo Duarte, Antonio García Domínguez e Inmaculada Medina Bulo. Enhancing WS-BPEL dynamic invariant generation using XML Schema and XPath Information. En *ICWE '09: Proceedings of the 9th International Conference on Web Engineering*. San Sebastián, España, 2009.

[Pal10] Manuel Palomo Duarte, Antonio García Domínguez, Inmaculada Medina Bulo, Alejandro Álvarez Ayllón y Javier Santacruz. Takuan: A tool for ws-bpel composition testing using dynamic invariant generation. En Boualem Benatallah, Fabio Casati, Gerti Kappel y Gustavo Rossi, editores, *ICWE*, tomo 6189 de *Lecture Notes in Computer Science*, páginas 531–534. Springer, 2010. ISBN 978-3-642-13910-9.

[Pan08] Kapil Pant. *Business Process Driven SOA using BPMN and BPEL: From Business Process Modeling to Orchestration and Service Oriented Architecture*. Packt Publishing, 2008. ISBN 1847191460.

[Pao02] Massimo Paolucci, Takahiro Kawamura, Terry Payne y Katia Sycara. Importing the semantic web in UDDI. En Christoph Bussler, Richard Hull, Sheila McIlraith,

Maria Orlowska, Barbara Pernici y Jian Yang, editores, *Web Services, E-Business, and the Semantic Web*, tomo 2512 de *Lecture Notes in Computer Science*, páginas 815–821. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36189-8_18.

[Pap07] Michael P Papazoglou. Web services technologies and standards, 2007. Computing Surveys (submitted).

[Par] Parasoft. Parasoft SOAtest. `http://www.parasoft.com/jsp/solutions/soa_solution.jsp?itemId=319#bpel`.

[PD11] Manuel Palomo Duarte. Web con información adicional sobre la tesis. `http://purl.org/mpalomo/tesis`, 2011.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):páginas 223–252, 1977. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/356698.356702.

[Ram07] Isabel Ramos Román, Pablo Javier Tuya González y Javier Dolado Cosín. *Técnicas Cuantitativas para la Gestión en la Ingeniería del Software*. Netbiblo, 2007. ISBN 978-84-9745-2.

[Rap85] Sandra Rapps y Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:páginas 367–375, 1985. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.1985.232226.

[San10] Francisco Javier Santacruz López-Cepero. Idiginbpel. `http://cusl4-idigin.forja.rediris.es`, 2010.

[SeC08] SeCSE. Testing method definition V4 (final). `http://www.secse-project.eu/wp-content/uploads/a1d34-testing-method-definition-v4-final.pdf`, 2008. Service Centric System Engineering.

[Sne07] Harry M. Sneed y Shihong Huang. The design and use of WSDL-Test: a tool for testing web services: Special issue articles. *J. Softw. Maint. Evol.*, 19(5):páginas 297–314, 2007. ISSN 1532-060X. doi:http://dx.doi.org/10.1002/smr.v19:5.

[Sof04] Software Engineering Education Project (SWEEP) members. *Computing Curricula - Software Engineering 2004*. IEEE Computer Society and Association for Computing Machinery, 2004.

[Sri05] Naveen Srinivasan, Massimo Paolucci y Katia Sycara. An efficient algorithm for OWL-S based semantic search in UDDI. En Jorge Cardoso y Amit Sheth, editores, *Semantic Web Services and Web Process Composition*, tomo 3387 de *Lecture Notes in Computer Science*, páginas 96–110. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30581-1_9.

[Ste09] Alin Stefanescu, Sebastian Wieczorek y Andrei Kirshin. Mbt4chor: A model-based testing approach for service choreographies. En Richard F. Paige, Alan Hartman y Arend Rensink, editores, *ECMDA-FA*, tomo 5562 de *Lecture Notes in Computer Science*, páginas 313–324. Springer, 2009. ISBN 978-3-642-02673-7.

[Tan10] WSO2 Oxygen Tank. WSO2 Business Process Server WS-BPEL samples. `https://wso2.org/repos/wso2/branches/bps/1.1.0/product/modules/samples/src/main/resources/bpel/2.0`, 2010.

[Tsa02] W. T. Tsai, Ray Paul, Yamin Wang, Chun Fan y Dong Wang. Extending WSDL to facilitate web services testing. En *HASE '02: Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering*, página 171. IEEE Computer Society, Washington, DC, EE.UU., 2002. ISBN 0-7695-1769-2.

[W3C04a] W3C. Web services architecture, W3C working group note 11 february 2004. `http://www.w3.org/TR/ws-arch`, 2004.

[W3C04b] W3C. Web services glossary. `http://www.w3.org/TR/ws-gloss`, 2004.

[W3C04c] W3C. XML Schema primer. `http://www.w3.org/TR/xmlschema-0`, 2004.

[W3C05a] W3C. WS-CDL 1.0 candidate recommendation. `http://www.w3.org/TR/ws-cdl-10`, 2005.

[W3C05b] W3C. WSDL 1.1 W3C note. `http://www.w3.org/TR/wsdl`, 2005.

[W3C06] W3C. WSDL 1.1 to 2.0 converter. `http://www.w3.org/2006/02/WSDLConvert.html`, 2006.

[W3C07a] W3C. SOAP 1.2 recommendation. `http://www.w3.org/2000/xp/Group`, 2007.

[W3C07b] W3C. Web Services Policy 1.5 recommendation. `http://www.w3.org/TR/ws-policy`, 2007. W3c.

[W3C07c] W3C. WSDL 2.0 recommendation. `http://www.w3.org/TR/wsdl20`, 2007.

[W3C07d] W3C. XSL Recommendations (incluye XPath 1.0 y 2.0). `http://www.w3.org/Style/XSL`, 2007.

[W3C09] W3C. About W3C. `http://www.w3.org/Consortium`, 2009.

[W3C10] W3C. XML Schema. `http://www.w3.org/XML/Schema`, 2010.

[W3S10] W3Schools. XPath tutorial. `http://www.w3schools.com/XPath`, 2010.

[Web10] WebserviceX.NET. List of web available services. `http://www.webservicex.net`, 2010.

[Wei07] Matthias Weidlich, Gero Decker y Mathias Weske. Efficient analysis of BPEL 2.0 processes using Pi-Calculus. En Jie Li, Minyi Guo, Qun Jin, Yongbing Zhang, Liang-Jie Zhang, Hai Jin, Masahiro Mambo, Jiro Tanaka y Hiromu Hayashi, editores, *IEEE Asia-Pacific Service Computing Conference (APSCC)*, páginas 266–274. IEEE, 2007. ISBN 0-7695-3051-6.

[Woo93] Martin R. Woodward. Mutation testing —its origin and evolution. *Information and Software Technology*, 35(3):páginas 163–169, 1993.

[XMe10] XMethods. List of free web services. `http://www.xmethods.net`, 2010.

[Yeo09] Gwyduk Yeom, Wei-Tek Tsai, Xiaoying Bai y Dugki Min. Design of a contract-based web services QoS management system. En *ICDCSW '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, páginas 306–311. IEEE Computer Society, Washington, DC, EE.UU., 2009. ISBN 978-0-7695-3660-6. doi:http://dx.doi.org/10.1109/ICDCSW.2009.74.

[Yoo07] Hoijin Yoon. A convergence of context-awareness and service-orientation in ubiquitous computing. *International Journal of Computer Science and Network Security*, 7(3):páginas 253–257, 2007.

[Zak09] Zulfa Zakaria, Rodziah Atan, Abdul Azim Abdul Ghani y Nor Fazlida Mohd. Sani. Unit testing approaches for BPEL: A systematic review. *Asia-Pacific Software Engineering Conference*, 0:páginas 316–322, 2009. ISSN 1530-1362. doi:http://doi.ieeecomputersociety.org/10.1109/APSEC.2009.72.

[Zhe07] Yongyan Zheng, Jiong Zhou y Paul Krause. An automatic test case generation framework for web services. *JSW*, 2(3):páginas 64–77, 2007.

[Zho04] Zhi Quan Zhou, D. H. Huang, T. H. Tse, Zongyuan Yang, Haitao Huang y T. Chen. Metamorphic testing and its applications. En *Proceedings of the 8th International Symposium on Future Software Technology - ISFST 2004*, páginas 316–322. 2004.

[Zhu97] Hong Zhu, Patrick A. V. Hall y John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):páginas 366–427, 1997. ISSN 0360-0300.

Esta tesis se terminó de imprimir el día 7 de abril de 2011,
festividad de San Juan Bautista de la Salle, educador y
fundador de los Hermanos de las Escuelas Cristianas.