ESCUELA SUPERIOR DE INGENIERÍA

Segundo Ciclo en Ingeniería Informática

Implementación de operadores de mutación para WS-BPEL 2.0

Curso 2009-2010

Juan Boubeta Puig

Cádiz, 14 de julio de 2010

ESCUELA SUPERIOR DE INGENIERÍA

Segundo Ciclo en Ingeniería Informática

Implementación de operadores de mutación para WS-BPEL 2.0

DEPARTAMENTO: Lenguajes y Sistemas Informáticos.
DIRECTORES DEL PROYECTO: Inmaculada Medina Bulo y Antonio García Domínguez.
AUTOR DEL PROYECTO: Juan Boubeta Puig.

Cádiz, 14 de julio de 2010

Fdo.: Juan Boubeta Puig

Índice general

Índice de figuras

Índice de tablas

1 Introducción

El alumno de este Proyecto Fin de Carrera (PFC, en adelante) colabora y trabaja en una de las líneas del grupo de investigación SPIFM (Software Process Improvement and Formal Methods)[1], denominada "Ingeniería de la Web", concretamente en las pruebas de composiciones de WS (Web Services). De aquí en adelante, nos referiremos siempre a esta línea de investigación.

Las composiciones de WS que utilizan el grupo para realizar estas pruebas están escritas en WS-BPEL (Web Services Business Process Execution Language) 2.0 [43], lenguaje que facilita dichas composiciones, estandarizado por OASIS (Organization for the Advancement of Structured Information Standards). Los WS permiten un desarrollo rápido de aplicaciones, caracterizado tanto por un coste bajo, como por la facilidad de componer aplicaciones distribuidas. Algunos de los beneficios que aporta WS-BPEL son: es un lenguaje estándar de industria para expresar procesos de negocio, expresado por completo en XML (eXtensible Markup Language), portable puesto que se ejecutará en cualquier motor WS-BPEL y ayuda a la programación en gran escala.

El impacto económico de los WS y las composiciones WS-BPEL se ha incrementado, por ello es necesario prestar especial atención a las pruebas de este tipo de software [23].

Gran parte de las investigaciones del grupo se centran en las pruebas de mutaciones, que como explicaremos más adelante, es una técnica de prueba del software de caja blanca basada en errores. Para ello, a partir de un programa original se generan un gran número de programas, denominados *mutantes*, que contienen una única diferencia con respecto al programa original[2]. Los mutantes se generan aplicando al código fuente un conjunto de reglas definidas previamente, los *operadores de mutación*, que introducen pequeños cambios sintácticos basados en los errores que suelen cometer habitualmente los programadores, o bien pretenden forzar ciertos criterios de

[1] El grupo de investigación SPIFM se creó en el año 2004, dentro del Plan Andaluz de Investigación, que le asigna la referencia TIC-195. Está integrado por un grupo de profesores del Departamento de Lenguajes y Sistemas Informáticos y tiene su sede en la Universidad de Cádiz. Como su nombre indica, las líneas de investigación del grupo se centran en los campos de la mejora del proceso software y en los métodos formales, teniendo como objetivo final contribuir a mejorar la calidad del software.

[2] Los mutantes que contienen una única diferencia con respecto al programa original se denominan "mutantes de primer orden" (First Order Mutant). A los mutantes que contienen más de una diferencia se le denominan "mutantes de orden superior" (Higher Order Mutant).

cobertura del código. Estos operadores introducen cambios en el programa a probar manteniendo su validez sintáctica.

El alumno ayudará a completar una herramienta real denominada GAmera [13] desarrollada en este grupo de investigación. GAmera es una herramienta de generación de mutantes para WS-BPEL que en vez de generarlos todos genera sólo un subconjunto de ellos. Dicha herramienta se divide en dos partes: un algoritmo genético que dirige todo el proceso y un entorno que compara los mutantes con el proceso original, para comprobar si su comportamiento varía o no. Este PFC se enmarca dentro de esta segunda parte.

Para que esta herramienta pueda funcionar necesita los operadores de mutación definidos para WS-BPEL 2.0 [16, 17]. Sin estos operadores, esta herramienta no podría generar los mutantes a partir del código original escrito en WS-BPEL y, por tanto, no se podrían llevar a cabo investigaciones sobre el análisis de mutaciones, proceso que mide la calidad de conjuntos de casos de prueba [21].

La propuesta de este PFC nace de la demanda por parte del grupo de disponer de todos los operadores de mutación para WS-BPEL necesarios para completar las investigaciones en el análisis de mutaciones de dichas composiciones Web. Hasta la fecha, antes de poner en marcha este PFC, se habían implementado sólo parte de estos operadores de mutación.

La aportación principal de este PFC es implementar el resto de operadores de mutación, 7 en concreto, hasta completar los 25 operadores definidos por Estero y col. [16, 17]. Como comentaremos más adelante, cabe destacar la complejidad de esta tarea, puesto que algunos de estos operadores mutarán código específico del lenguaje WS-BPEL, que no encontraremos en ningún otro lenguaje de programación tradicional.

Los ejemplos existentes sobre este lenguaje de ejecución de procesos de negocio son muy escasos y los que podemos encontrar en la bibliografía suelen ser ejemplos simples cuya finalidad es su explicación dentro de un entorno académico, más que ejemplos que se utilicen en entornos reales. Por tanto, nos encontramos ante un proyecto de investigación, en el sentido de que será necesario conocer muy bien el lenguaje para mejorar los ejemplos de estas composiciones Web existentes a las que sean aplicables el mayor número de operadores de mutación definidos para WS-BPEL.

En este PFC también se ha pretendido estudiar y analizar las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL y los definidos para otros lenguajes. Es la primera vez que se ha realizado esta comparativa y presentado en un artículo [4], que está aceptado en el V Taller de PRIS (Pruebas en Ingenieria del Software) 2010.

1.1. Pruebas de software

Las pruebas de software [47, 52], *software testing*, consisten en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, adecuadamente seleccionados del dominio de ejecuciones, por lo general infinito, contra el comportamiento especificado esperado.

Myers [41] define las pruebas de software como:

> "Hacer pruebas es el proceso de ejecutar un programa con la intención de encontrar errores." (Myers 1979)

Esta definición es simple pero precisa, y determina cuál debe ser la actitud de una persona que se dedique a probar software. No es lo mismo probar que un software funciona correctamente (no contiene errores) que demostrar que no funciona (tiene errores).

La definición de Dijkstra [12] sobre las pruebas de software es más determinante:

> "Las pruebas del software pueden probar la presencia de errores pero no la ausencia de ellos." (Dijkstra 1972)

Algunos niveles de prueba software pueden ser:

Pruebas de unidad Se utiliza para probar una sola clase, rutina o componente, de forma aislada del resto del sistema al que pertenece.

Pruebas de integración Se utiliza para realizar pruebas de componentes combinados o integrados de un sistema software.

Pruebas de regresión Se emplea para la ejecución repetida de casos de prueba para encontrar errores de depuración en software de actualización, que ya ha pasado todas las pruebas.

Pruebas del sistema Es el proceso de probar el sistema completo y final, que incluye a los usuarios e interacciones con otros sistemas.

Existen dos tipos de prueba de software:

Dinámicas o estáticas Según si ejecutan o no el software a probar.

Caja negra o blanca Según si la prueba está basada en el análisis de la especificación del software o de la estructura interna (código fuente) del software.

Cuando se realizan pruebas de software es muy importante tener siempre presente el siguiente oráculo:

> "Un mecanismo que produce la salida esperada del programa de acuerdo a sus especificaciones con objeto de compararla con la salida real del programa para los diferentes casos de prueba. Cualquier agente que es capaz

de decidir si el comportamiento del programa es el correcto respecto a un caso de prueba.”

El resultado de la ejecución de una prueba unitaria puede ser:

Fracaso Las expectativas se violan.

Error Se produce una situación inesperada.

Éxito Se superan todas las expectativas sin producirse situaciones inesperadas.

1.2. Pruebas de mutaciones

La prueba de mutaciones es una técnica de prueba del software de caja blanca basada en errores.

El análisis de mutaciones es el proceso de medir la calidad de conjuntos de casos de prueba [21]. El proceso genérico de análisis de mutación puede observarse en la Figura 1.1, extraída de [25]. Para ello genera un gran número de programas *p’*, denominados *mutantes*, que contienen una o varias diferencias con respecto al programa original *p*.

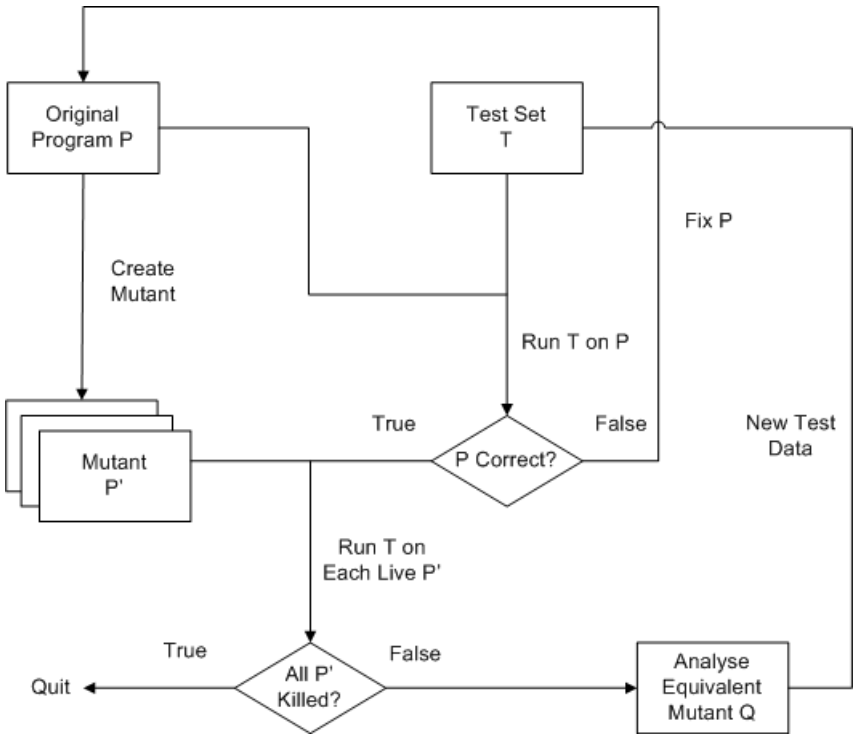

Figura 1.1: Proceso genérico de análisis de mutaciones

Los mutantes se generan aplicando al código fuente un conjunto de reglas definidas previamente, los *operadores de mutación*, que introducen pequeños cambios sintácticos basados en los errores que suelen cometer habitualmente los programadores, o bien pretenden forzar ciertos criterios de cobertura del código. Estos operadores de mutación introducen cambios en el programa a probar manteniendo su validez sintáctica. Veamos un ejemplo:

Código original:

```
if(cantidad < 10000)
  ...
```

Mutante generado por un operador de mutación de una expresión relacional:

```
if(cantidad > 10000)
  ...
```

Una vez generados, los mutantes se ejecutan sobre los casos de prueba; si la salida que produce el mutante es diferente de la que produce el programa original sobre un determinado caso de prueba, se dice que el mutante está muerto. En ocasiones, aparecen mutantes que siempre provocan la misma salida que el programa original, por lo que no va a existir ningún caso de prueba que permita matarlos; éstos se denominan *mutantes equivalentes*. Por tanto, los mutantes pueden ser clasificados como:

Muerto Un caso de prueba distingue al programa original del mutante.

Vivo Ningún caso de prueba diferencia al mutante del programa original.

- **Equivalente** El mutante tiene la misma salida que el programa original.
- **Resistente** El conjunto de casos de prueba no es suficiente para detectarlo.

Erróneo El mutante no puede ser ejecutado.

Para medir la calidad de un conjunto de casos de prueba debemos eliminar los mutantes equivalentes (esto suele realizarse a mano), ya que esta calidad se va a calcular mediante la *puntuación de mutación* (*mutation score*), el cociente entre el número de mutantes muertos y el número de mutantes no equivalentes. Una de las dificultades que presenta las pruebas de mutaciones es la detección de mutantes equivalentes. Es importante determinar si el conjunto de casos de prueba definido es suficiente para diferenciar los mutantes.

Los elementos necesarios para realizar las pruebas de mutaciones son: la definición de los operadores de mutación y el código principal del programa. Para ello, se llevarán a cabo los siguientes pasos:

1. Generar los mutantes.

2. Ejecutar los mutantes: cada mutante se ejecuta sobre el conjunto de casos de prueba.
3. Comparar los mutantes: se compara la salida del mutante con la salida del programa original.

Las definiciones de los operadores de mutación dependen del lenguaje de programación: C, Java, Fortran, SQL, WS-BPEL, etc. Existen operadores de mutación que modelan errores típicos cometidos por los programadores y otros que intentan cumplir algunos criterios de cobertura de pruebas como, por ejemplo, la ejecución de todas las sentencias o la ejecución de todas las condiciones.

Otro de los problemas de esta técnica de prueba de software es el alto coste computacional que conlleva. Esto se debe a que normalmente se dispone de un gran número de operadores de mutación que generan una gran cantidad de mutantes, y a que cada uno de estos mutantes deberán ser ejecutados sobre el conjunto de casos de prueba hasta que muera.

Para ello, existen varias técnicas para reducir el número de mutantes:

Mutant Sampling Sólo se ejecuta un subconjunto de mutantes relacionados aleatoriamente.

Selective Mutation Sólo se aplica un subconjunto de los operadores de mutación definidos para el lenguaje.

Mutant Clustering Agrupa a los mutantes en función del conjunto de casos de prueba que los mata, y de cada grupo seleccciona un número pequeño de mutantes.

Evolutionary Mutation Testing Genera de forma seleccionada un subconjunto de mutantes mediante un algoritmo evolutivo.

También existen varias técnicas para reducir el tiempo dedicado a cada mutante:

Mutación Débil Compara el estado interno del mutante y el programa original inmediatamente después de la porción mutada.

Mutación Fuerte Compara el estado intermedio después de la ejecución, así como la salida final.

Técnicas de optimización del tiempo de ejecución Ejecutan al mutante lo más rápido posible como, por ejemplo, empleando compiladores específicos, metamutantes y generadores a partir de código objeto.

Soporte de plataformas avanzadas Distribución del coste entre varios equipos, como hipercubos, redes de computadores, procesadores vectoriales, etc.

1.3. Objetivos

El objetivo principal de este PFC es implementar un conjunto de operadores de mutación para el lenguaje WS-BPEL 2.0 que permita modelar los fallos comunes que pueden cometer los programadores a la hora de escribir un programa en este lenguaje.

Los operadores para WS-BPEL, propuestos en [16, 17], se clasifican en cuatro categorías atendiendo al tipo de elemento sintáctico de WS-BPEL al que afectan: mutación de identificadores, mutación de expresiones, mutación de actividades (clasificados a su vez en dos tipos: los relacionados con la concurrencia y los no concurrentes) y mutación de condiciones excepcionales y eventos.

En este PFC se van a implementar algunos de estos operadores de mutación para WS-BPEL, en concreto:

- **EMF**: modifica una expresión de fecha límite.
- **ASI**: intercambia el orden de dos actividades hijas de una actividad `sequence`.
- **XMF**: elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos.
- **XMT**: elimina la definición de un manejador de terminación.
- **XTF**: cambia el fallo lanzado por una actividad `throw`.
- **XER**: elimina una actividad `rethrow`.
- **XEE**: elimina un elemento `onEvent` de un manejador de eventos.

Este objetivo principal puede subdividirse en cinco:

- Estudiar y analizar las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL y los definidos para otros lenguajes. Es la primera vez que se ha realizado esta comparativa y presentado en un artículo [4], que está aceptado en el V Taller de PRIS 2010.
- Implementar el código XSLT (eXtensible Stylesheet Language Transformations) de cada operador, que realice la mutación adecuada del código original escrito en WS-BPEL.
- Definir los casos de prueba, utilizando el framework JUnit, para comprobar que los operadores se han implementado correctamente.
- Adaptar un ejemplo de una composición WS-BPEL existente para que le sea aplicable, al menos, los operadores que se van a implementar.
- Crear el conjunto de casos de prueba BPTS (BPELUnit Test Suite) y comprobar que contiene los casos de prueba suficientes para matar a todos los mutantes no equivalentes que produce la composición WS-BPEL adaptada. El conjunto de casos de prueba para BPELUnit, denominado BPTS, es un fichero XML que define qué proceso se va a probar y cómo.

1.4. Alcance

El PFC se compone de varios productos:

- Un estudio y análisis de las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL y los definidos para otros lenguajes.
- Las hojas XSLT de los operadores de mutación implementados.
- El conjunto de casos de prueba JUnit para probar que los operadores de mutación se han implementado correctamente.
- La adaptación y mejora de la composición WS-BPEL *TravelReservationService*, dentro de este PFC.
- El conjunto de casos de prueba BPTS necesarios para matar a todos los mutantes no equivalentes que produce dicha composición.

1.5. Visión general

En primer lugar, se presenta el estado del arte de los operadores de mutación existentes definidos para el lenguaje WS-BPEL y para otros lenguajes de programación.

Tras este estudio exhaustivo, se realiza un análisis específico sobre cuáles son los operadores de mutación de otros lenguajes comparables con los definidos para WS-BPEL.

En tercer lugar, se presenta el calendario seguido para llevar a cabo este PFC.

A lo largo del resto de la memoria se detalla el proceso de análisis, diseño, codificación y pruebas que se ha seguido al realizar el proyecto.

Los manuales de instalación, de usuario y del desarrollador se incluyen tras un resumen de los aspectos más destacables del proyecto y las conclusiones.

1.6. Glosario

1.6.1. Acrónimos

API Application Programming Interface

BPTS BPELUnit Test Suite

CVS Concurrent Versions System

DBMS DataBase Management System

DOM Document Object Model

DTD Document Type Definition

ebXML Electronic Business using eXtensible Markup Language

FCT Fault-Compensation-Termination

GNU GNU is Not Unix

HAZOP Hazard and Operability Studies

HTML Hyper Text Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IIOP Internet Inter-ORB Protocol

IMA Inbound Message Activity

MEP Message Exchange Pattern

OASIS Organization for the Advancement of Structured Information Standards

OME Object Mutation Engine

RMI Java Remote Method Invocation

RPC Remote Procedure Call

PRIS Pruebas en Ingenieria del Software

SAX Simple API for XML

SCV Sistema de Control de Versiones

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SPIFM Software Process Improvement and Formal Methods

UDDI Universal Description Discovery and Integration

UML Unified Modeling Language

URL Uniform Resource Locator

W3C World Wide Web Consortium

XDM XML Data Model

XML eXtensible Markup Language

XPath XML Path Language

XQuery XML Query

XSLT eXtensible Stylesheet Language Transformations

XSL eXtensible Stylesheet Language

W3C World Wide Web Consortium

WS Web Services

XSD XML Schema Definition

WS-BPEL Web Services Business Process Execution Language

WSDL Web Services Description Language

WS-I Web Services Interoperability Organization

1.6.2. Definiciones

BPR Extensión del fichero JAR que contiene todos los ficheros necesarios para realizar el despliegue de un proceso de negocio WS-BPEL.

Desplegar En el ámbito de los SOA (Service Oriented Architecture), *desplegar* un proceso se refiere a ponerlo en funcionamiento en un motor de ejecución, escuchando en un determinado puerto.

Mockup Un *mockup* es un servicio sustituto de un servicio real en una simulación, que implementa un comportamiento predefinido, y que suele emplearse en pruebas para reemplazar a un servicio costoso, al que no se puede acceder, o también para comprobar si la salida de un programa para un caso de prueba es la esperada. No tienen lógica interna, limitándose a responder con mensajes predefinidos, o "fallando" si así se especifica.

PDD Fichero empleado en ActiveBPEL que contiene la información necesaria para desplegar un proceso WS-BPEL.

Proceso de negocio síncrono Proceso de negocio en el que el cliente queda a la espera de su respuesta.

Proceso de negocio asíncrono Proceso de negocio en el que el cliente continúa su ejecución tras enviar la petición, y el proceso servidor le enviará posteriormente una notificación del resultado de la petición.

1.6.3. Otras tecnologías

Para llevar a cabo todos los objetivos de este PFC es necesaria la utilización de diversos lenguajes y tecnologías:

- WSDL (Web Services Description Language): para describir WS.
- SOAP (Simple Object Access Protocol): para intercambiar mensajes en un entorno distribuido.
- WS-BPEL: para realizar la composición de WS.
- XSLT: para transformar documentos XML en otros, utilizando reglas de plantilla.

- XPath (XML Path Language): para construir expresiones que recorran y procesen documentos XML.
- XML Schema: para describir la estructura y las restricciones de los contenidos de los documentos XML.
- JUnit: para hacer pruebas unitarias, permite realizar la ejecución de clases Java de manera controlada y evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.
- BPELUnit: framework para realizar pruebas de unidad de composiciones WS-BPEL.

Servicios Web

Los WS son una implementación del paradigma de Arquitectura Orientada a Servicios, SOA, que define un nuevo estilo de arquitectura abstracta que no está limitado a ninguna tecnología específica en particular. La arquitectura SOA define un paradigma en el que un "servicio" es el elemento atómico. Un servicio Web es definido por el W3C (World Wide Web Consortium) como:

> "Un sistema software diseñado para ofrecer interacción máquina-a-máquina sobre una red. Tiene una interfaz descrita en un formato procesable por la máquina: WSDL. Otros sistemas interactúan con el servicio Web en una forma prescrita por su descripción utilizando mensajes SOAP, normalmente transmitidos usando HTTP (HyperText Transfer Protocol) con una serialización XML y con otros estándares Web relacionados." (W3C 2004a)

Los WS dependen en gran medida de los estándares abiertos basados en XML, proporcionando así una integración perfecta y la interoperabilidad entre aplicaciones software a través la red.

La Figura 1.2 representa los roles básicos que se toman en una infraestructura SOA. Como se puede ver, se trata de tres actores principales: un proveedor de servicios, un agente de servicio y un consumidor de servicios.

Proveedor de servicios Crea un servicio Web y publica su interfaz y la información de acceso al registro agente del servicio. Cada proveedor tiene que decidir qué servicios exponer, cómo hacer compensaciones entre seguridad y fácil disponibilidad, cómo fijar el precio del servicio... El proveedor también tiene que clasificar el servicio en una categoría de un agente de servicio dado y qué tipo de "acuerdos" de socios de negocio son necesarios para utilizar el servicio expuesto.

Agente de servicios También conocido como "registro de servicios", es el responsable de construir la interfaz de servicio Web y la implementación accesible a cualquier solicitante potencial de servicio. El implementador del agente decide sobre el ámbito del agente. Los agentes públicos están disponibles a través de Internet,

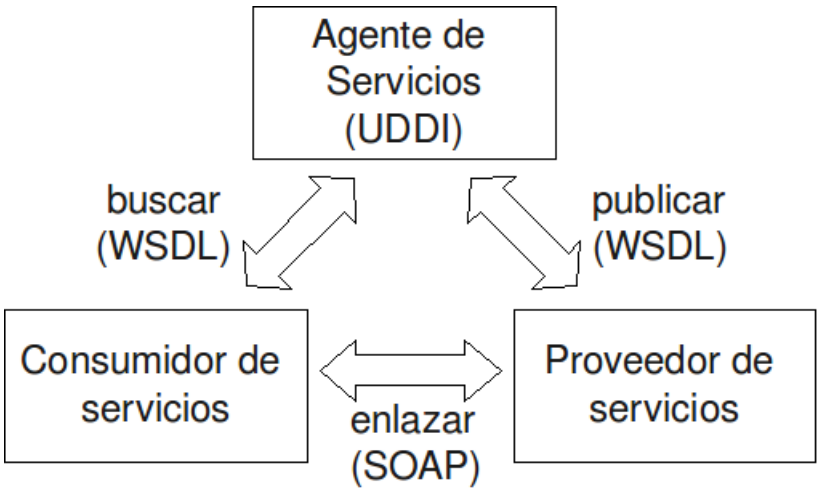

Figura 1.2: Arquitectura de servicios Web con sus especificaciones UDDI, SOAP y WSDL

mientras que los agentes privados sólo son accesibles a una audiencia limitada, por ejemplo, a los usuarios de la intranet de una empresa. Por otra parte, se debe decidir qué cantidad de información se va a ofrecer. Algunos agentes se especializan en algunos listados. Otros ofrecen un alto nivel de confianza en los servicios listados. Algunas cubren una gran cantidad de servicios y otros se centran en una industria. También hay agentes que catalogan a otros agentes. La especificación UDDI (Universal Description Discovery and Integration) [44] de OASIS define una forma de publicar y descubrir información sobre los WS. Otras tecnologías de agente de servicios incluyen, por ejemplo, ebXML (Electronic Business using eXtensible Markup Language) [42] de OASIS.

Consumidor de servicios El solicitante del servicio o el cliente de servicio Web localiza entradas en el registro agente realizando diferentes búsquedas y se une al proveedor de servicios con el fin de invocar uno de sus WS. Los WS se construyen a partir de varios estándares, pero cinco de ellos se consideran fundamentales. Algunos ya existían y se han utilizado para crear WS, como XML y HTTP, y otros se han creado específicamente para los WS, como WSDL, SOAP y UDDI.

WSDL

WSDL [54] es un lenguaje basado en XML utilizado para describir la interfaz de WS: qué pueden hacer, dónde se encuentran y qué tipo de datos esperan y en qué formato. WSDL informa sobre cómo se puede interactuar con el servicio Web pero no dice nada acerca de cómo trabaja el servicio Web. Esto da lugar a sólo tener que describir el servicio Web, nada con respecto a su implementación, que ayuda a reducir los problemas de compatibilidad entre dos sercicios Web que ofrecen la misma función, pero utilizan diferentes implementaciones.

WSDL en un lenguaje híbrido creado por IBM y Microsoft combinando sus lenguajes propietarios de descripción de servicios. WSDL ofrece la posibilidad de describir llamadas a procedimientos remotos (RPC), usar cualquier tipo de protocolo y formato de datos, y usar mensajes.

WSDL se basa en siete principios: extensibilidad, soporte para sistemas de tipo múltiples, uso de mensajes y RPC, separación de "qué" del "cómo" y "dónde", soporte de protocolos y transportes múltiples, no permite especificar el orden en el que las operaciones deben ser llamadas y tampoco describe ninguna semántica de servicios.

WSDL define los dos niveles de un servicio Web: un nivel abstracto y un nivel concreto.

En un nivel abstracto, WSDL define un servicio Web en términos del mensaje que puede enviar y recibir, normalmente utilizando un formato de datos basado en XML Schema. Define *operations* que asocian un patrón de intercambio de mensaje, MEP (Message Exchange Pattern), que puede diferir dependiendo del protocolo de mensaje concreto que se use. También define los *portType* (*interfaces* en WSDL 2.0) que básicamente agrupan las operaciones sin ningún compromiso con un protocolo de transporte.

En el nivel concreto, WSDL define un *binding* que especifica los detalles de transporte y de formato para una o más interfaces. Un *port* (*endpoint* en WSDL 2.0) asocia una dirección de red con un *binding* y, además, una URL (Uniform Resource Locator) a cada servicio. Finalmente, un *service* agrupa los *port* implementando una interfaz común.

A continuación, definimos la estructura general de WSDL 1.1:

Types Contiene las definiciones de los tipo de datos, normalmente utilizando un sistema de tipo XSD (XML Schema Definition). Es decir, es donde se definen los tipos de datos de los mensajes de los WS.

Message Define el formato de los mensajes que se intercambian entre el servicio Web y el cliente del servicio Web. Está compuesto de una o más partes lógicas. Cada parte puede referirse a un elemento XSD y/o a tipos XSD simples o complejos.

PortType Puede definirse como un contenedor de una o más operaciones abstractas, que básicamente definen la signatura del método.

Operation Una operación normalmente consiste, como en cada lenguaje de programación, en un parámetro o parámetros de entrada, un valor de salida y un elemento de fallo opcional.

- **Input** Define la entrada de la operación, que es un mensaje específico.
- **Output** Define el valor de salida de la operación. El valor es también expresado como un mensaje.
- **Fault** Puede considerarse como un mensaje de excepción que se devuelve desde el servicio Web al que ha llamado, en caso de que se capture un error. El error está dentro de un mensaje.

Una operación WSDL 1.1 soporta 4 tipos de patrones de intercambio de mensajes:

One-way El *endpoint*, por ejemplo, el servicio Web, recibe un mensaje.

Request-response El *endpoint* recibe un mensaje y devuelve un mensaje correlacionado al que ha llamado.

Solicit-response El *endpoint* envía un mensaje y recibe una respuesta correlacionada.

Notification El *endpoint* envía un mensaje.

Binding Contiene los detalles de cómo los elementos abstractos de un *PortType* se limitan a un concreto conjunto de elementos, por ejemplo, en el *binding* se puede definir qué protocolo de comunicación usa el servicio, el formato de los datos, etc. El protocolo de comunicación más utilizado es HTTP y el de representación de mensajes entre las partes es SOAP.

Service Un servicio es un contenedor que agrupa los elementos *port*.

Port Especifica la dirección, en concreto la URL, para un *binding*.

UDDI

UDDI [44] es un estándar OASIS, que define un registro para los servicios Web. Básicamente, define los mecanismos para:

- Descripción de los servicios
- Descubrir las entidades de negocio
- Integración de servicios de negocio a través de Internet

Es un framework independiente de la plataforma, que actúa como una agenda telefónica, pero especializado para los WS. Los proveedores de servicios utilizan un registro UDDI para publicar los servicios que ofrecen, clasificándolos en una categoría, y describiéndolos técnicamente utilizando sus interfaces WSDL.

SOAP

SOAP [20] es un protocolo de intercambio de mensajes utilizado por los WS. Es un protocolo simple y extensible basado en XML utilizado para el intercambio de mensajes de información estructurada en un entorno distribuido. Su principal ventaja sobre otros protocolos como IIOP (Internet Inter-ORB Protocol) o RMI (Java Remote Method Invocation) es que fue construido para trabajar con HTTP, el protocolo de transporte más utilizado en Internet, que permite la comunicación detrás de *proxies* y cortafuegos.

SOAP fue creado en 1999 por Microsoft, Developmentor y Userland, y significa Simple Object Access Protocol. Actualmente, está estandarizado por el W3C.

SOAP ofrece:

- Formato de Mensaje Estandarizado: SOAP especifica cómo la información es empaquetada en un documento XML estandarizado (un mensaje SOAP) para transferirse en una comunicación.
- Convenciones de Correspondencia: La especificación SOAP contiene un conjunto de convenciones para relacionar datos de aplicaciones en los mensajes SOAP, por ejemplo, especificando una llamada de procedimiento remoto.
- Modelo de Procesamiento: este modelo define los roles de los que reciben y envían mensajes SOAP, especificando qué partes de un mensaje deben ser procesadas por un rol.
- Enlaces de Protocolos de Red: SOAP especifica enlaces, que describen cómo los mensajes SOAP son transportados sobre HTTP, SMTP y otros protocolos de transporte.

SOAP define un conjunto de elementos para representar un mensaje:

Envelope Es el elemento raíz de cada mensaje SOAP.

Header Es un elemento opcional que puede contener información adicional, como información relativa a la seguridad, información de enrutamiento, etc.

Body Contiene la información principal del mensaje, que representa el cuerpo del mensaje.

Fault Representa un bloque específico contenido en el **body** que define un error que ocurrió durante la interacción.

WS-BPEL 2.0

WS-BPEL [43] es un lenguaje basado en XML que permite especificar el comportamiento de un proceso de negocio basado en interacciones con WS (véase Anexo A).

La estructura de un proceso WS-BPEL se divide en cuatro secciones:

1. Definición de relaciones con los socios externos, que son el cliente que utiliza el proceso de negocio y los WS a los que llama el proceso.
2. Definición de las variables que emplea el proceso.
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso. Pueden definirse manejadores de fallos, que indican las acciones a realizar en caso de producirse un fallo interno o en un WS al que se llama. También se definen los manejadores de eventos, que especifican las acciones a realizar en caso de que el proceso reciba una petición durante su flujo normal de ejecución.

4. Descripción del comportamiento del proceso de negocio; esto se logra a través de las actividades que proporciona el lenguaje.

Todos los elementos definidos anteriormente son globales si se declaran dentro del proceso. Sin embargo, también existe la posibilidad de declararlos de forma local mediante el contenedor `scope`, que permite dividir el proceso de negocio en diferentes ámbitos.

Los principales elementos constructivos de un proceso WS-BPEL son las actividades, que pueden ser de dos tipos: básicas y estructuradas. Las actividades básicas son las que realizan una determinada labor (recepción de un mensaje, manipulación de datos, etc.). Las actividades estructuradas pueden contener otras actividades y definen la lógica de negocio.

A las actividades pueden asociarse un conjunto de atributos y un conjunto de contenedores. Estos últimos pueden incluir diferentes elementos, que a su vez pueden tener atributos asociados. Veamos un ejemplo:

```
<flow>  ← Actividad estructurada
  <links>  ← Contenedor
    <link name="comprobarVuelo-A-reservarVuelo" ← Atributo
    /> ← Elemento
  </links>
  <invoke name="comprobarVuelo" ... > ← Actividad básica
    <sources>  ← Contenedor
      <source linkName="comprobarVuelo-A-reservarVuelo" ← Atributo
      /> ← Elemento
    </sources>
  </invoke>
  <invoke name="comprobarHotel" ... />
  <invoke name="comprobarAlquilerCoche" ... />
  <invoke name="reservarVuelo" ...>
    <targets>  ← Contenedor
      <target linkName="comprobarVuelo-A-reservarVuelo" /> ← Elemento
    </targets>
  </invoke>
</flow>
```

Además, WS-BPEL permite realizar acciones en paralelo y de forma sincronizada. Un ejemplo de ello es la actividad `flow`, mediante la cual se puede especificar un conjunto de actividades que se van a realizar concurrentemente, así como las condiciones de sincronización entre ellas. En el ejemplo anterior, la actividad `flow` invoca tres WS en paralelo, `comprobarVuelo`, `comprobarHotel`, y `comprobarAlquilerCoche`, además hay otra actividad WS, `reservarVuelo`, que sólo será invocada si se completa `comprobarVuelo`. Esta sincronización entre actividades se consigue estableciendo un enlace (link), por lo que la actividad objetivo del enlace sólo será ejecutada si la actividad fuente del enlace se ha completado.

XSLT 2.0

XSLT [27] se trata de un lenguaje XML estandarizado por el W3C que define hojas de estilo capaces de transformar una entrada XML a otros formatos (véase Anexo B).

El formato de salida no ha de ser necesariamente XML también: se podría generar texto arbitrario, o un documento HTML (Hyper Text Markup Language).

El uso de hojas XSLT permite definir las transformaciones de forma declarativa, simplificando mucho la tarea de transformación frente a otros métodos, como el uso del API DOM (Document Object Model), o el procesado mediante SAX (Simple API for XML).

XPath 2.0

XPath 2.0 [29] es un estándar del W3C relacionado con XML que además de poderse usar independientemente, es parte fundamental de otras tecnologías XML, como XSLT, o XQuery, un lenguaje especializado para consultas sobre datos XML de cualquier fuente (incluso una base de datos).

Su propósito es permitir identificar fácilmente conjuntos de nodos de un documento XML, siguiendo un modelo de datos similar al del estándar DOM, con ligeras diferencias.

La sintaxis de una expresión XPath es declarativa, usando condiciones que deben cumplir los nodos resultado de dicha expresión. Se pueden filtrar nodos en función de su tipo, nombre (en caso de ser elemento), atributos, descendientes, etc.

XML Schema

El propósito del estándar XML Schema [35] es definir la estructura de los documentos XML que estén asignados a tal esquema y los tipos de datos válidos para cada elemento y atributo.

Al restringir el contenido de los ficheros XML es posible intercambiar información entre aplicaciones con gran seguridad. Disminuye el trabajo de comprobar la estructura de los ficheros y el tipo de los datos.

Los tipos de datos tienen en XML Schema la función de las clases en la Programación Orientada a Objetos. El usuario puede construir tipos de datos a partir de tipos predefinidos, agrupando elementos y atributos de una determinada forma y con mecanismos de extensión parecidos a la herencia. Los tipos de datos se clasifican en función de los elementos y atributos que contienen.

Los tipos de datos en XML Schema pueden ser simples o complejos:

- Tipos simples: no tienen elementos hijos ni atributos.
- Tipos complejos: tienen elementos hijos y/o atributos.

XML Schema incluye el uso de *namespaces*. Los "espacios de nombres" permiten definir elementos con igual nombre dentro del mismo contexto, siempre y cuando se anteponga un prefijo al nombre del elemento, además, evitan confusiones en la reutilización de código.

JUnit

JUnit [26] es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.

JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que devolvió el método durante la ejecución, JUnit informará de que el caso de prueba ha fracasado.

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requisitos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

BPELUnit

BPELUnit [51] es un marco de pruebas unitarias para hacer pruebas de composiciones WS-BPEL repetibles, de caja blanca y automáticas, que pone énfasis en los siguientes puntos:

- Soporte para elementos específicos de WS-BPEL como manejo de actividades paralelas, compensación y mensajes asíncronos.
- Fácil uso para el desarrollador.
- Una vista realista de la ejecución del proceso WS-BPEL.

Estos puntos se llevan a cabo mediante el empleo de un formato de especificación de datos a nivel WS-BPEL, un lenguaje de dominio específico para escribir la lógica de las pruebas, un formato para el conjunto de casos de prueba que permita la reutilización de éstos, y mediante el despliegue y ejecución del proceso WS-BPEL.

Las características principales de BPELUnit son las siguientes:

- Utiliza datos literales XML como formato de especificación de datos, en lugar de sobres (*envelopes*) SOAP completos, que hace que estén al mismo nivel tanto las pruebas como las composiciones WS-BPEL.
- Permite la especificación de hilos paralelos de actividades secuenciales necesarios para la simulación de varios socios de un PUT a la vez, cada uno cumpliendo un cierto protocolo de negocio.

2 Estado del arte

2.1. Antecedentes

Se han publicado diversos trabajos que tratan sobre la definición de operadores de mutación para un determinado lenguaje de programación.

En concreto, Agrawal y col. [2] han definido un conjunto de 77 operadores de mutación para el lenguaje C. Estos operadores modelan diversos errores que suelen cometer los desarrolladores al programar en C y proporcionan cobertura parcial de caminos (véase §C.1).

King y Offutt [31] han construido la herramienta de análisis de mutaciones Mothra para el lenguaje Fortran, que utiliza un conjunto de 22 operadores de mutación (véase §C.8).

Offut y col. [45] han definido un total de 65 operadores de mutación para el lenguaje Ada, que han sido clasificados en cinco categorías: operadores de reemplazamiento de operandos, operadores de sentencias, operadores de expresiones, operadores de cobertura y operadores de tareas (véase §C.6).

Chan y col. [6] definen 7 operadores de mutación para el lenguaje SQL, clasificados en una única categoría: operadores de reemplazamiento. Tuya y col. [53] han definido un conjunto de operadores de mutación para SQL que ha sido integrado en una herramienta para la generación de mutantes. Shahriar y Zulkernine [48, 49] definen 9 operadores clasificados en operadores de condiciones `WHERE` y operadores de llamadas a "funciones predefinidas" de la base de datos. Derezińska [11] define 22 operadores de mutación para este lenguaje clasificados en cuatro categorías: cláusulas SQL, reemplazo de operador, `NULL` y reemplazo de identificador (véase §C.5).

Zhang [55] ha definido 24 operadores de mutación para el lenguaje C++, que los ha clasificado en tres categorías: cambio de tipo de operadores, inserción de operadores y cambio de operadores aritméticos (véase §C.3).

Derezińska [9, 10] ha definido 40 operadores de mutación para el lenguaje C#, que los ha clasificado en diez categorías: errores típicos de los programadores, ocultación de información, herencia, polimorfismo y construcción dinámica, sobrecarga de métodos, características de lenguajes orientado a objetos, excepciones, delegación de métodos, propiedad de reemplazo y otros operadores (véase §C.2).

Mansour y Houri [36] han propuesto un conjunto de operadores de mutación para evaluar la calidad de las pruebas realizadas a aplicaciones Web desarrolladas en el entorno .NET (véase §C.7).

El lenguaje Java también ha sido objeto de estudio para definir sus operadores de mutación en diversos trabajos. Kim y col. [30] han aplicado la técnica HAZOP (Hazard and Operability Studies) a la definición de la sintaxis de Java para identificar desviaciones de las construcciones del lenguaje y, así, definir un conjunto de operadores de mutación para este lenguaje. Alexander y col. [3] han definido un conjunto de operadores para introducir mutaciones en los objetos de Java; estos operadores son utilizados por la herramienta OME (Object Mutation Engine) para generar mutantes. Ma y col. [33] definen 26 operadores de mutación para el lenguaje Java basándose en una lista exhaustiva de los fallos que pueden darse en los programas orientados a objetos. Estos operadores han sido integrados en la herramienta *MuJava*. Bradbury y col. [5] han definido un total de 24 operadores específicos para el comportamiento concurrente de Java (J2SE 5.0). Smith y Williams [50] han definido unos operadores clasificados en operadores a nivel de método y operadores de clase. Ji y col. [24] han definido 5 operadores para la captura de excepciones. Madeyski y Radyk [34] han definido un conjunto de operadores clasificados en cuatro categorías: tradicionales, propios del lenguaje, orientados a objetos y operadores de clase (véase §C.4).

2.2. Operadores de mutación para WS-BPEL 2.0

Esta sección describe los operadores de mutación que se han definido en [16, 17] para el lenguaje WS-BPEL 2.0.

Estos operadores se clasifican en cuatro categorías, de acuerdo con el tipo de elemento sintáctico de WS-BPEL con el que se relacionan. Las categorías se identifican con una letra mayúscula y son las siguientes:

- Mutación de identificadores (I)
- Mutación de expresiones (E)
- Mutación de actividades (A)
- Mutación de condiciones excepcionales y eventos (X)

Dentro de cada categoría se definen varios operadores de mutación que se identifican mediante tres letras mayúsculas: la primera de ellas coincide con la que identifica la categoría a la que pertenece el operador, mientras que las dos últimas identifican al operador dentro de la categoría.

Los operadores definidos modelan fallos que podrían cometerse al generar una composición de servicios con WS-BPEL, habiéndose tenido en cuenta que normalmente no se suele escribir código WS-BPEL de forma directa, sino que se utilizan herramientas

gráficas que ayudan a realizar la composición. Como consecuencia, muchos de los fallos que pueden aparecer en programas escritos en otros lenguajes, debidos a errores al teclear código, no aparecerán en WS-BPEL.

Agrawal y col. especifican en [2] que es difícil determinar el dominio para cualquier operador de mutación que pueda operar sobre una declaración, por lo que no introducen mutaciones en ellas. En este sentido, hay ciertos aspectos de WS-BPEL que no se van a mutar: las definiciones de variables y las relaciones con los socios externos.

2.2.1. Operadores de Mutación de Identificadores

Un error común que puede cometerse al escribir un programa es cambiar el identificador de una variable por el de otra. Para otros lenguajes se han definido operadores de este tipo, aunque se suele establecer un operador por cada tipo de variable [2, 31, 45].

ISV Sustituye el identificador de una variable por el de otra del mismo tipo; esta sustitución sólo se realizará si ambas pertenecen al mismo ámbito.

A continuación se muestra un ejemplo de la aplicación del operador **ISV**. En la composición se llama a dos WS de dos compañías aéreas, devolviendo cada una de ellas su respuesta en una variable diferente, `respuestaVueloIb` y `respuestaVueloAE`. El operador **ISV** cambia la variable `respuestaVueloIb` por `respuestaVueloAE`.

Código original:

```
<flow>
  <sequence>
    <invoke partnerLink="Iberia" ...
       inputVariable="DetalleVuelo" />
    <receive partnerLink="Iberia" ...
       variable="respuestaVueloIb"/>
  </sequence>
</flow>
```

Mutante generado por ISV:

```
<flow>
  <sequence>
    <invoke partnerLink="Iberia" ...
       inputVariable="DetalleVuelo" />
    <receive partnerLink="Iberia" ...
       variable="respuestaVueloAE"/>
  </sequence>
</flow>
```

2.2.2. Operadores de Mutación de Expresiones

Para definir los operadores de mutación de expresiones se han tenido en cuenta operadores similares definidos para otros lenguajes [31, 45, 53]. Se han definido operadores para todos los tipos de expresiones que podemos encontrar en WS-BPEL, pero sólo se han considerado aquellos operadores de mutación que sustituyen un operador por otro del mismo tipo, ya que éstos son los fallos que pueden cometerse con mayor frecuencia. Así tenemos:

EAA Sustituye, en una expresión aritmética, un operador aritmético (`+`, `-`, `*`, `div`, `mod`) por otro del mismo tipo.

EEU Elimina el operador – unario de cualquier expresión en la que aparezca. No se considera el añadirlo porque modelaría un fallo poco frecuente.

ERR Sustituye en una expresión relacional un operador relacional (`<`, `>`, `<=`, `>=`, `=`, `!=`) por otro del mismo tipo. A partir de ahora los operadores `<`, `>`, `<=` y `>=` aparecerán en los ejemplos como se escriben en el código WS-BPEL: `<`, `>`, `<=` y `>=`, respectivamente.

ELL Sustituye en una expresión lógica un operador lógico (`and`, `or`) por otro del mismo tipo.

ECC Sustituye en una expresión de camino un operador de camino (`/`, `//`) por otro del mismo tipo.

Dado que todos los operadores de mutación anteriores son muy similares, diferenciándose sólo en el tipo de operador sobre el que actúan, daremos un único ejemplo para todos ellos:

Código original:

```
$peticion.cantidad &lt; 10000
```

Mutante generado por ERR:

```
$peticion.cantidad &gt; 10000
```

Otro operador de mutación dentro de esta categoría es **ECN**, que tiene como dominio las constantes numéricas que aparecen en el programa. Este operador modela los errores que se pueden cometer al introducir constantes numéricas modificándolas de varias formas: incrementa o decrementa su valor en una unidad, o bien añade o elimina alguna cifra. Algunos ejemplos de su aplicación son:

Código original:

```
$peticion.cantidad &gt; 10000
```

Mutante 1 generado por ECN:

```
$peticion.cantidad &gt; 10001
```

Mutante 2 generado por ECN:

```
$peticion.cantidad &gt; 1000
```

El operador **EMD** modifica una expresión de duración de dos formas: la sustituye por 0, lo que implica que la condición se cumple inmediatamente, y por la mitad del valor expresado inicialmente; esto tiene como objetivo verificar si el margen de seguridad especificado por la duración es adecuado. Este operador es aplicable a la actividad `wait`, y al elemento `onAlarm` de la actividad `pick` y de los manejadores de eventos.

Código original:

```
<onAlarm>
  <for> 'P0DT12H' </for>
</onAlarm>
```

Mutante 1 generado por EMD:

```
<onAlarm>
  <for> 'P0DT6H' </for>
</onAlarm>
```

Mutante 2 generado por EMD:

```
<onAlarm>
  <for> 'P0DT0H' </for>
</onAlarm>
```

El operador **EMF** modifica una expresión de fecha límite. Se puede aplicar a la actividad `wait` y al elemento `onAlarm` de la actividad `pick` y de los manejadores de eventos. Las modificaciones que introduce son equivalentes a las del operador **EMD**.

2.2.3. Operadores de Mutación de Actividades

Se incluyen dentro de esta categoría los operadores de mutación de WS-BPEL que afectan a las actividades que proporciona el lenguaje. Algunos de estos operadores modelan el fallo que puede cometerse al elegir una actividad que no es la más adecuada para las acciones que se deben realizar, sustituyendo una actividad por otra. Los demás modelan la elección de un valor incorrecto para los atributos de las actividades, sustituyendo para ello su valor actual por otro valor válido. Estos operadores se clasifican en dos tipos, los relacionados con la concurrencia y los no concurrentes.

Relacionados con la Concurrencia

Las actividades `receive` y `pick` permiten recibir mensajes. Ambas pueden especificar un atributo denominado `createInstance` que, si está activado, creará una nueva instancia del proceso cuando llegue un nuevo mensaje y, si no lo está, hará que el nuevo mensaje sea consumido por una de las instancias que ya existen.

El operador **ACI** cambia el atributo `createInstance` de "`yes`" a "`no`" sólo en el caso de que el proceso tenga más de una actividad con dicho atributo a "`yes`", ya que en caso contrario el mutante no sería válido. El cambio contrario no se contempla debido a que las restricciones que impone el estándar sobre la correlación entre el mensaje y la instancia del proceso a la que va dirigido, harían también que los mutantes generados no fueran válidos.

Código original:

```
<receive ... createInstance="yes">
```

Mutante generado por ACI:

```
<receive ... createInstance="no">
```

WS-BPEL proporciona una actividad `forEach` que permite ejecutar el conjunto de actividades asociadas un número fijo de veces. Existen dos variantes de `forEach` que se diferencian en el modo en que se ejecutan las iteraciones, secuencial o paralelamente.

El operador **AFP** cambia el carácter secuencial de una actividad `forEach` por paralelo, modificando el valor del atributo `parallel` de "no" a "yes". No se propone el cambio contrario porque éste no introduciría un error que se pudiera reflejar en los resultados del proceso, sino que únicamente se tardaría más tiempo en completar las actividades, por lo que se generaría un mutante equivalente.

Código original:

```
<forEach parallel="no" ... >
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>5</finalCounterValue>
  <scope>
    <invoke name="comprobarDisponib" ... />
  </scope>
</forEach>
```

Mutante generado por AFP:

```
<forEach parallel="yes" ... >
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>5</finalCounterValue>
  <scope>
    <invoke name="comprobarDisponib" ... />
  </scope>
</forEach>
```

La actividad `sequence` obliga a ejecutar secuencialmente las actividades que contiene, mientras que la actividad `flow` las ejecuta en paralelo. El operador **ASF** cambia una actividad `sequence` por `flow`; el cambio contrario no se realiza porque normalmente la estructura de una actividad `flow` es más compleja que la de `sequence`, por lo que es difícil que se cometa dicho error. Por otro lado, si la actividad `flow` tuviera una estructura similar a la de `sequence`, dicha mutación generaría un mutante equivalente.

Código original:

```
<sequence name="MensajePedido">
  <receive name="recibePedido" ... />
  <invoke name="compruebaPago" ... />
</sequence>
```

Mutante generado por ASF:

```
<flow name="MensajePedido">
  <receive name="recibePedido" ... />
  <invoke name="compruebaPago" ... />
</flow>
```

Para las ejecuciones en paralelo, WS-BPEL proporciona un mecanismo para proteger el acceso concurrente a datos globales, se trata del atributo `isolated` de un `scope`. Cuando el valor del atributo es "`yes`" se protege el acceso a variables compartidas,

no permitiendo que las actividades de un `scope` puedan acceder a ellas mientras las de otro, en el que se utilizan las mismas variables, no hayan terminado. El operador **AIS** pone el atributo `isolated` de un `scope` en el que se manipulan variables compartidas a "no", por lo que no se protegería el acceso a estas variables, produciéndose resultados impredecibles. No se plantea el cambio contrario porque produciría un mutante equivalente.

Código original:

```
<flow>
  <scope name="S1" isolated="yes">
    <sequence>
       ...
       <invoke ... outputVariable="global"/>
       ...
    </sequence>
  </scope>
  <scope name="S2" isolated="yes">
    <sequence>
       ...
       <assign>
           <copy>
             <from> ... </from>
             <to variable="global" />
           </copy>
       </assign>
       ...
    </sequence>
  </scope>
</flow>
```

Mutante generado por AIS:

```
<flow>
  <scope name="S1" isolated="no">
    <sequence>
       ...
       <invoke ... outputVariable="global"/>
       ...
    </sequence>
  </scope>
  <scope name="S2" isolated="yes">
    <sequence>
       ...
```

```
        <assign>
            <copy>
              <from> ... </from>
              <to variable="global" />
            </copy>
        </assign>
        ...
    </sequence>
  </scope>
</flow>
```

No Concurrentes

La actividad `if` permite definir las actividades que se van a ejecutar en función del valor verdadero o falso de un conjunto de condiciones. Esta actividad permite especificar opcionalmente uno o varios elementos `elseif` y un elemento `else`. El operador **AIE** elimina un elemento `elseif` o el elemento `else` de una actividad `if`, modelando el olvido de una rama de esta actividad. Un ejemplo de uno de los mutantes generados es:

Código original:

```
<if name="PedidoMayor5000euros">
  <condition>$pedido &gt; 5000</condition>
  <invoke name="calculaDiezPCDesc" ... />
  <elseif>
    <condition>$pedido &gt; 2500</condition>
    <invoke name="calculaCincoPCDesc" ... />
  </elseif>
  <else>
    <reply name="enviarInfoNoDescuento" ... />
  </else>
</if>
```

Mutante generado por AIE:

```
<if name="PedidoMayor5000euros">
  <condition>$pedido &gt; 5000</condition>
  <invoke name="calculaDiezPCDesc" ... />


  <else>
    <reply name="enviarInfoNoDescuento" ... />
```

```
  </else>
</if>
```

Aparte de `forEach`, WS-BPEL proporciona otras dos actividades repetitivas, `while` y `repeatUntil`. El operador **AWR** cambia una actividad `while` por otra `repeatUntil` y viceversa, modelando el error cometido al no elegir el tipo de actividad repetitiva adecuada.

Código original:

```
<while>

  <condition>
    $iteraciones &gt; 3
  </condition>
  <invoke name="incrementaContIter" ... />
</while>
```

Código generado por AWR:

```
<repeatUntil>
  <invoke name="incrementaContIter" ... />
  <condition>
    $iteraciones &gt; 3
  </condition>

</repeatUntil>
```

A todas las actividades que proporciona WS-BPEL pueden asociarse los contenedores opcionales `sources` y `targets`, los cuales contienen elementos `source` y `target`, respectivamente. Estos elementos se utilizan para lograr la sincronización de actividades. Se puede especificar para el contenedor `targets` un elemento denominado `joinCondition`, cuyo valor es una expresión booleana. Cuando no se especifica este elemento, la condición que se considera es la disyunción del estado de todos los `target` de esta actividad. El operador **AJC** elimina el elemento `joinCondition`. No se considera el cambio contrario porque es un error poco probable.

Código original:

```
<targets>
  <target linkName="link1" />
  <target linkName="link2" />
  <joinCondition>
    $link1 and $link2
```

```
  </joinCondition>
</targets>
```

Código generado por AJC:

```
<targets>
  <target linkName="link1" />
  <target linkName="link2" />


</targets>
```

El operador **ASI** intercambia el orden de dos actividades dentro de una actividad `sequence`. Si las actividades intercambiadas no tienen una dependencia de datos entre ellas, se producirán mutantes equivalentes. Un ejemplo de uno de los mutantes generados es:

Código original:

```
<sequence name="MensajePedido">
  <receive name="recibePedido" ... />
  <invoke name="compruebaPago" ... />
  <invoke name="servicioEnvio" ... />
  <reply name="enviaConfirm" ... />
</sequence>
```

Mutante generado por ASI:

```
<sequence name="MensajePedido">
  <invoke name="compruebaPago" ... />
  <receive name="recibePedido" ... />
  <invoke name="servicioEnvio" ... />
  <reply name="enviaConfirm" ... />
</sequence>
```

Una de las actividades de recepción de mensajes que proporciona WS-BPEL es `pick`, que mediante los elementos `onMessage` permite especificar las actividades a realizar cuando el proceso de negocio recibe un mensaje determinado. También nos permite definir, mediante el elemento `onAlarm`, las acciones a llevar a cabo si no se recibiera ningún mensaje en un tiempo determinado.

El operador **APM** elimina un evento `onMessage` de una actividad `pick`, siempre que haya más de uno, modelando el error que se cometería al olvidar la posible recepción de un mensaje.

Código original:

```
<pick>
  <onMessage partnerLink="comprador"
    operation="nuevoArticulo" ... > ...
  </onMessage>
  <onMessage partnerLink="comprador"
    operation="pedidoCompleto" ... > ...
  </onMessage>
  <onAlarm>
    <for> 'P3DT10H' </for> ...
  </onAlarm>
</pick>
```

Mutante generado por APM:

```
<pick>
  <onMessage partnerLink="comprador"
    operation="nuevoArticulo" ... > ...
  </onMessage>


  <onAlarm>
    <for> 'P3DT10H' </for> ...
  </onAlarm>
</pick>
```

El operador **APA** elimina el elemento `onAlarm` que puede aparecer en una actividad `pick`, modelando el error que se produciría al olvidar introducir este elemento. Este operador también se puede aplicar al elemento `onAlarm` de un manejador de eventos. Si aplicamos el operador sobre el mismo ejemplo que el anterior, el mutante que se obtendría sería:

Código original:

```
<pick>
  <onMessage partnerLink="comprador"
    operation="nuevoArticulo" ... > ...
  </onMessage>
  <onMessage partnerLink="comprador"
    operation="pedidoCompleto" ... > ...
  </onMessage>
  <onAlarm>
    <for> 'P3DT10H' </for> ...
  </onAlarm>
</pick>
```

Código generado por APA:

```
<pick>
  <onMessage partnerLink="comprador"
    operation="nuevoArticulo" ... > ...
  </onMessage>
  <onMessage partnerLink="comprador"
    operation="pedidoCompleto" ... > ...
  </onMessage>


</pick>
```

2.2.4. Operadores de Mutación Relacionados con las Condiciones Excepcionales y Eventos

Los operadores definidos dentro de esta categoría están relacionados con los distintos tipos de manejadores que proporciona WS-BPEL: de fallos, de eventos, de compensación y de terminación.

Los manejadores de fallos permiten especificar mediante los elementos `catch` las actividades a llevar a cabo en caso de que se produzca un fallo determinado, y mediante el elemento `catchAll` las relacionadas con cualquier otro fallo no especificado anteriormente. El operador **XMF** elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos. Por tanto, modela el olvido por parte del programador de incluir un manejador específico para un fallo determinado, o bien el de un manejador por omisión.

Código original:

```
<faultHandlers>
  <catch faultName="LibroNoDisponible" ... >
    ...
  </catch>
  <catchAll> ... </catchAll>
</faultHandlers>
```

Mutante generado por XMF:

```
<faultHandlers>
  <catch faultName="LibroNoDisponible" ... >
    ...
  </catch>
```

```
</faultHandlers>
```

A veces, cuando se produce un fallo durante la ejecución de un proceso, es necesario deshacer el trabajo que ya ha sido hecho; para ello se dispone del manejador de compensación. Por otro lado, dentro de un `scope` se puede definir un manejador de terminación, que indica las actividades a realizar si se debe terminar la ejecución de dicho `scope`.

Un `scope` termina su ejecución con éxito o sin éxito dependiendo de los siguientes casos (véase §A.7):

- Finalización normal: si la actividad principal finaliza sin lanzar un fallo y no se detectan actividades de mensajes de entrada, entonces todos los manejadores de eventos se deshabilitan y el manejador de compensación se instala. El `scope` finaliza con éxito.
- Fallo interno: si se lanza un fallo dentro del `scope`, entonces se terminan las otras actividades en ejecución y las intancias del manejador de eventos del `scope`, y se ejecuta un manejador de fallos. El `scope` finaliza sin éxito.
- Terminación externa: si un `scope` en ejecución ha recibido una señal de terminación, entonces se terminan las actividades en ejecución y las instancias del manejador de eventos. El `scope` finaliza sin éxito.

El operador **XMC** elimina la definición de un manejador de compensación, y el operador **XMT**, la de un manejador de terminación. En ambos casos, WS-BPEL los sustituirá por el manejador por omisión de cada uno de ellos. Veamos un ejemplo:

Código original:

```
<scope name="compra" ... >
  ...
  <faultHandlers>
      <catch faultName="rechazarOrden"> ...
      </catch>
  </faultHandlers>
  <compensationHandler>
      <invoke partnerLink="Vendedor" ... />
  </compensationHandler>
</scope>
```

Mutante generado por XMC:

```
<scope name="compra" ... >
  ...
```

```
  <faultHandlers>
      <catch faultName="rechazarOrden"> ...
      </catch>
  </faultHandlers>


</scope>
```

La actividad `throw` permite lanzar un fallo determinado, cuyo nombre se especifica mediante el atributo `faultName`. El operador **XTF** cambia el nombre del fallo lanzado por una actividad `throw` por otro del mismo ámbito, modelando la confusión a la hora de especificar el fallo a lanzar.

Código original:

```
<if>
  <condition>Stock &gt; 100</condition>
  <flow> ... </flow>
  <elseif>
    <condition>Stock &ge; 0</condition>
    <throw faultName="NoHayStock" ... />
  </elseif>
  <else>
     <throw faultName="ArticuloNoDisp" />
  </else>
</if>
```

Mutante generado por XTF:

```
<if>
  <condition>Stock &gt; 100</condition>
  <flow> ... </flow>
  <elseif>
    <condition>Stock &ge; 0</condition>
    <throw faultName="ArticuloNoDisp" ... />
  </elseif>
  <else>
     <throw faultName="ArticuloNoDisp" />
  </else>
</if>
```

La actividad `rethrow` permite volver a lanzar un fallo previamente capturado, pudiéndose utilizar solamente dentro de un manejador de fallos. El operador **XER** elimina una actividad `rethrow`, modelando el olvido de su inclusión en el manejador de fallos.

Código original:

```
<faultHandlers>
  <catch faultName="LibroNoStockExcep"
    faultVariable="LibroNoStockVar">
    ...
    <rethrow> ... </rethrow>
  </catch>
  <catchAll> ... </catchAll>
</faultHandlers>
```

Mutante generado por XER:

```
<faultHandlers>
  <catch faultName="LibroNoStockExcep"
    faultVariable="LibroNoStockVar">
    ...

  </catch>
  <catchAll> ... </catchAll>
</faultHandlers>
```

Los manejadores de eventos pueden contener elementos `onEvent`, que determinan las acciones a realizar cuando se produce un evento dado, y un elemento `onAlarm` que especifica las actividades a realizar si después de un tiempo determinado no se ha producido ningún evento. El operador **XEE** elimina un elemento `onEvent` de un manejador de eventos, modelando el olvido a la hora de especificar un evento que puede recibir el proceso. La eliminación del elemento `onAlarm` ya está modelada por el operador **APA**.

Código original:

```
<eventHandlers>
  <onEvent partnerLink="compra"
    operation="estadoPedido" ... >
    <scope> ... </scope>
  </onEvent>
  <onEvent partnerLink="compra"
    operation="cancelarPedido" ... >
    <scope> ... </scope>
  </onEvent>
</eventHandlers>
```

Mutante generado por XEE:

```
<eventHandlers>


  <onEvent partnerLink="compra"
    operation="cancelarPedido" ... >
    <scope> ... </scope>
  </onEvent>
</eventHandlers>
```

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

Con este capítulo se ha llevado a cabo uno de los objetivos de este PFC: estudiar y analizar las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL y los definidos para otros lenguajes.

Es la primera vez que se ha realizado este mismo estudio, que se ha presentado en un artículo. Este artículo [4] ha sido aceptado en el V Taller PRIS 2010.

3.1. Equivalencias entre operadores

En esta sección se realiza un estudio sobre las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL 2.0 en [16, 17] con los definidos para otros lenguajes lenguajes de programación: C definidos en [2, 8], C# definidos en [9, 10], C++ definidos en [55], Java definidos en [32, 5, 24, 50, 34], SQL definidos en [6, 53, 48, 49, 11], Ada definidos en [45], ASP .NET definidos en [36] y Fortran definidos en [31].

Los operadores definidos para WS-BPEL 2.0 se clasifican en cuatro categorías, de acuerdo con el tipo de elemento sintáctico de WS-BPEL con el que se relacionan. Las categorías se identifican con una letra mayúscula y son las siguientes:

- Mutación de identificadores (I)
- Mutación de expresiones (E)
- Mutación de actividades (A)
- Mutación de condiciones excepcionales y eventos (X)

Dentro de cada categoría se definen varios operadores de mutación que se identifican mediante tres letras mayúsculas: la primera de ellas coincide con la que identifica la categoría a la que pertenece el operador, mientras que las dos últimas identifican al operador dentro de la categoría. La Tabla 3.1 muestra el nombre y una descripción de estos operadores de mutación para WS-BPEL 2.0 definidos en [16, 17], clasificados por categorías.

Tabla 3.1: Operadores de mutación para WS-BPEL 2.0

OPERADOR	DESCRIPCIÓN
	MUTACIÓN DE IDENTIFICADORES
ISV	Sustituye el identificador de una variable por el de otra del mismo tipo
	MUTACIÓN DE EXPRESIONES
EAA	Sustituye un operador aritmético (`+`, `-`, `*`, `div`, `mod`) por otro del mismo tipo
EEU	Elimina el operador menos unario de cualquier expresión
ERR	Sustituye un operador relacional (`<`, `>`, `>=`, `<=`, `=`, `!=`) por otro del mismo tipo
ELL	Sustituye un operador lógico (`and`, `or`) por otro del mismo tipo
ECC	Sustituye un operador de camino (`/`, `//`) por otro del mismo tipo
ECN	Modifica una constante numérica incrementando o decrementando su valor en una unidad, añadiendo o eliminando un dígito
EMD	Modifica una expresión de duración cambiando por 0 o por la mitad el valor inicial
EMF	Modifica una expresión de fecha límite cambiando por 0 o por la mitad el valor inicial
	MUTACIÓN DE ACTIVIDADES
	Relacionados con la concurrencia
ACI	Cambia el atributo `createInstance` de las actividades de recepción de mensajes a *no*
AFP	Cambia una actividad `forEach` secuencial a paralela
ASF	Cambia una actividad `sequence` por una actividad `flow`
AIS	Cambia el atributo `isolated` de un `scope` a *no*
	No concurrentes
AIE	Elimina un elemento `elseif` o el elemento `else` de una actividad `if`
AWR	Cambia una actividad `while` por una actividad `repeatUntil` y viceversa
AJC	Elimina el atributo `joinCondition` de cualquier actividad en la que aparezca
ASI	Intercambia el orden de dos actividades hijas de una actividad `sequence`
APM	Elimina un elemento `onMessage` de una actividad `pick`
APA	Elimina el elemento `onAlarm` de una actividad `pick` o de un manejador de eventos
	MUTACIÓN DE CONDICIONES EXCEPCIONALES Y EVENTOS
XMF	Elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos
XMC	Elimina la definición de un manejador de compensación
XMT	Elimina la definición de un manejador de terminación
XTF	Cambia el fallo lanzado por una actividad `throw`
XER	Elimina una actividad `rethrow`
XEE	Elimina un elemento `onEvent` de un manejador de eventos

A continuación, presentamos los operadores equivalentes entre dichos lenguajes de programación. La Tabla 3.2 muestra para cada operador de mutación definido para el lenguaje WS-BPEL sus operadores equivalentes definidos para otros lenguajes de programación. Nos encontramos, por tanto, ante un resumen de la comparativa que explicamos y desarrollamos a lo largo de esta sección.

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

> El operador **ISV** sustituye el identificador de una variable por el de otra del mismo tipo; pero esta sustitución sólo se realizará si ambas pertenecen al mismo ámbito. (véase Sección 2.2.1)

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

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

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

Los **operadores de mutación de expresiones** para WS-BPEL sólo sustituyen un operador por otro del mismo tipo, puesto que Estero-Botaro y col. [16, 17] consideran que éste es el fallo que puede cometerse con mayor frecuencia; mientras que los operadores definidos para otros lenguajes además de realizar sustituciones, insertan o eliminan nuevas expresiones.

Tabla 3.2: Operadores de mutación de otros lenguajes comparables con los de WS-BPEL 2.0

WS-BPEL	**C**	**C#**	**C++**	**Java**	**SQL**	**Ada**	**ASP .NET**	**Fortran**
ISV	VGSR[1] VLSR[2]		OVV		IRP[3]	OVV		SVR
EAA	OAAN		IBO1 AOR	AORB AOR	AOR	EOR	ORO	AOR
EEU				UOD				
ERR	ORRN		IBO2	ROR	ROR	ERR	ORO	ROR
ELL	OLLN		IBO3	LCR	LCR	ELR[4]	ORO	LCR
ECN	CRCR[5]					EDT[6]	EMO	CRP[7]
EMD				MXT[8]				
AIE	SSDL[9]						SMO[10]	ELSE ELSEIF
AWR	SDWD[11] SWDD[12]					SWR[13] SRW[14]		
XMF		EHR		CBD				
XTF		ENC				SER		

[1]Para que este operador sea comparable con **ISV** consideramos que muta variables globales de un proceso WS-BPEL.

[2]Para que este operador sea comparable con **ISV** consideramos que muta variables pertenecientes al mismo ámbito (*scope*) de un proceso WS-BPEL.

[3]Este operador puede compararse con **ISV** si lo definimos como: *cada "variable" (en lugar de parámetro) es reemplazada por otra "variable" (de tipo compatible)*.

[4]Consideramos que sólo cambia cada operador por `AND` o `OR`, y no por `XOR`, `AND THEN` ni `OR ELSE`.

[5]Además de incrementar o decrementar el valor de la constante en una unidad, **ECN** permite añadir o eliminar alguna cifra. Sin embargo, **CRCR** permite aumentar o disminuir el valor en un número real.

[6]Este operador puede comparse con **ECN** si consideramos únicamente mutaciones de una expresión constante.

[7]Si el valor de la constante es de precisión doble, se incrementa o decrementa su valor en un $10\,\%$, si es 0 es reemplazado por $,01$ y $-0{,}1$.

[8]**EMD** modifica una expresión de tiempo mientras que **MXT** modifica el parámetro tiempo de las llamadas de unos métodos determinados. Cada uno en su contexto realiza una tarea similar.

[9]**AIE** elimina el elemento `elseif` o `else` mientras que **SSDL** elimina la sentencia contenida en el `else`.

[10]**AIE** elimina el elemento `elseif` o `else` mientras que **SMO** elimina parte o todo de una sentencia.

[11]Este operador cambia la sentencia `do-while`, en lugar de la actividad `repeatUntil` considerada en **AWR**, por un `while`.

[12]Este operador cambia la sentencia `while` por un `do-while`, en lugar de la actividad `repeatUntil` considerada en **AWR**.

[13]Este operador cambia la sentencia `loop`, en lugar de la actividad `repeatUntil` considerada en **AWR**, por un `while`.

[14]Este operador cambia la sentencia `while` por un `loop`, en lugar de la actividad `repeatUntil` considerada en **AWR**.

> El operador **EAA** sustituye, en una expresión aritmética, un operador aritmético (+, −, *, `div`, `mod`) por otro del mismo tipo. (véase Sección 2.2.2)

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

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

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

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

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

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

No hemos encontrado ningún operador para C# definido para este propósito.

> El operador **EEU** elimina el operador − unario de cualquier expresión en la que aparezca. (véase Sección 2.2.2)

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

> El operador **ERR** sustituye en una expresión relacional un operador relacional (<, >, <=, >=, =, !=) por otro del mismo tipo. (véase Sección 2.2.2)

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

Tampoco hemos encontrado ningún operador para C# definido para este propósito.

> El operador **ELL** sustituye en una expresión lógica un operador lógico (`and`, `or`) por otro del mismo tipo. (véase Sección 2.2.2)

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

No existe un operador para C# definido para este propósito.

> El operador **ECN** modifica una constante numérica incrementando o decrementando su valor en una unidad, añadiendo o eliminando un dígito. (véase Sección 2.2.2)

El operador de mutación para C equivalente a **ECN** es **CRCR**. Éste, además de aumentar o disminuir el valor de la constante en una unidad, lo hace por un número real. Sin embargo, este operador para C, al igual que para el resto de lenguajes, no permite añadir o eliminar un dígito a la constante.

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

> El operador **EMD** modifica una expresión de duración de dos formas: la sustituye por 0, lo que implica que la condición se cumple inmediatamente, y por la mitad del valor expresado inicialmente; esto tiene como objetivo verificar si el margen de seguridad especificado por la duración es adecuado. Este operador es aplicable a la actividad `wait`, y al elemento `onAlarm` de la actividad `pick` y de los manejadores de eventos. (véase Sección 2.2.2)

El único operador existente para modificar una expresión de duración, además del **EMD**, es definido para Java: **MXT**. Éste modifica el parámetro opcional de tiempo de las llamadas de métodos `wait()`, `sleep()`, `join()` y `await()`. La mutación consiste en incrementar o decrementar el tiempo en un factor de 2 (`t/2` o `t*2`). Mientras que el operador para Java modifica el parámetro tiempo de las llamadas de unos métodos determinados, el operador definido para WS-BPEL modifica una expresión de

tiempo. Cada uno en su contexto realiza una tarea similar, aunque en Java las mutaciones consisten en incrementar o decrementar el tiempo en un factor de 2 y en WS-BPEL consisten en cambiar el valor inicial por 0 o por la mitad.

Algunos de los **operadores de mutación de actividades** para WS-BPEL modelan el fallo que puede cometerse al elegir una actividad que no es la más adecuada para las acciones que se deben realizar, sustituyendo una actividad por otra. Los demás modelan la elección de un valor incorrecto para los atributos de las actividades, sustituyendo para ello su valor actual por otro valor válido. Estos operadores se clasifican en dos tipos, los **relacionados con la concurrencia** y los **no concurrentes**.

> El operador **AIE** elimina un elemento `elseif` o el elemento `else` de una actividad `if`. (véase Sección 2.2.3)

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

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

> El operador **AWR** cambia una actividad `while` por otra `repeatUntil` y viceversa. (véase Sección 2.2.3)

Existen dos operadores para C que podrían ser equivalentes a **AWR**: **SDWD** y **SWDD**, si consideramos que los cambios se van a realizar entre elementos `repeatUntil` y `while`, en lugar de `do-while`. **SDWD** cambia la sentencia `while` por un `do-while` y **SWDD** realiza la operación inversa. En este caso, mientras que Estero-Botaro y col. [16, 17] han definido un único operador, **AWR**, para realizar ambas mutaciones, Agrawal y col. [2] definen dos operadores distintos: **SDWD** y **SWDD**.

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

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

Los **operadores de mutación relacionados con las condiciones excepcionales y eventos** para WS-BPEL están relacionados con los distintos tipos de manejadores que proporciona WS-BPEL: de fallos, de eventos, de compensación y de terminación.

> El operador **XMF** elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos. Si al realizar la mutación sólo existe un elemento (`catch` o `catchAll`) en el manejador de fallos, éste también se eliminará. (véase Sección 2.2.4)

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

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

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

> El operador **XTF** cambia el nombre del fallo lanzado por una actividad `throw` por otro del mismo ámbito. (véase Sección 2.2.4)

La actividad `throw` permite lanzar un fallo determinado, cuyo nombre se especifica mediante el atributo `faultName`. En el caso de los lenguajes tradicionales, la instrucción `throw` también tiene el mismo comportamiento.

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

3.2. Conclusiones y trabajo futuro

Hemos realizado un estudio exhaustivo sobre los diferentes trabajos que tratan sobre la definición de operadores de mutación para los lenguajes C, C#, C++, Java, SQL, Ada, ASP .NET y Fortran.

Además, hemos investigado sobre cuáles son las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para los lenguajes de programación estudiados.

Es la primera vez que se realiza un análisis específico sobre cuáles son los operadores de mutación de otros lenguajes comparables con los definidos para WS-BPEL, que hemos presentado en una tabla resumen. Podemos observar que sólo 11 de los 25 operadores definidos para WS-BPEL, es decir, un $44\,\%$, son equivalentes a operadores definidos para otros lenguajes.

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

4 Desarrollo del calendario

A continuación se especifican y enumeran las distintas tareas que se han llevado a cabo en este PFC. Estas tareas se representan mediante bloques. Algunas de éstas se pueden solapar en el tiempo y, en algunas ocasiones, no es necesario que termine una actividad para comenzar la siguiente.

4.1. Fases

4.1.1. Fase 1: Elicitación de requisitos

En esta fase el alumno tomó contacto, por primera vez, con el grupo de investigación SPI&FM. Este grupo presentó sus líneas de investigación y explicó a grandes rasgos en dónde se situaba el presente PFC.

Durante esta fase hubo distintas reuniones con los miembros más expertos en GAmera y las pruebas de mutaciones. A partir de varias reuniones se realizó la elicitación de requisitos de este PFC.

4.1.2. Fase 2: Estudio de lenguajes y tecnologías

Una vez obtenidos los requisitos del PFC se pasó a la fase de estudio de los lenguajes y las tecnologías que se utilizan en este PFC: WSDL, SOAP, WS-BPEL, XSLT, XPath, XML Schema, JUnit y BPELUnit.

Cabe destacar el gran esfuerzo que se realizó durante esta fase, sobre todo, durante el aprendizaje de los lenguajes WS-BPEL y WSDL, y el framework BPELUnit desconocidos en su totalidad por el alumno.

4.1.3. Fase 3: Búsqueda bibliográfica y operador XEE

En esta fase se trabajó, sobre todo, en búsqueda bibliográfica especializada en ejemplos de composiciones WS-BPEL. Como se comentará más adelante, no existe mucha información al respecto.

Tras una búsqueda exhaustiva se comenzó a trabajar para incorporar el operador de mutación **XEE** al PFC.

Ésta es la fase más complicada de las que se refieren a los operadores de mutación. Se encontraron problemas complejos de resolver, debido a la gran cantidad de restricciones estáticas que presenta el lenguaje WS-BPEL. Lo que conllevó una mayor dedicación a esta fase.

Cabe destacar que en cada fase en la que interviene un operador, se realiza la fase de análisis, diseño, implementación y pruebas, ya que se ha seguido un desarrollo de software por incrementos. Para implementar cada operador fue necesario, además, adaptar la composición WS-BPEL para que se le pudiese aplicar el operador, y crear los casos de prueba unitarios tanto para los operadores como para la composición modificada.

4.1.4. Fase 4: Operador XMF y XER

Debido a la complejidad del desarrollo del operador **XEE** se comenzó esta fase antes de terminar la anterior. En esta fase se desarrolló el operador **XMF**.

Puesto que el operador **XER** pertenece a la misma categoría que **XMF** y, por tanto, estos dos operadores están relacionados, se llevó a cabo su desarrollo en paralelo.

4.1.5. Fase 5: Periodo de investigación

Durante los meses de febrero a mayo se realizó un periodo de investigación sobre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes de programación.

Una parte del contenido creado durante esta fase ha sido presentado y aceptado en un artículo de investigación [4].

4.1.6. Fase 6: Operador XMT

En esta fase se desarrolló el operador **XMT**.

4.1.7. Fase 7: Operador XTF

En esta fase se desarrolló el operador **XTF**.

4.1.8. Fase 8: Redacción memoria PFC

La redacción de la memoria del PFC comenzó en el mes de marzo. Se podría haber comenzado antes, pero se decidió comenzar cuando se obtuviesen los resultados de las primeras fases. De esta forma, se tenía una visión más amplia y unos conocimientos más sólidos para escribir con precisión.

4.1.9. Fase 9: Operador ASI

En esta fase se desarrolló el operador **ASI**.

4.1.10. Fase 10: Operador EMF

En esta fase se desarrolló el operador **EMF**.

4.1.11. Fase 11: Optimización XTF y ASI

En el mes de abril el grupo propuso una mejora en la definición de estos operadores de mutación por lo que fue necesario modificarlos y optimizarlos.

4.1.12. Fase 12: Revisión y validación de todos los operadores

Durante esta fase se revisó que todos los operadores funcionaban correctamente, eran válidos y que no existía ningún fallo durante la ejecución de todas las pruebas unitarias tanto de los operadores de mutación como de la composición WS-BPEL adaptada.

Finalmente, los directores del PFC confirmaron que el desarrollo de este PFC era correcto y se integró dentro del código de GAmera.

4.2. Diagrama de Gantt

Se ha elaborado un diagrama Gantt (Figura 4.1 en la página 67) para poder visualizar con mayor facilidad la distribución de las tareas.

Se ha considerado que un día ideal equivale a la cantidad de trabajo que puede realizarse en un día por una persona, sin distracción alguna y a máximo rendimiento.

Se ha empleado la herramienta Gantt Project para dibujar el diagrama y GNOME Planner para calcular las fechas con mejor precisión. Ambos son código abierto: la primera está hecha en Java y disponible en http://ganttproject.sourceforge.net, y la segunda se halla escrita en C++ con la biblioteca GTK+ y se puede encontrar bajo la dirección http://live.gnome.org/Planner.

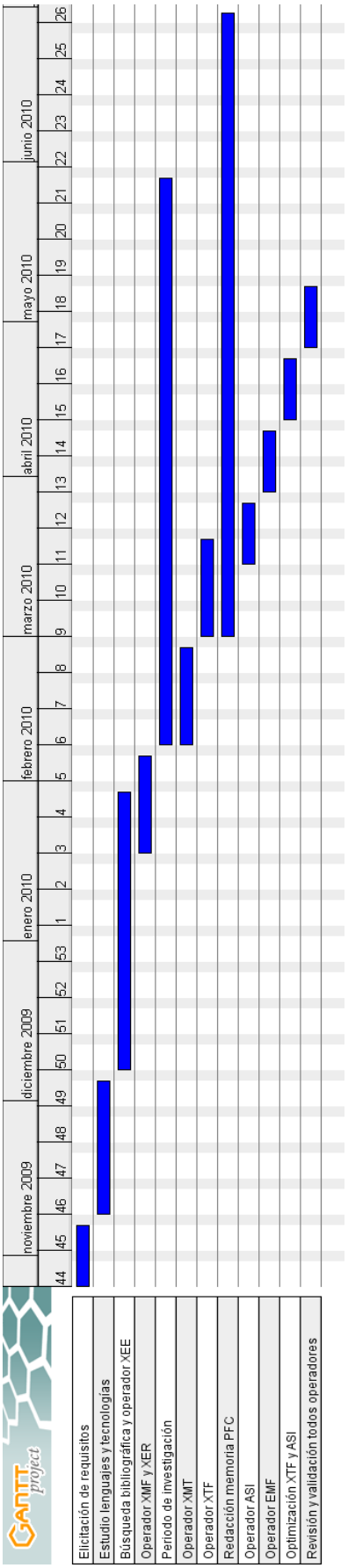

Figura 4.1: Diagrama de Gantt

5 Descripción general del proyecto

5.1. Perspectiva del producto

5.1.1. Entorno del producto

La herramienta **mutationtool** se enmarca dentro de GAmera [13], una herramienta de generación de mutantes para WS-BPEL que en vez de generarlos todos genera sólo un subconjunto de ellos. GAmera se divide en dos partes: un algoritmo genético que dirige todo el proceso y un entorno que compara los mutantes con el proceso original, para comprobar si su comportamiento varía o no. En concreto, **mutationtool** se integra en esta segunda parte.

Como se comenta en 6.6.1, el sistema de ejecución de GAmera [13] es una simplificación y una extensión del que utiliza Takuan [46].

Ambos sistemas dependen de Java 6, Tomcat 5.5, ActiveBPEL 4.1, XMLBeans 2.1.0, BPELUnit 1.4 y Saxon-B 9.1.0.1. Además, GAmera depende específicamente de **mutationtool** 1.0.7, galib 2.4.7 y JUnit 4.

A su vez, **mutationtool** depende de los módulos AnalizadorXPath, ConversorXPath e InstrumentadorBPEL. Estos módulos pueden descargarse en `https://neptuno.uca.es/redmine/projects/sources-fm/repository/show/trunk/src`.

5.1.2. Interfaz de usuario

La herramienta **mutationtool** no presenta una interfaz gráfica. Para interactuar con esta herramienta se utilizará la terminal de GNU (GNU is Not Unix)/Linux mediante líneas de órdenes.

5.2. Funciones

Las funciones de **mutationtool** son las siguientes:

- Analizar una composición WS-BPEL. Esto consiste en identificar qué instrucciones o elementos del programa original pueden mutarse.
- Aplicar un operador a la composición WS-BPEL original para generar un mutante.
- Aplicar, de una vez, todos los operadores a la composición WS-BPEL original para generar todos los mutantes posibles.
- Ejecutar la composición WS-BPEL sobre el conjunto de casos de prueba, para obtener si ha pasado con éxito o no cada uno de estos casos.
- Ejecutar los mutantes y comparar hasta que se encuentre la primera diferencia entre la salida de la composición original y las salidas de las ejecuciones de las mutaciones.
- Ejecutar todos los mutantes y comparar la salida de la composición original con las salidas de las ejecuciones de las mutaciones.
- Comparar dos ficheros que contengan dos salidas de ejecución de una composición WS-BPEL, ya sea la composición original o un mutante de dicha composición, sobre el conjunto de casos de prueba.
- Normalizar una composición WS-BPEL, es decir, obtener una forma canónica de ésta.

5.3. Características del usuario

Como ya se ha comentado en más de una ocasión, este PFC tiene como objetivo completar una herramienta real denominada GAmera, desarrollada en el grupo de investigación de SPI&FM, que se encarga de la generación de mutantes para WS-BPEL, pero en lugar de generarlos todos genera sólo un subconjunto de ellos. Dicha herramienta se divide en dos partes: un algoritmo genético que dirige todo el proceso y un entorno que compara los mutantes con el proceso original, para comprobar si su comportamiento varía o no.

Por tanto, los usuarios que utilicen este producto deberán tener conocimientos, al menos, de las pruebas de mutaciones de programas [25], del análisis de mutaciones y del lenguaje WS-BPEL 2.0 [43].

5.4. Restricciones generales

5.4.1. Control de versiones

Puesto que se ha realizado un proceso incremental de desarrollo de software, se ha necesitado un sistema de control de versiones de todas las fuentes del proyecto.

Estos sistemas permiten almacenar todas las versiones de un árbol de ficheros, pudiendo así manipular todas las revisiones de cualquier fichero en cualquier momento.

Además de servir como una medida de seguridad contra la pérdida de información accidental, agilizan los cambios drásticos, ya que no hay que establecer medidas especiales por si fallaran: se pueden revertir los cambios hechos en cualquier momento.

En particular, Subversion es un sistema que trata de resolver las insuficiencias del conocido CVS (Concurrent Versions System), pudiendo mantener revisiones de directorios completos, establecer propiedades especiales sobre los elementos del repositorio y enviar nuevas revisiones de forma atómica, entre otras cosas.

Dispone de excelente documentación, con un libro [7] disponible bajo la licencia Creative Commons y otro libro [37] sobre Subversion.

Como este PFC se ha enmarcado dentro de un proyecto más complejo, se ha creado una rama de desarrollo para este PFC. De esta forma, hasta que el desarrollo realizado en esta rama no ha sido revisado y validado por los directores del PFC no se ha fusionado con la rama principal del proyecto.

5.4.2. Lenguajes de programación y tecnologías

Los lenguajes de programación y las tecnologías que se han utilizado durante el desarrollo de este PFC son:

- WSDL: para describir servicios Web.
- SOAP: para intercambiar mensajes en un entorno distribuido.
- WS-BPEL: para realizar la composición de servicios Web.
- XSLT: para transformar documentos XML en otros, utilizando reglas de plantilla.
- XPath: para construir expresiones que recorran y procesen documentos XML.
- XML Schema: para describir la estructura y las restricciones de los contenidos de los documentos XML.
- JUnit: framework para realizar pruebas unitarias, permite realizar la ejecución de clases Java de manera controlada y evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

- BPELUnit: framework para realizar pruebas de unidad de composiciones WS-BPEL.

5.4.3. Herramientas

Para desarrollar este PFC se han usado las siguientes herramientas:

- Apache Ant: Sistema de gestión de tareas de compilación, empaquetado, generación de documentación y lanzamiento de pruebas de unidad, entre otras. Es similar al conocido GNU Make, siendo la principal diferencia su soporte multiplataforma, gracias a estar implementado en Java.
- Eclipse: IDE (Integrated Development Environment) desarrollado en sus inicios por IBM que es, junto con NetBeans, uno de los más usados hoy en día. Aunque es particularmente popular en la comunidad Java, existen versiones para otros lenguajes, como C o C++. Dispone de herramientas para hacer rápidamente refactorizaciones en el código, facilitando enormemente cualquier cambio al diseño de una aplicación.
- TkDiff: un visor de diferencias gráfico basado en el framework Tk.
- XMLEye: un visor extensible de formatos XML.

5.4.4. Sistemas operativos y hardware

El Proyecto en el que se enmarca este PFC ha sido desarrollado y probado únicamente en GNU/Linux, ejecutando la versión 9.10 de la distribución Ubuntu y la versión 11.0 de la distribución openSUSE. Por tanto, este PFC se ha desarrollado y probado únicamente en GNU/Linux.

En cuanto al hardware se recomienda disponer de una máquina con una memoria, al menos, de 1 GB de RAM, puesto que el proceso de ejecución de GAmera (véase 6.6.1) consume muchos recursos.

6 Desarrollo del proyecto

Procederemos a describir el análisis, diseño, implementación y realización de pruebas del proyecto.

En particular, las secciones de análisis y diseño describirán la última iteración del producto, al contener ésta la arquitectura y diseño definitivos, que también han ido cambiando y han ido siendo refinados a lo largo de las iteraciones.

6.1. Modelo de ciclo de vida

Para desarrollar este sistema se ha utilizado el modelo de ciclo de vida de desarrollo software "incremental".

Como veremos más adelante, todas las mutaciones que se generan son de orden uno, es decir, dado una composición WS-BPEL original sólo se aplicará, en un momento dado, un único operador. Por tanto, este modelo "incremental" se adapta perfectamente a nuestras necesidades.

Se parte de un conjunto de operadores ya incorporados al sistema y, en cada incremento, se realizan las fases de anális, diseño, implementación y pruebas para cada nuevo operador de mutación WS-BPEL que se añada al sistema.

De esta forma, podrá utilizarse el sistema sin tener que esperar a que la composición original y el conjunto de casos de prueba estén adaptados para que puedan utilizarse todos los operadores.

6.2. Herramienta de modelado usada: BOUML

Genera código Java/C++/Python/Perl a partir de diagramas UML (Unified Modeling Language), realiza ingeniería inversa de Java/C++, y permite dibujar diversos diagramas UML 2.0, como los diagramas de clases, despliegue, componentes o casos de uso, entre otros.

Está disponible para Windows, GNU/Linux y Mac OS X en la web http://bouml.free.fr. Se trata de un programa escrito en C++ usando la versión 3.3.8 de la

biblioteca Qt de la compañía Trolltech, conocida por ser la biblioteca sobre la cual se halla implementado el gestor de escritorio KDE.

6.3. Requisitos

6.3.1. Funcionales

- Analizar una composición WS-BPEL. Esto consiste en identificar qué instrucciones o elementos del programa original pueden mutarse. El resultado de este análisis debe ser un fichero que contenga para cada operador de mutación el número de las instrucciones en las que el operador puede ser aplicado y el máximo valor del atributo.
- Aplicar un operador a la composición WS-BPEL original para generar un mutante. Para ello, deberá indicarse el operador, el operando (o instrucción) y, en algunos casos, también el valor del atributo. El operador, la instrucción (operando) y el valor del atributo representan a un individuo.
- Aplicar, de una vez, todos los operadores a la composición WS-BPEL original para generar todos los mutantes posibles.
- Ejecutar la composición WS-BPEL sobre el conjunto de casos de prueba, para obtener si ha pasado con éxito o no cada uno de estos casos.
- Ejecutar los mutantes y comparar hasta que se encuentre la primera diferencia entre la salida de la composición original y las salidas de las ejecuciones de las mutaciones.
- Ejecutar todos los mutantes y comparar la salida de la composición original con las salidas de las ejecuciones de las mutaciones.
- Comparar dos ficheros que contengan dos salidas de ejecución de una composición WS-BPEL, ya sea la composición original o un mutante de dicha composición, sobre el conjunto de casos de prueba.
- Normalizar una composición WS-BPEL, es decir, obtener una forma canónica de ésta.

6.3.2. De información

- El sistema deberá almacenar el nombre de cada operador de mutación definido para WS-BPEL 2.0, así como su valor que estará en el rango de 1 a 25, y el máximo valor del atributo.
- Cada individuo será representado por el operador, el operando y el atributo.

- La composición WS-BPEL original, los mutantes, el conjunto de casos de prueba y las salidas de ejecución se podrán almacenar en ficheros que serán identificados por su nombre y ubicación.

6.3.3. Requisitos de reglas de negocio

- Siempre se realizarán mutaciones de orden uno (nunca de orden superior), es decir, cada mutante se generará al aplicar únicamente uno de los operadores sobre la composición WS-BPEL original.

6.3.4. Requisitos de interfaz

Este sistema, al que denominaremos **mutationtool**, se relacionará con la herramienta denominada GAmera [13].

Para que esta herramienta pueda funcionar necesita los operadores de mutación definidos para WS-BPEL 2.0 [16, 17]. Sin estos operadores, esta herramienta no podría generar los mutantes, a partir del código original escrito en WS-BPEL y, por tanto, no se podrían llevar a cabo el análisis de mutaciones, proceso que mide la calidad de conjuntos de casos de prueba [21].

Los resultados obtenidos al realizar las distintas funcionalidades deben poderse almacenar en ficheros, identificados por un nombre y su ubicación, y ser compatibles con la herramienta GAmera.

6.3.5. Requisitos no funcionales

- Rendimiento: el sistema debe tener un alto rendimiento, sobre todo, al generar mutantes y al comparar las salidas de ejecución.
- Fiabilidad: todas las mutaciones deben realizarse correctamente y las comparaciones entre salidas deben ser muy fiables, puesto que serán fundamentales para realizar el análisis de mutaciones.
- Mantenibilidad: es fundamental para que posteriores ampliaciones y correcciones se hagan de forma rápida y sencilla, como puede ser añadir nuevos operadores de mutación para el lenguaje WS-BPEL.

6.4. Análisis del sistema

6.4.1. Casos de uso

Los casos de uso describen la secuencia de acciones, incluyendo variantes, que ejecuta un sistema y que producen unos resultados observables para un actor particular. Estos casos de uso nos permiten analizar de manera general y abstracta qué es lo se requiere en nuestro sistema.

La Figura 6.1 ilustra las relaciones existentes entre los distintos casos de uso.

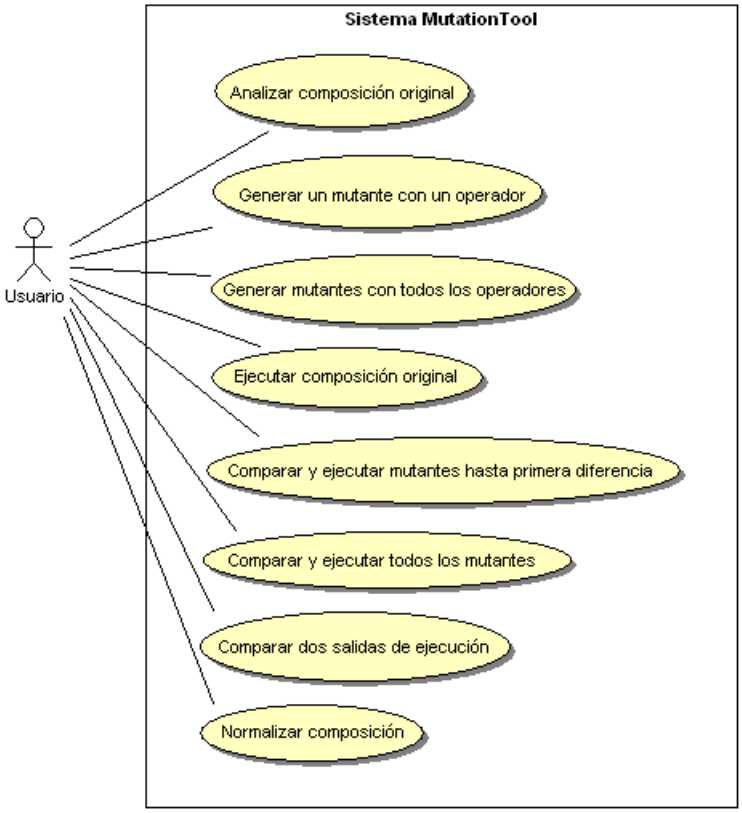

Figura 6.1: Diagrama de casos de uso

Analizar composición original

Actor principal Usuario: desea analizar la composición WS-BPEL original para obtener cuáles son los operadores que son aplicables a esta composición y cuál es el número de operandos y de atributos para cada operador.

Precondiciones Existe una composición WS-BPEL.

Postcondiciones Se muestra una lista con los operadores indicando para cada uno el número de operandos y atributos.

Escenario principal

1. El usuario indica su intención de analizar la composición original.
2. El sistema proporciona un listado con el número de operandos disponibles para cada operador en la definición de proceso WS-BPEL especificado. Para cada operador se especifica su nombre, el número de operandos y el valor máximo del atributo.

Generar un mutante con un operador

Actor principal Usuario: desea generar un mutante de la composición WS-BPEL utilizando un operador concreto.

Precondiciones Existe una composición WS-BPEL.

Postcondiciones Se obtiene el mutante.

Escenario principal

1. El usuario indica la composición WS-BPEL original, y el operador que desea aplicarle junto con el operando y el atributo.
2. El sistema genera la mutación.

Variaciones

2a. El operador no es aplicable a la composición:
   1. El sistema muestra un mensaje informando que ese operador no puede ser aplicado.

Generar mutantes con todos los operadores

Actor principal Usuario: desea generar todos los mutante de la composición WS-BPEL utilizando todos los operadores de mutación.

Precondiciones Existe una composición WS-BPEL.

Postcondiciones Se obtienen los mutantes generados con todos los operadores.

Escenario principal

1. El usuario indica la composición WS-BPEL original.
2. El sistema genera las mutaciones de la composición original, aplicando cada operador a todos sus operandos, utilizando todo el rango de valores del atributo.

Variaciones

2a. Ninguno de los operadores es aplicable a la composición:

1. El sistema muestra un mensaje informando que ningún operador puede ser aplicado.

Ejecutar composición original

Actor principal Usuario: desea ejecutar la composición WS-BPEL original sobre el conjunto de casos de prueba.

Precondiciones Existe una composición WS-BPEL y un conjunto de casos de pruebas definido.

Postcondiciones Se obtiene la salida de la composición original en la que se indica si han pasado con éxito todos los casos de prueba o no.

Escenario principal

1. El usuario indica el conjunto de casos de prueba y la composición WS-BPEL original.
2. El sistema ejecuta la composición sobre el conjunto de casos de prueba, e informa si han pasado con éxito o no cada uno de estos casos.

Comparar y ejecutar mutantes hasta primera diferencia

Actor principal Usuario: desea comparar la salida de la ejecución de la composición WS-BPEL original con las salidas de las ejecuciones de los mutantes de dicha composición sobre un conjunto de casos de prueba, hasta que se encuentre la primera diferencia.

Precondiciones Existe una composición WS-BPEL, un conjunto de casos de pruebas definido, la salida de la ejecución de la composición original y, al menos, un mutante.

Postcondiciones Se obtiene la comparación de la salida de la composición original con las de las mutaciones, indicando si los mutantes están muertos o vivos.

Escenario principal

1. El usuario indica el conjunto de casos de prueba, la composición WS-BPEL original, la salida de la ejecución de dicha composición y, al menos, un mutante.
2. El sistema compara la salida de la composición original con las de las mutaciones hasta que encuentre la primera diferencia.

Comparar y ejecutar todos los mutantes

Actor principal Usuario: desea comparar la salida de la ejecución de la composición WS-BPEL original con las salidas de las ejecuciones de todos los mutantes de dicha composición sobre un conjunto de casos de prueba.

Precondiciones Existe una composición WS-BPEL, un conjunto de casos de pruebas definido, la salida de la ejecución de la composición original y, al menos, un mutante.

Postcondiciones Se obtiene la comparación de la salida de la composición original con las de todas las mutaciones, indicando si los mutantes están muertos o vivos.

Escenario principal

1. El usuario indica el conjunto de casos de prueba, la composición WS-BPEL original, la salida de la ejecución de dicha composición y, al menos, un mutante.
2. El sistema compara la salida de la composición original con las de todas las mutaciones.

Comparar dos salidas de ejecución

Actor principal Usuario: desea comparar el resultado obtenido por dos composiciones WS-BPEL diferentes ejecutadas sobre un conjunto de casos de prueba.

Precondiciones Existen dos salidas de ejecución.

Postcondiciones Se obtiene la comparación de las dos salidas.

Escenario principal

1. El usuario indica las salidas de ejecución de dos composiciones WS-BPEL diferentes.
2. El sistema compara las dos salidas.

Variaciones

2a. El número de casos de prueba de una salida es distinto al de la otra.
   1. El sistema muestra un mensaje informando que el número de casos de prueba es distinto y no realiza la comparación.

Normalizar composición

Actor principal Usuario: desea normalizar una composición WS-BPEL.

Precondiciones Existe una composición WS-BPEL.

Postcondiciones Se obtiene la normalización de la composición, es decir, se obtiene una forma canónica de ésta.

Escenario principal

1. El usuario indica la composición WS-BPEL.
2. El sistema normaliza la composición y la muestra.

6.4.2. Modelo conceptual de datos del dominio

En la Figura 6.2 se presenta el diagrama de clases conceptuales para este sistema. Se ha utilizado la notación UML 2.0.

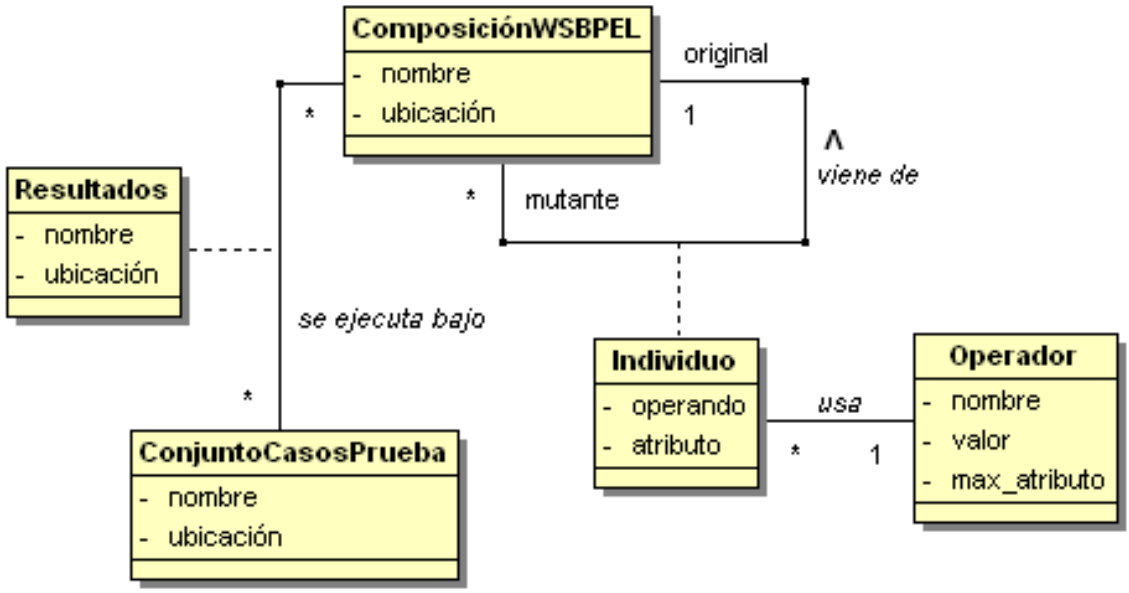

Figura 6.2: Diagrama de clases conceptuales

Una *ComposiciónWSBPEL* tendrá un nombre y la ubicación que indicará la ruta en la que se encuentra. Representa tanto la composición WS-BPEL original como uno de sus posibles mutantes, por ello, se ha empleado los roles *original* y *mutante*, respectivamente.

Un *mutante* viene de una composición *original*, tras aplicarle a la original un individuo concreto, modelado como *Individuo*.

Un *Individuo* codifica a un mutante mediante el *operador* de mutación que se aplica a la composición original. Por ello, se considera la relación en la que un *Individuo* usa un *Operador*.

Cada *Operador* tendrá un nombre, compuesto por tres letras, un valor numérico que lo identificará comprendido entre 1 y 25 (el número máximo de operadores definidos

para WS-BPEL) y el máximo valor que puede tomar el atributo, que dependerá de cada operador concreto.

El *Individuo* usa también un *operando* o instrucción que indica la instancia donde puede aplicarse el operador, y un *atributo* que representa la información necesaria para la aplicación del operador de mutación.

Para poder llevar a cabo el análisis de mutaciones, será necesario que las composiciones WS-BPEL, *ComposiciónWSBPEL*, ya sean original o mutante, se ejecuten bajo algunos conjuntos de casos de prueba, *ConjuntoCasosPrueba*, que tendrán un nombre y una ubicación concreta. Se obtendrán unos resultados, *Resultados*, que se identificarán también por un nombre y una ubicación.

6.4.3. Diagramas de secuencia

A continuación, presentamos los diagramas de secuencia que muestran la secuencia de eventos que producen los actores del sistema. Nos ayudan a la identificación de las operaciones del sistema.

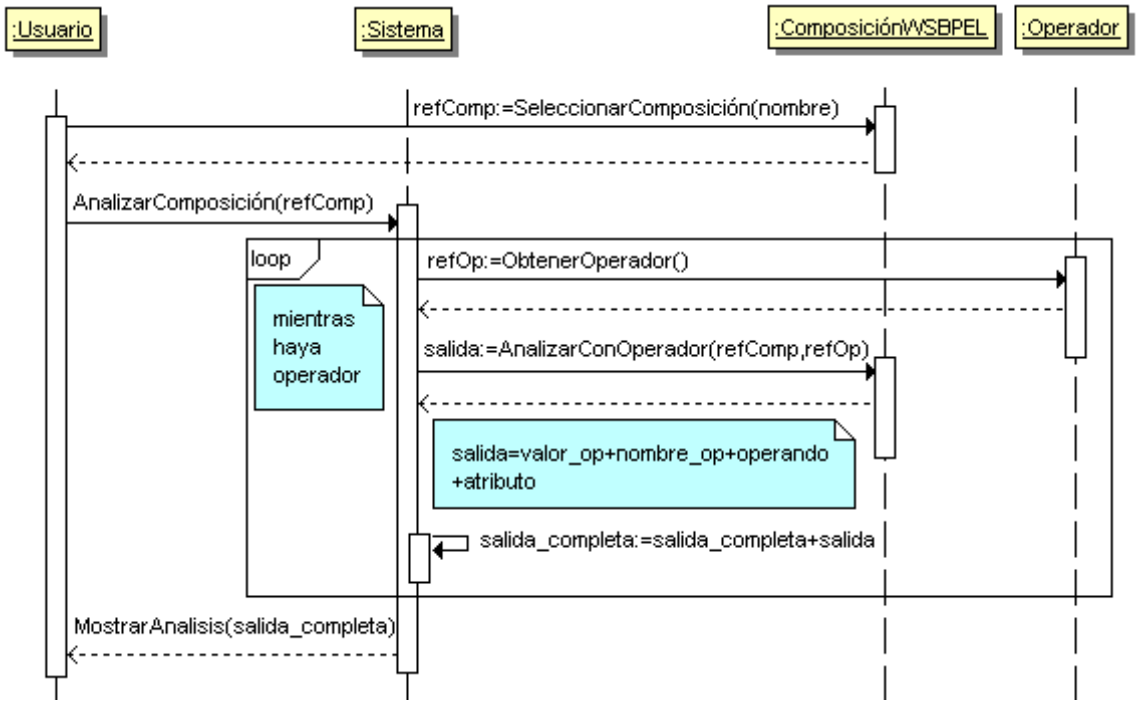

Figura 6.3: Diagrama de secuencia de analizar composición WS-BPEL

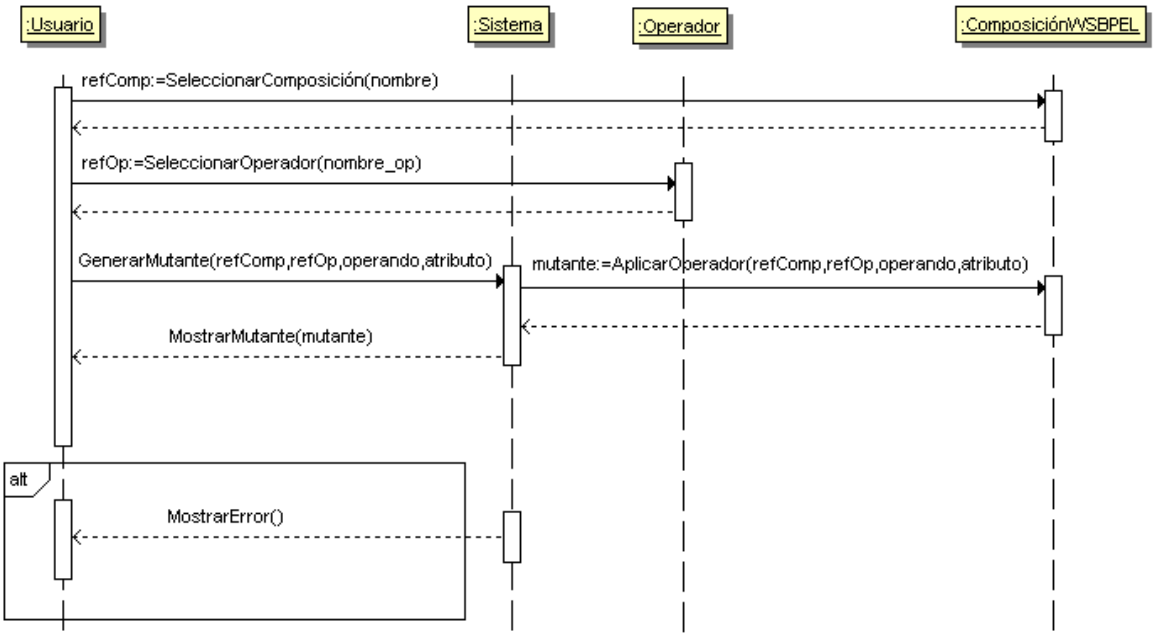

Figura 6.4: Diagrama de secuencia de generar un mutante con un operador

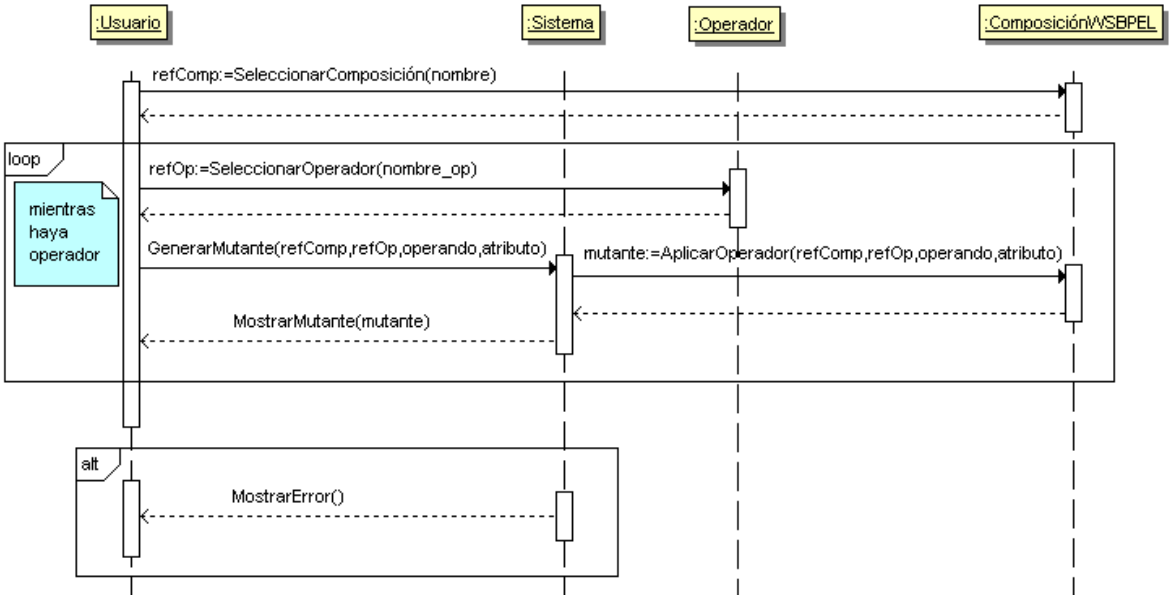

Figura 6.5: Diagrama de secuencia de generar mutantes con todos los operadores

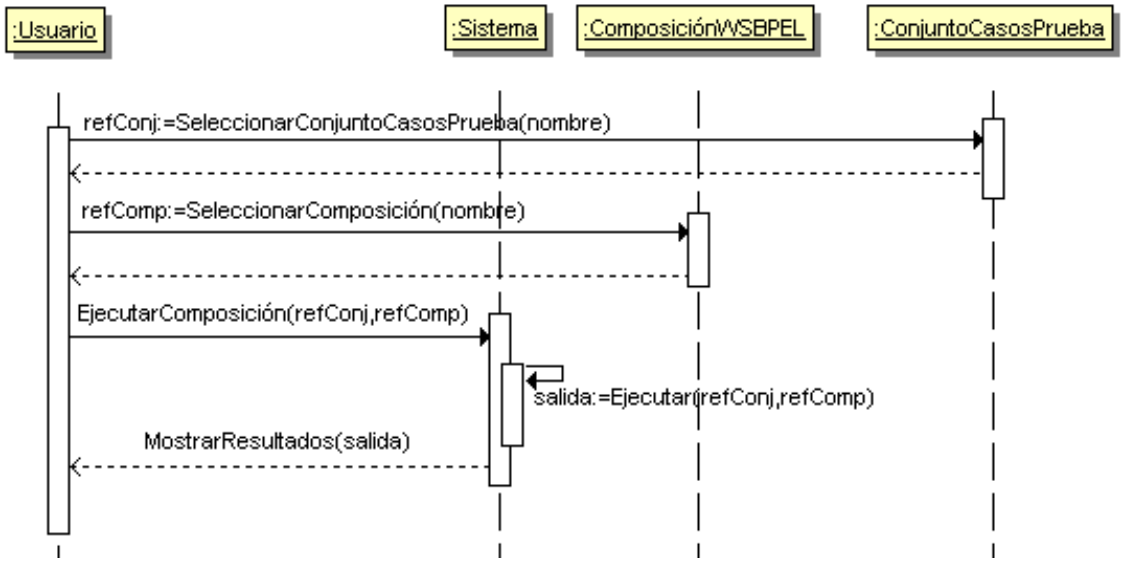

Figura 6.6: Diagrama de secuencia de ejecutar la composición WS-BPEL original

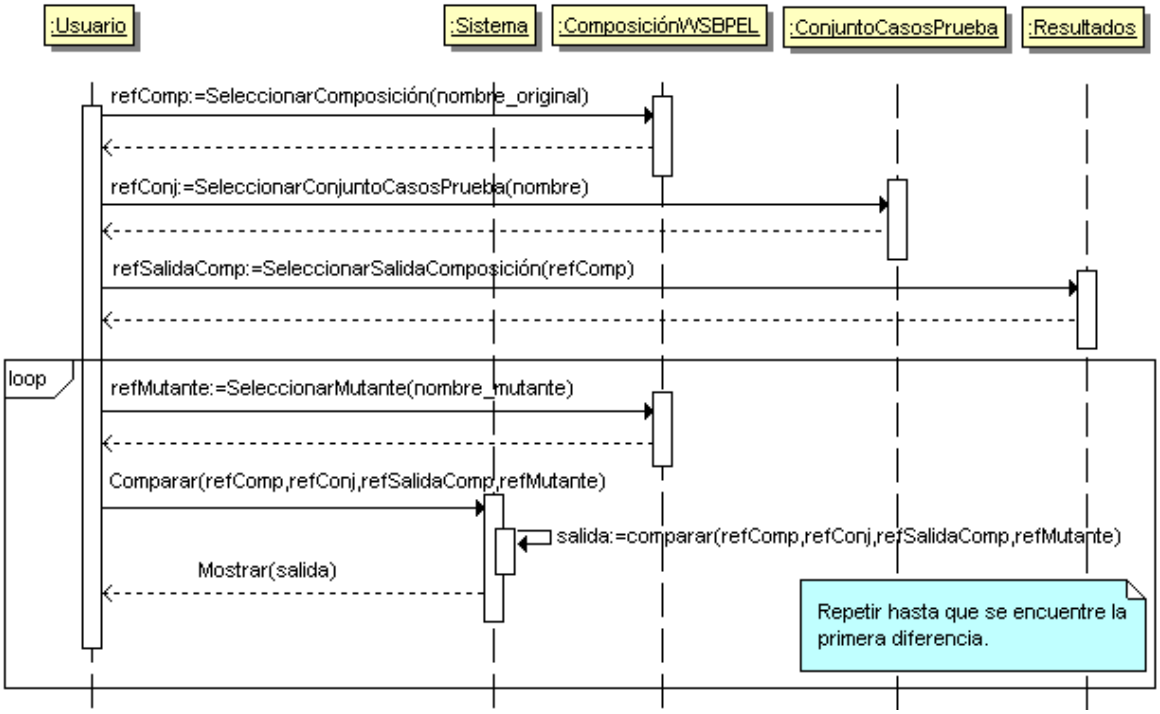

Figura 6.7: Diagrama de secuencia de comparar y ejecutar hasta la primera diferencia

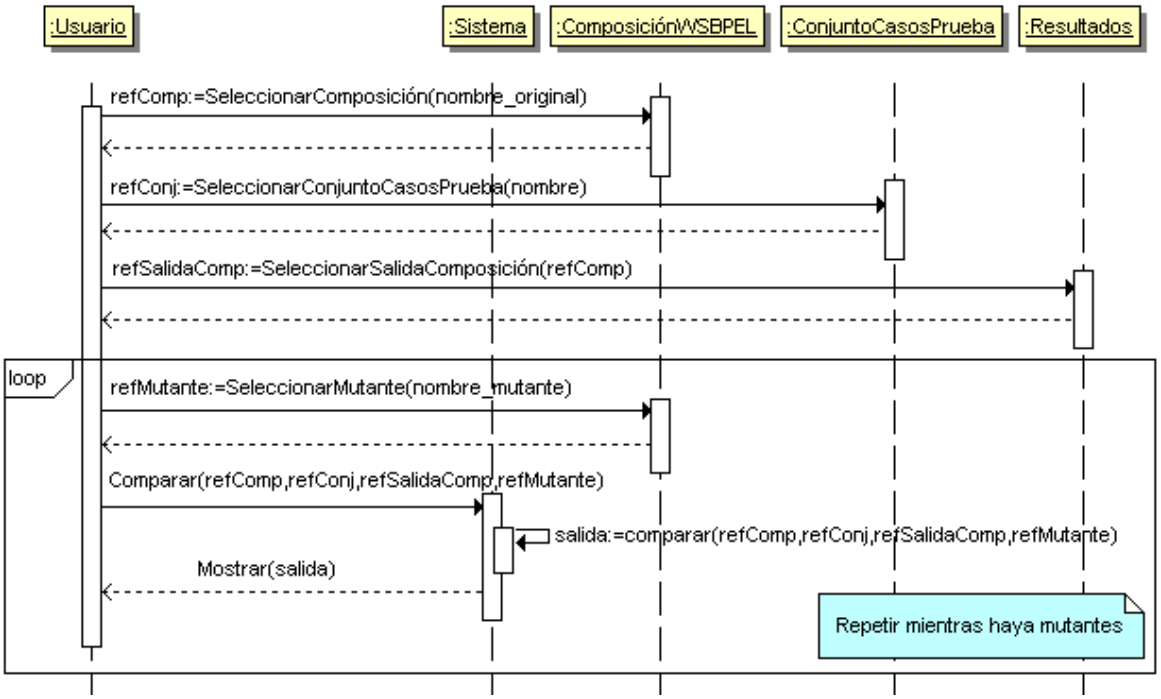

Figura 6.8: Diagrama de secuencia de comparar y ejecutar todos los mutantes

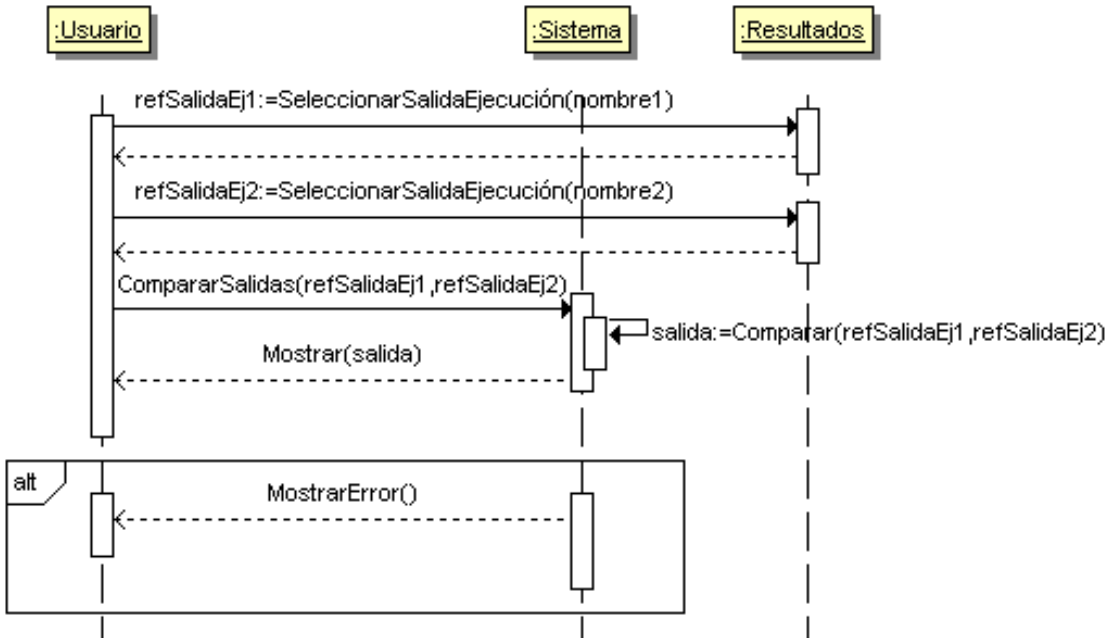

Figura 6.9: Diagrama de secuencia de comparar dos salidas de ejecución

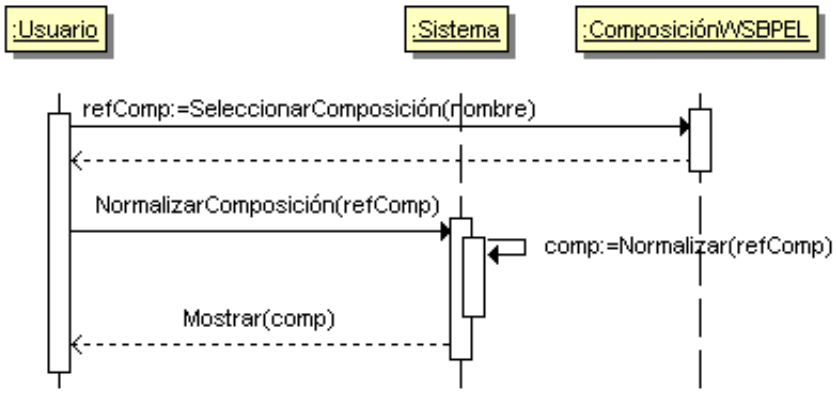

Figura 6.10: Diagrama de secuencia de normalizar una composición WS-BPEL

6.5. Diseño del sistema

Como se ha comentado anteriormente, la finalidad del desarrollo de este PFC es completar una herramienta real denominada GAmera [13].

A continuación, se resume qué es GAmera y cuál es su estructura, para poder enmarcar este PFC y comprender la importancia de su desarrollo.

6.5.1. GAmera

En esta sección se presenta una herramienta novedosa, GAmera, para la generación y ejecución automática de mutantes para composiciones de servicios web en WS-BPEL. GAmera incorpora un mecanismo de optimización que permite seleccionar un subconjunto de los mutantes totales que pueden generarse. Esto se logra mediante la utilización de un algoritmo genético que genera y selecciona sólo los mutantes de mayor calidad, reduciendo el coste computacional que implicaría la ejecución de todos los mutantes. Los resultados que proporciona esta herramienta permiten mejorar la calidad de los casos de prueba.

Estructura de GAmera

La herramienta GAmera está constituida por tres componentes principales: el *analizador*, el *generador de mutantes* y el *sistema de ejecución*, que ejecuta y evalúa los mutantes. La Figura 6.11 ilustra la estructura de GAmera.

El analizador Es el primer componente de la herramienta que actúa. Recibe como entrada la composición WS-BPEL a probar y determina los operadores de mutación que se le pueden aplicar.

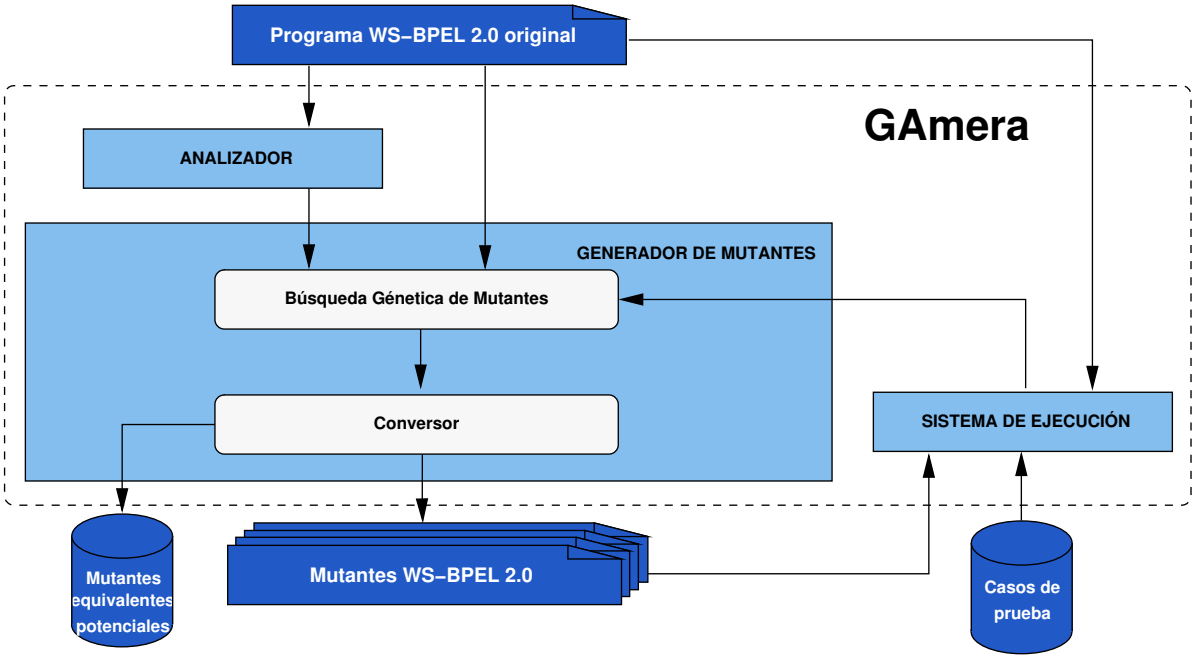

Figura 6.11: Estructura de GAmera

Generador de mutantes Los mutantes se generan a partir de la información que se recibe del *analizador*. La herramienta nos da la posibilidad de generar todos los mutantes posibles, o bien, un subconjunto de éstos que va a ser seleccionado por el algoritmo genético. En este último caso, se llamará al componente denominado *generador de mutantes*, que está compuesto por dos elementos. El primero, denominado Búsqueda Genética de Mutantes, es un algoritmo genético en el que cada individuo representa a un mutante, capaz de generar y seleccionar de forma automática un conjunto de mutantes. Esta selección se realiza aplicando una función de aptitud que mide su calidad en función de si hay o no casos de prueba que lo matan [14]. El segundo elemento es el Conversor, que transforma un individuo del algoritmo genético en un mutante WS-BPEL. Para realizar esta conversión, se utilizan hojas de estilos XSLT, una por cada operador de mutación.

Ejecución de mutantes A medida que se van generando los mutantes, el sistema los ejecuta frente a un conjunto de casos de prueba, distinguiéndose tres posibles estados para cada mutante según la salida que producen.

Muerto La salida del mutante es diferente a la del proceso original para al menos un caso de prueba.

Vivo La salida del mutante es la misma que la del proceso original para todos los casos de pruebas suministrados.

Erróneo Se ha producido un error en el despliegue del mutante y no se ha podido ejecutar. La existencia de este estado permite determinar si el diseño e implementación de los operadores de mutación es adecuado, o bien, si se

están generando mutantes que no se pueden desplegar.

Para la ejecución del programa original y los mutantes, GAmera emplea el motor WS-BPEL 2.0 ActiveBPEL 1.4 [1] y BPELUnit [39], una biblioteca de pruebas unitarias para WS-BPEL que utiliza documentos XML para describir los casos de prueba.

GAmera permite visualizar los resultados obtenidos en la ejecución de los mutantes. Muestra el número total de mutantes generados, el de mutantes muertos, vivos y erróneos. También muestra estos valores para cada operador de mutación utilizado. A partir de estos valores se puede medir la calidad del conjunto de casos de prueba utilizado.

6.5.2. Enmarcación de este PFC dentro de GAmera

Este PFC se enmarca dentro de la herramienta GAmera, concretamente en el componente de "Generador de mutantes".

Como se ha comentado anteriormente, para poder generar las mutaciones necesitamos aplicar los denominados "operadores de mutación" a las composiciones WS-BPEL originales.

En este PFC se han implementado algunos de estos operadores de mutación definidos para el lenguaje WS-BPEL 2.0 (véase §2.2), concretamente:

- **EMF**: modifica una expresión de fecha límite.
- **ASI**: intercambia el orden de dos actividades hijas de una actividad `sequence`.
- **XMF**: elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos.
- **XMT**: elimina la definición de un manejador de terminación.
- **XTF**: cambia el fallo lanzado por una actividad `throw`.
- **XER**: elimina una actividad `rethrow`.
- **XEE**: elimina un elemento `onEvent` de un manejador de eventos.

Debemos tener en cuenta que si hubiésemos implementado únicamente los operadores no se habrían cubierto las necesidades de dicha herramienta, puesto que es necesario, además, disponer de una composición WS-BPEL a la que pueda aplicarse estos operadores, así como un conjunto de casos de prueba acorde con las posibles mutaciones que se pueden generar.

El proceso que se ha establecido para conseguir los objetivos ha sido el siguiente:

1. Implementar el código XSLT de cada operador, que realice la mutación adecuada del código original escrito en WS-BPEL.

2. Definir los casos de prueba, utilizando el framework JUnit, para comprobar que los operadores se han implementado correctamente, atendiendo a:
   a) Número de operadores detectados en la composición modificada.
   b) Modificaciones introducidas al ejecutar los operadores.
   c) Comprobaciones de que las salidas de la composición original y la de los mutantes son diferentes.
3. Adaptar el ejemplo de la composición WS-BPEL 2.0 *TravelReservationService* para que le sean aplicables, al menos, los operadores implementados.
4. Crear el conjunto de casos de prueba BPTS y comprobar que contiene los casos de prueba suficientes para matar a todos los mutantes no equivalentes que produce la composición WS-BPEL adaptada.

6.5.3. Operadores de mutación

A continuación se especifica cómo se ha propuesto el diseño de los individuos en [15].

Cada individuo codifica la mutación a realizar al programa original mediante tres campos: uno para identificar el operador, otro para referenciar la instrucción donde se aplicará, y un tercero que contiene información para la aplicación del operador:

Operador Operador de mutación que se aplica. Se codifica con un valor entero entre 1 y OM, donde OM es el número de operadores de mutación definidos.

Operando Instancia donde puede aplicarse el operador. Representa el número de instrucción del programa original donde se aplicará el operador. Con objeto de realizar una distribución uniforme entre todos los operadores independientemente del número de instrucciones asociadas a cada uno, el campo se codifica con un valor entero comprendido en el rango de 1 a I, donde $I = mcm\{m_i, 1 \leq i \leq OM\}$, siendo m_i el número de instrucciones que existen en el programa original a las que se puede aplicar el operador de mutación i-ésimo. Posteriormente, es necesario realizar una traducción, de manera que el valor I_i representará una operación de mutación en la instrucción $\lceil (I_i \times m_i)/I \rceil$.

Atributo Valor que representa la información necesaria para la aplicación del operador de mutación. Dado que una mutación consiste en la alteración de un elemento del programa, este campo especifica cuál será el nuevo valor que tomará el elemento afectado por la mutación. Así, los operadores de mutación podemos clasificarlos en:

- Aquellos que cambian un elemento por otro de un conjunto. En este caso, el atributo indica el nuevo elemento que se quiere poner. Consideremos, por ejemplo, los operadores de mutación para las expresiones relacionales (>, =, <, $\geq$, $\leq$, !=) con sus respectivos valores (`1`, `2`, `3`, `4`, `5`, `6`). El individuo

que indicaría cambiar la instrucción $a > 2500$ por $a < 2500$, se codificaría con un 3.

- Aquellos que eliminan un elemento. En este caso, el atributo no tendría significado. Se mantiene en la codificación con valor 1. Un ejemplo de este tipo de operadores pueden ser aquellos que eliminan una rama `else` de una construcción `if-else`.
- Aquellos que suponen un cambio de posición de la instrucción. El atributo indicará cuál es la nueva posición donde se ubicará. Si se intercambia con la instrucción anterior se codifica con 1, y si es con la instrucción posterior se codifica con 2.

Al igual que en el campo *Operando* o *Instrucción*, con objeto de realizar una distribución uniforme entre todos los individuos, el campo *Atributo* contiene un valor entero dentro del rango 1 a V, donde $V = mcm\{v_i, 1 \leq i \leq OM\}$, siendo v_i el número de valores que puede tomar el operador de mutación i-ésimo. La Tabla 6.1 muestra para cada operador, su valor y el máximo valor del campo *Atributo*. Algunos operadores como **EEU** sólo tienen un único valor, ya que producen un único mutante, en estos casos este valor no es significante. De forma análoga se realiza una traducción para la obtención del desplazamiento.

Este esquema de codificación de individuos permite que la arquitectura pueda adaptarse fácilmente a cualquier lenguaje de programación. Simplemente habría que numerar los distintos operadores de mutación disponibles y, para cada uno de ellos, sus posibles valores.

Para llevar a cabo la implementación de dichos operadores de mutación para WS-BPEL se ha utilizado, principalmente, un lenguaje de programación orientado a objetos (Java) y un lenguaje extensible de hojas de estilos (XSL (eXtensible Stylesheet Language)), como se describe a continuación.

Java

Se ha utilizado un lenguaje de programación orientado a objetos para cumplir algunos de los requisitos no funcionales de nuestro sistema: la mantenibilidad y adaptabilidad.

Si en un futuro próximo se definen nuevos operadores de mutación para el lenguaje WS-BPEL 2.0, como es la intención de sus autores, Estero y col., nuestro sistema podrá incorporar estos nuevos operadores de una forma fácil y flexible.

Para ello, se ha utilizado el patrón *Fábrica* [18, 19] para aislar al resto del sistema de la lógica necesaria para identificar qué tipo de operador hay que crear exactamente.

Podemos ver en el diagrama de clases de diseño de la Figura 6.12 una clase abstracta, *Operator*. Las instancias de sus subclases se crean mediante el método *newOperator* de *Operator::Factory*.

Tabla 6.1: Valores y atributos para los operadores de mutación

Operador	**Valor**	**Máx. Valor del Atributo**	
ISV	1	1	
EAA	2	5	`+`, `-`, `*`, `div`, `mod`
EEU	3	1	
ERR	4	6	`<`, `>`, `>=`, `<=`, `=`, `!=`
ELL	5	2	`and`, `or`
ECC	6	2	`/`, `//`
ECN	7	4	+1, -1, añadir, eliminar
EMD	8	2	0, mitad
EMF	9	2	0, mitad
ACI	10	1	
AFP	11	1	
ASF	12	1	
AIS	13	1	
AIE	14	1	
AWR	15	1	
AJC	16	1	
ASI	17	1	
APM	18	1	
APA	19	1	
XMF	20	1	
XMC	21	1	
XMT	22	1	
XTF	23	1	
XER	24	1	
XEE	25	1	

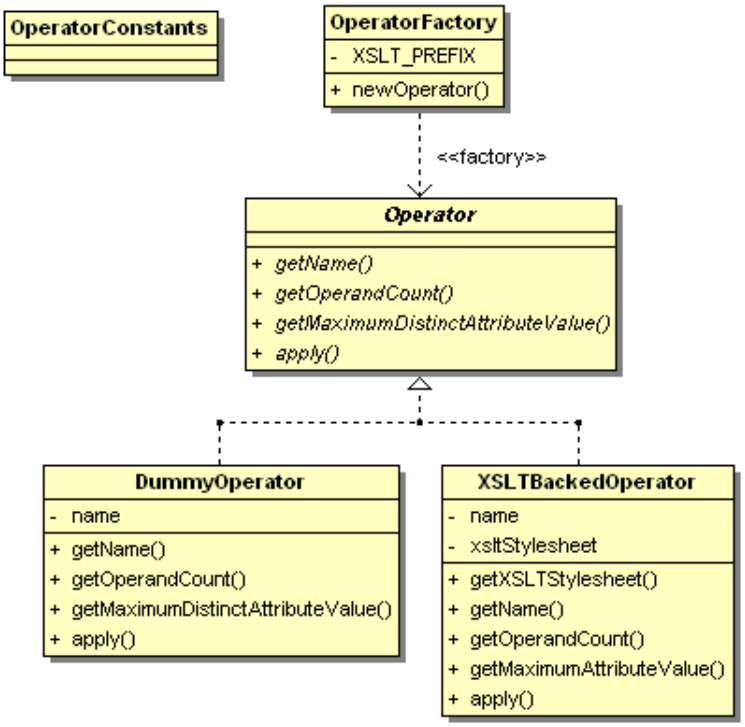

Figura 6.12: Diagrama de clases de operadores

De esta forma, se busca en el sistema un fichero del tipo *nombre-operador.xsl*. Si existe se creará un *XSLTBackedOperator*; el nombre de este fichero será el argumento que se proporcionará y que determinará qué subclase concreta de la clase abstracta base se va a crear. En caso contrario, se creará un *DummyOperator*.

Resumiendo, *Operator* es la interfaz de todo operador de mutación y *OperatorConstants* define las constantes necesarias. *DummyOperator* es un operador nulo que devuelve el proceso original y no reporta ningún resultado al analizar, y *XSLTBackedOperator* es la clase de todos los operadores cuya implementación está basada en una hoja XSLT. Las instancias no se crean directamente, sino a través de una fábrica llamada *OperatorFactory*, que sabe si crear un objeto de una clase u otra en función de si existe el *.xsl* correspondiente o no.

Hojas de estilos

Se eligió XSLT para implementar los operadores de mutación por la flexibilidad que ofrece al estar basado en ficheros de texto fácilmente intercambiables y por su capacidad de describir declarativamente las transformaciones sobre la fuente XML a otros formatos.

Las hojas de estilos se estructuran en repositorios y se utiliza la herencia entre distintas hojas.

Organización del repositorio Los operadores de mutación son parte del código fuente del proyecto Eclipse MutationAnalysis bajo source: `src/MutationAnalysis`, y las hojas XSLT que los implementan están bajo source: `src/MutationAnalysis/src/es/uca/webservices/mutants/operators/`. La estructura general es:

- `op-base.xsl`: incluye el código común a todos los operadores de mutación.
- `xpath-op-base.xsl`: incluye el código común a todos los operadores de mutación que modifican expresiones XPath.
- `delete-op-base.xsl`: incluye el código común a todos los operadores de mutación que eliminan actividades.
- `nombre-operador.xsl`: son operadores de mutación particulares, que afectan a actividades.

`generate_invalid_mutant.xsl` es un operador especial diseñado para generar mutantes *stillborn*. Se utiliza en las pruebas unitarias para comprobar que se distinguen con el valor 2 los mutantes que no se llegan a desplegar correctamente.

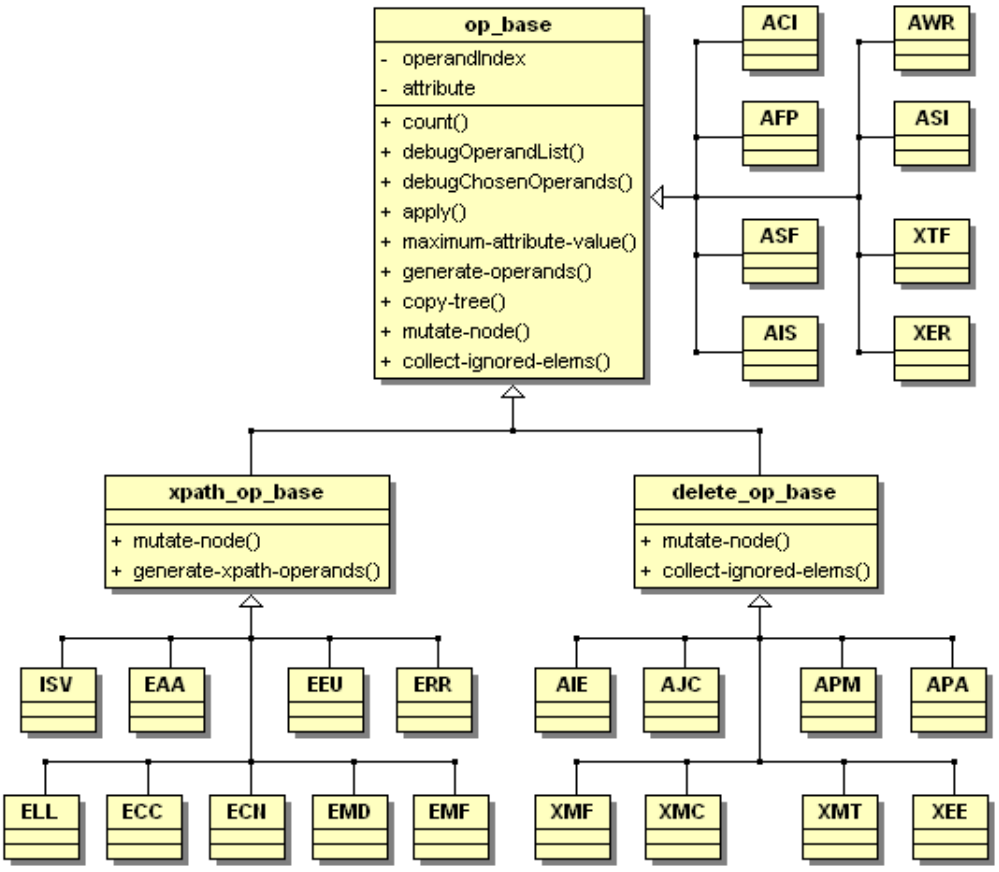

Figura 6.13: Diagrama de hojas de estilos de operadores

La Figura 6.13 muestra la herencia entre las distintas hojas de estilos, en la que puede comprobarse cuál es la función principal de cada uno de los 25 operadores definidos para WS-BPEL: operadores que afectan a actividades, operadores que afectan a expresiones XPath u operadores que eliminan actividades.

6.5.4. Adaptación de una composición WS-BPEL

Como se ha comentado anteriormente, WS-BPEL es un lenguaje reciente, por tanto, una de las limitaciones al trabajar con este lenguaje es que apenas existen ejemplos que se aproximen a la realidad.

La mayoría de los ejemplos que encontramos en artículos y en tesis son ejemplos clásicos, como el del Préstamos bancario que se incluye en el estándar [43].

Otros ejemplos disponibles en libros, por ejemplo, son muy sencillos cuya finalidad simplemente es explicar al lector las funcionalidades de este lenguaje a grandes rasgos. Sin embargo, ninguno suele profundizar en detalles específicos de este lenguaje

y, menos aún, ejemplos que cumplan una gran parte de las "restricciones estáticas" impuestas por el estándar [43].

Para poder implementar los operadores citados anteriormente, es necesario disponer de una composición WS-BPEL 2.0 a la que pueda aplicarse estos operadores. Se pensó en la posibilidad de modificar alguno de los ejemplos ya existentes, en lugar de crear uno nuevo. Se estudiaron varios ejemplos de composiciones, que comentamos a continuación:

Shipping Este ejemplo procede de la documentación del estándar OASIS [43]. Es relativamente simple. Implementa un servicio de envío de paquetes. Este proceso es asíncrono.

Auction También procede del estándar WS-BPEL 2.0 y es asíncrono. Consideramos una subasta de una casa. El proceso recoge la información del comprador y del vendedor de una subasta concreta, reporta los resultados de la subasta a un servicio de registro de subasta y, entonces, envía los resultados del registro al vendedor y al comprador.

Ordering Extraído del estándar OASIS. Utiliza una gran cantidad de elementos opacos, es decir, elementos que no especifican las operaciones concretas que se realizan en el servicio real. Por tanto, es un modelo abstracto y poco útil para nuestros propósitos.

Complex Data Exchange Procede de la Web de ActiveVOS [1]. Es un ejemplo incompleto y simple.

Loan Approval Consiste en un servicio de préstamo bancario. Los clientes envían peticiones de préstamo junto con su información personal y la cantidad requerida. El servicio de préstamo ejecuta un proceso simple que indicará con un mensaje si se concede o no el préstamo solicitado.

Loan Approval RPC Es una modificación de *LoanApproval* con llamadas a procedimientos remoto.

Meta Search Procede de BPELUnit [51]. Es un servicio de un motor de búsquedas por Internet. El cliente solicita búsquedas y el servicio devuelve al cliente los resultados encontrados.

Meta Search ForEach Emplea una nueva versión de MetaSearch que utiliza dos bucles *forEach* anidados, en vez de bucles *while*.

Marketplace También extraído de la Web de ActiveVOS. Es un proceso asíncrono que trabaja con dos peticiones: la del comprador y la del vendedor. Su funcionamiento básico consiste en enviar un mensaje de éxito si el precio solicitado por el vendedor es menor o igual a la cantidad ofrecida por el comprador. Ha sido mejorado por el grupo de investigación para que despliegue y se ejecute en ActiveBPEL, y con un BPTS más completo que incluye comunicaciones asíncronas: *receiveSendAsync* y *sendReceiveAsync*.

Market Place Flow Es la versión *MarketPlace* modificado con *flow*, que cancela el trato si la otra parte no responde en 5 segundos. Usa manejadores de compensación y suprime *joinFailure* para forzar a que el mutante sea distinto.

Sum Squares Este ejemplo básicamente realiza un sumatorio de variables al cuadrado, $\sum_i^n i^2$, que utiliza *forEach* paralelo y variables compartidas. *isolated* es mutado a "no".

Tac Service Es una imitación de la orden *tac* de UNIX, que invierte las líneas que se le envían.

Travel Reservation Service Ejemplo extraído de la Web de NetBeans [40]. Es un ejemplo definido con enlaces tanto síncronos como asíncronos. Este servicio permite a un cliente realizar la reserva de un viaje: el cliente podrá reservar un billete de avión, alquilar un vehículo y reservar un hotel.

Tras analizar cada uno de estos ejemplos, se consideró utilizar el ejemplo *TravelReservationService*, al ser uno de los ejemplos más completos de los estudiados junto con *MetaSearch*, y adaptarlo para que puedan aplicarse, al menos, los operadores **EMF**, **ASI**, **XMF**, **XMT**, **XTF**, **XER** y **XEE**. De esta forma, una vez modificado será la composición WS-BPEL más completa, actualmente, que mejor cubrirá las necesidades del grupo de investigación, para profundizar en el análisis de mutaciones.

Composición *TravelReservationService*

A continuación, detallamos el comportamiento de la composición WS-BPEL *TravelReservationService* de la que hemos partido. Más adelante, comentaremos cuáles son las modificaciones que se han realizado.

Básicamente, la composición original consiste en:

1. Si el cliente solicita un billete de avión, se manda la petición. Si se recibe una respuesta en 10 segundos, se copia la respuesta al itinerario actual y se sigue. De lo contrario, se envía una cancelación de la petición actual y continúa.
2. Si el cliente solicita el alquiler de un vehículo, se manda la petición. Si se recibe una respuesta en 10 segundos, se copia la respuesta al itinerario actual y se sigue. De lo contrario, se envía una cancelación de la petición actual y continúa.
3. Si el cliente solicita la reserva de un hotel, se manda la petición. Si se recibe una respuesta en 10 segundos, se copia la respuesta al itinerario actual. De lo contrario, se envía una cancelación de la petición.
4. Finalmente, se envía al cliente los resultados de su reserva, indicando si se ha podido realizar la reserva o no de cada servicio solicitado.

Adaptación de *TravelReservationService* para aplicar XEE

Para proponer posibles mejorar a esta composición para que sea aplicable el operador **XEE**, antes que nada debemos tener en cuenta cuál es la definición de dicho operador:

> El operador **XEE** elimina un elemento `onEvent` de un manejador de eventos, modelando el olvido a la hora de especificar un evento que puede recibir el proceso. (véase §2.2.4)

Puesto que *TravelReservationService* carece de un manejador de eventos, la primera modificación será añadir, al menos, un manejador de este tipo.

Que se necesite este manejador para poder mutar esta composición no justifica su incorporación. Por tanto, se añade la posibilidad de que el servicio de reserva de vuelos pueda notificar explícitamente a esta composición WS-BPEL que no ha podido realizar la reserva. El manejador de eventos se ocupará de las notificaciones de reservas fallidas: éste recibirá un mensaje en el que se indica que la reserva de avión ha fallado y se encargará de notificárselo al consumidor final.

Para consultar los problemas acontecidos y las soluciones propuestas, así como los detalles de implementación véase §6.6.6.

Adaptación de *TravelReservationService* para aplicar XMF

> El operador **XMF** elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos. Si al realizar la mutación sólo existe un elemento (`catch` o `catchAll`) en el manejador de fallos, éste también se eliminará. (véase § 2.2.4)

En esta composición original no se lanza ninguna fallo explícitamente, por tanto, no es necesario utilizar un manejador de fallos explícito, puesto que el lenguaje WS-BPEL propone un manejador de fallos por defecto.

Para poder aplicar este operador a la composición hemos tenido que considerar la posibilidad de lanzar fallos ante distintas situaciones, para que tenga sentido el uso de elementos `catch`, cada uno de éstos capturará el fallo que tenga un nombre concreto, y el uso de `catchAll` que recogerá los fallos que no hayan sido capturados por los elementos `catch`.

En el caso de la reserva de billete de avión se ha añadido la posibilidad de poder lanzar fallos atendiendo a:

- El cliente no facilita su nombre al solicitar la reserva.
- El cliente no facilita su dirección de correo electrónico al solicitar la reserva.
- El cliente no especifica la forma de pago del billete.

Por tanto, se han creado manejadores de fallos. También se ha modificado el proceso para que estos fallos puedan comunicarse al cliente.

Adaptación de *TravelReservationService* para aplicar XER

> El operador **XER** elimina una actividad `rethrow`, modelando el olvido de su inclusión en el manejador de fallos. (véase § 2.2.4)

Para ello, se ha mejorado el proceso relanzando fallos desde los elementos `catch` y `catchAll` de los manejadores de fallos.

Adaptación de *TravelReservationService* para aplicar XTF

> El operador **XTF** cambia el nombre del fallo lanzado por una actividad `throw` por otro del mismo ámbito, modelando la confusión a la hora de especificar el fallo a lanzar. (véase §2.2.4)

Puesto que se han añadido lanzamientos de fallos explícitamente a la composición, no hay que introducir nuevos cambios en ella para poder aplicar este operador.

Adaptación de *TravelReservationService* para aplicar XMT

> El operador **XMF** elimina la definición de un manejador de terminación. (véase §2.2.4)

La composición original no tiene ningún manejador de terminación. Por tanto, se ha añadido uno que especificará las acciones que se seguirán cuando se termine de manera forzada la ejecución de un ámbito o `scope` concreto, en nuestro caso, se hará que el proceso espere durante 2 segundos.

Adaptación de *TravelReservationService* para aplicar EMF

> El operador **EMF** modifica una expresión de fecha límite de dos formas: la sustituye por 0, lo que implica que la condición se cumple inmediatamente, y por la mitad del valor expresado inicialmente; esto tiene como objetivo verificar si el margen de seguridad especificado por la fecha es adecuado. Se puede aplicar a la actividad `wait` y al elemento `onAlarm` de la actividad `pick` y de los manejadores de eventos. (véase §2.2.2)

Para poder aplicarlo, se ha modificado el comportamiento del alquiler del vehículo y de la reserva del hotel como se especifica:

- Si el cliente solicita el alquiler de un vehículo, se manda la petición. Si se recibe una respuesta “antes de que se alcance la fecha límite” (en lugar de “en 10 segundos”), se copia la respuesta al itinerario actual y se sigue. De lo contrario, se envía una cancelación de la petición actual y continúa.
- Si el cliente solicita la reserva de un hotel, se manda la petición. Si se recibe una respuesta “antes de que se alcance la fecha límite” (en lugar de “en 10 segundos”), se copia la respuesta al itinerario actual. De lo contrario, se envía una cancelación de la petición.

Adaptación de *TravelReservationService* para aplicar ASI

> El operador ASI intercambia el orden de dos actividades dentro de una actividad `sequence`. (véase §2.2.3)

En este caso, no hay que realizar más modificaciones sobre esta composición, puesto que contiene varias actividades que podrán ser intercambiadas.

6.6. Implementación

6.6.1. Sistema de ejecución de GAmera

El sistema de ejecución de GAmera [13] es una simplificación y una extensión del que utiliza Takuan [46]. La Figura 6.14 muestra los diferentes pasos del sistema de ejecución.

El primer paso es el **paso de empaquetado**. En este paso, se toma el mutante y se prepara para que pueda ser desplegado en el motor WS-BPEL 2.0 de código abierto ActiveBPEL 4.1 [1], que lo ejecutará. La definición del proceso no cambia, simplemente se analiza para producir los ficheros de despliegue específicos del motor requeridos, que son empaquetados en un archivo que se enviará, más tarde, en el paso de ejecución al servicio Web de despliegue de ActiveBPEL. Estos ficheros son:

catalog.xml Un fichero que contiene todas las dependencias del proceso WS-BPEL.

process.pdd Fichero con la configuración necesaria para el despliegue del proceso en el servidor ActiveBPEL.

types.xml Agrupa las declaraciones de tipos y propiedades de todos los XML Schema y WSDL.

Como se ha comentado, todos estos ficheros, junto a los documentos WSDL, XML Schema originales y el conjunto de casos de prueba son empaquetados en un archivo “.bpr” que contiene toda la información y estructura necesaria para desplegar el proceso.

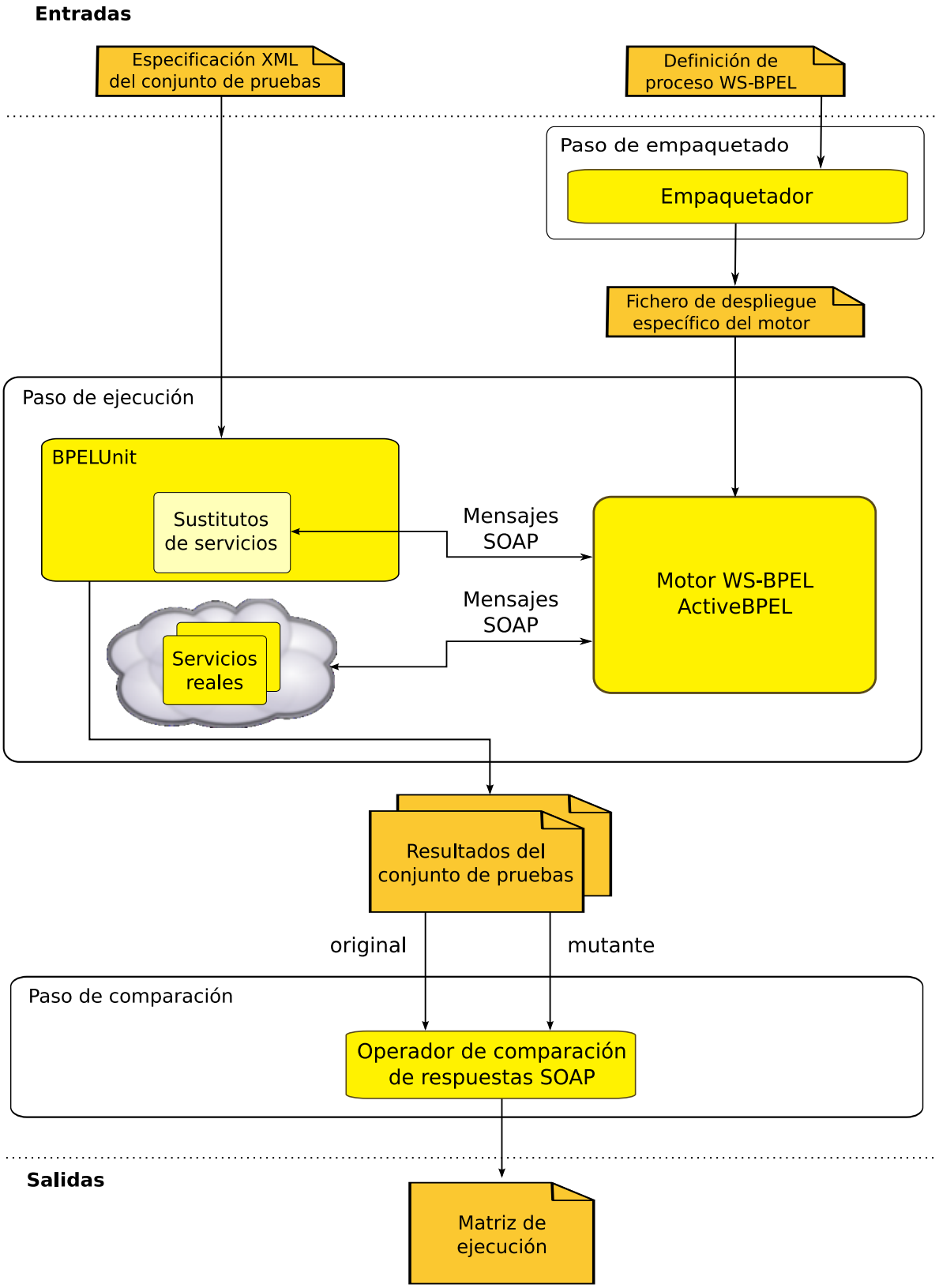

Figura 6.14: Sistema de ejecución de GAmera

El segundo paso es el **paso de ejecución**. Con los ficheros específicos del motor añadidos en el paso anterior, el mutante se desplegará, invocado una vez por cada caso de prueba. Los mensajes de respuesta de cada invocación son recogidos en el paso posterior. Estas tareas se llevan a cabo en GAmera gracias a BPELUnit [51].

El motor WS-BPEL, ActiveBPEL, permite ejecutar el proceso, sustituyendo las llamadas a servicios reales por otras declaradas por el usuario. Esta sustitución se puede desear hacer por tres motivos:

1. Los servicios reales pueden no estar disponibles, o no queremos interferir con ellos.
2. Los servicios reales son de pago, bloquean recursos, etc. - algo bastante común.
3. Queremos establecer de antemano cuáles van a ser las respuestas o peticiones de los servicios externos, de forma que podamos controlar en todo momento el entorno de prueba.

La biblioteca de prueba unitaria, BPELUnit, despliega el servicio, y actúa como cliente, invocando el servicio a probar, y como servidor para los *mockups*. Esta biblioteca es más compleja que las de otros lenguajes, ya que debe actuar:

1. Como “director”, preparando y monitorizando la ejecución del conjunto de casos de pruebas.
2. Como un servidor de *mockups*, manejando las peticiones entrantes y actuando como el servicio externo requerido por el proceso WS-BPEL.

Los pasos a realizar en este paso de ejecución son los siguientes:

1. Desplegar la composición WS-BPEL en el motor WS-BPEL. En este caso, se envía un archivo comprimido (.bpr) que contiene la lógica y algunos archivos específicos.
2. Se inicia el servidor que simula las respuestas externas.
3. Por cada caso de prueba:
   a) Se configura el servidor *mockup* con las respuestas predefinidas, o incluso su ausencia, si queremos simular un servicio inaccesible.
   b) Invocar el proceso que está siendo probado.
   c) Obtener los resultados.
4. Detener el proceso WS-BPEL.

Es en esta fase cuando BPELUnit, adquiere el control del proceso. Recibe como entrada un fichero .bpr con todos los ficheros necesarios, incluyendo uno con extensión .bpts que contiene la especificación del conjunto de casos de prueba, los *mockups* que reemplazarán a los servicios reales y el comportamiento del cliente simulado.

A partir de los *mockups* especificados, BPELUnit modifica el fichero BPEL asegurándose de que las llamadas realmente se realicen a los servicios simulados, tras lo cual indicará a ActiveBPEL que despliegue el proceso.

Una vez que el proceso esté listo, BPELUnit se conectará al mismo actuando como cliente, y ejecutando la operación indicada en el caso de prueba en el puerto correspondiente. En el caso de que el proceso inspeccionado realice alguna llamada a un servicio externo, será BPELUnit quien reciba la petición, y responderá en función a lo especificado en el fichero .bpts. Este proceso se repetirá por cada caso de prueba.

Finalmente, en el **paso de comparación** se compara la salida de cada mutante para cada caso de prueba con los del programa original para decidir si el mutante está muerto o sigue vivo. El operador de comparación es una estricta comparación uno-a-uno de sus mensajes de respuesta SOAP. A partir de estas comparaciones se obtendrá la matriz de ejecución, que permitirá calcular el *fitness* de cada individuo.

Los elementos de la matriz de ejecución pueden ser:

- $m_{ij} = 0$ si el mutante i no es muerto por el caso de prueba j.
- $m_{ij} = 1$ si el mutante i es muerto por el caso de prueba j.
- $m_{ij} = 2$ si el mutante i produce un error en la ejecución.

A continuación, se presenta un ejemplo de matriz de ejecución:

$$(m_{ij})_{M \times T} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 2 \end{pmatrix} \tag{6.1}$$

- Un mutante i está muerto, $\sum_{j=1}^{T} m_{ij} = 1$
- Un mutante i está vivo, $\sum_{j=1}^{T} m_{ij} = 0$
- Un mutante i es erróneo, $\sum_{j=1}^{T} m_{ij} = 2T$

6.6.2. Conversor individuo a mutante

En la Figura 6.15 se resume el proceso de generación de mutantes utilizando las hojas de estilos. Dado un programa original escrito en WS-BPEL y un individo concreto, identificado por el operador, el operando o instrucción y el atributo, se utilizará la hoja de estilo específica para realizar la mutación deseada. Se tomará el documento XML con la definición de proceso WS-BPEL original y se recorrerá cada nodo del documento, evaluándose si es la instrucción que se desea mutar. En caso afirmativo, se realizará la mutación tal y como esté indicado en esta hoja de estilo, en caso contrario, se copiará ese nodo. El resultado de este proceso se obtendrá en un nuevo documento XML, que será un mutante WS-BPEL de la composición original.

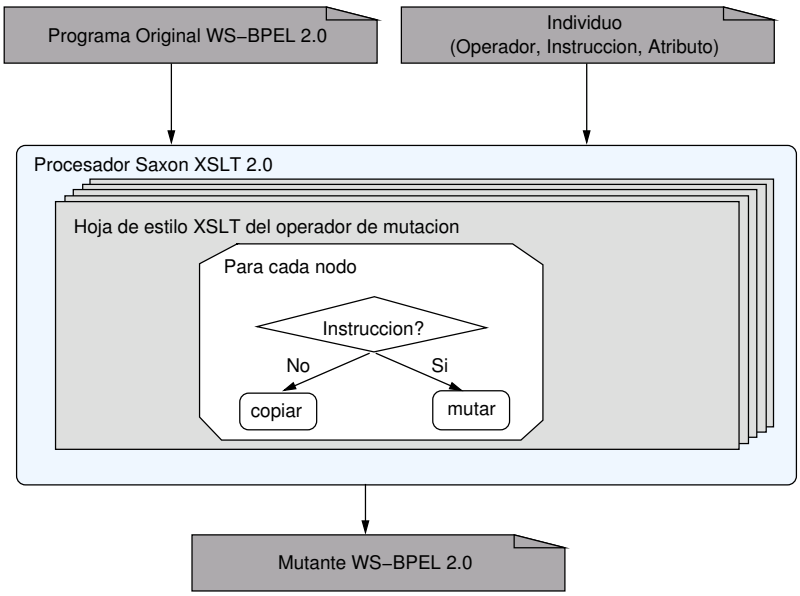

Figura 6.15: Conversor individuo a mutante

6.6.3. Implementación de operadores

Se ha utilizado el motor de XSLT Saxon 9.1-B (véase §B.1).

op-base.xsl

Esta hoja XSLT es base para todos los operadores de mutación definidos para WS-BPEL 2.0.

Si consideramos la hoja *op-base.xsl* como una clase de un lenguaje de programación orientado a objetos, implementa los atributos privados `operandIndex`, que indicará el operando que se desea mutar y `attribute`, que indicará el valor del atributo necesario para aplicar algunos operadores.

En cuanto a sus métodos o funciones públicos, tenemos los referentes a las acciones que se pueden llevar a cabo: `count()` para contar el número de operandos al que puede aplicarse un operador en concreto, `debugOperandList()` para depurar la lista de operandos, `debugChosenOperands()` para depurar cuáles son los operandos que se han elegido como candidatos, `apply()` para aplicar un operador sobre un operando en concreto y `maximum-attribute-value()`, para obtener cuál es el máximo valor del atributo de ese operador.

Por otro lado, también se definen varias plantillas. La plantilla `copy-tree()` que se encargará de copiar el árbol WS-BPEL y `collect-ignored-elems()` definido, por defecto, para que ningún elemento sea ignorado durante la copia.

Durante la copia del árbol se aplicarán las plantillas con modo `mutate-node`, que serán definidas en las hojas derivadas, y que se encargarán de mutar los nodos.

Además, se llamará a la plantilla `generate-operands()`, que estará definida en sus hojas derivadas, que se encargará de generar los operandos.

xpath-op-base.xsl

Esta hoja XSLT es base para todos los operadores que afectan a expresiones XPath.

Esta hoja define `mutate-node` para recorrer el árbol abstracto de sintaxis XPath, utilizando una plantilla interna `mutate-ast`.

Además, ofrece una plantilla denominada `generate-xpath-operands` para generar los operandos en el árbol abstracto de sintaxis de todas las expresiones XPath, mediante una consulta XPath. Cabe destacar el uso de `saxon:evaluate`, que es una función XPath que extiende al conjunto disponible en el estándar XPath 2.0.

delete-op-base.xsl

Esta hoja XSLT es base para todos los operadores que eliminan actividades o elementos de la definición de proceso WS-BPEL. Incluye la lógica requerida para eliminar el operando seleccionado y los enlaces (`links`) o dejar el operando seleccionado.

Las hojas de estilos que herenden de ésta únicamente necesitan definir la plantilla denominada `generate-operands`.

Define la plantilla `mutate-node` como vacía, por tanto, se aplicará la regla de mutación por defecto, que es la eliminación.

También define la plantilla `collect-ignored-elems`. Ésta es la que se encarga de realizar las eliminaciones atendiendo a los enlaces. Se distinguen tres casos distintos:

- El enlace se declara y se define dentro del operando. Por tanto, se elimina el operando y no es necesario hacer nada más.
- El enlace se declara fuera del operando pero se define dentro. En este caso, se elimina la declaración y se elimina el operando.
- El enlace se define fuera del operando. Ésta es la situación más compleja: se elimina el operando y el enlace.

Lista de operandos y de ignorados

Las hojas XSLT producen una **lista con los operandos** y valores adicionales. Esta lista es heterogénea, plana (puesto que no se puede anidar[1]) y los separadores son elementos de WS-BPEL. La subsecuencia correspondiente a un operando tiene este contenido:

- Nodo del árbol WS-BPEL del operando (parámetro `BPELNode`). Se distingue porque es un elemento que pertenece al espacio de nombres de WS-BPEL.
- Si corresponde, nodo raíz del árbol abstracto de sintaxis de XPath (parámetro `ast`). Es un elemento `uca:expression`.
- Si corresponde, cualquier información adicional que necesite un operando (parámetro `extra`). Es cualquier cosa que venga después del nodo del árbol WS-BPEL, antes del siguiente operando y no sea un elemento `uca:expression`. En los operadores que afectan a nodos XPath se introducirá en este parámetro el nodo del árbol abstracto de sintaxis correspondiente al operando, pero en los operadores **ASI**, **ISV** y **XTF** se introducirá el índice.

Las acciones `count`, `debug` y `apply` necesitan esta lista de operandos. Sin embargo, la acción `maximumAttributeValue` no la necesita.

La **lista de ignorados** es generada por `collect-ignored-elems` y se utiliza en la definición de `copy-tree`.

6.6.4. Implementación del operador XTF con una hoja de estilos

A continuación, presentamos la implementación del operador **XTF** utilizando una hoja de estilos, véase el Listado 6.1.

Ésta servirá como ejemplo para comprender cómo se han implementado el resto de operadores, que pueden consultarse en el CD adjunto.

Listado 6.1: Implementación del operador XTF

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!--
3    XTF (eXceptions) WS-BPEL mutation operator
4
5    Replaces the fault thrown by a throw activity .
6
7    Juan Boubeta Puig <juan.boubeta@uca.es>
8 -->
9 <xsl:stylesheet
```

[1]No se puede anidar la lista, ya que esto haría que se creara una copia de los nodos en cuestión y el operador identidad de XPath ("is"), que se usa para ver si el elemento actual es el operando deseado, dejaría de funcionar.

```
10    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
11    xmlns:xs="http://www.w3.org/2001/XMLSchema"
12    xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
13    xmlns:fn="http://www.w3.org/2005/xpath-functions"
14    xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
15    xmlns:uca="http://www.uca.es/xpath/2007/11"
16    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
17    version="2.0">
18
19  <xsl:import href="es/uca/webservices/mutants/operators/op-base.xsl"/>
20
21  <xsl:param name="action" as="xs:string"/>
22  <xsl:param name="operandIndex" as="xs:integer"/>
23  <xsl:param name="attribute" as="xs:integer"/>
24
25  <xsl:template name="generate-operands">
26    <xsl:for-each select="//bpel:throw">
27
28      <xsl:variable name="currentNode" select="."/>
29
30      <!-- If the current throw faultName can be replaced by at least one other faultName, then
31           list it as an operand. -->
32      <xsl:for-each select="@faultName">
33        <xsl:for-each select="uca:listFaultNames(., $currentNode)">
34          <xsl:sequence select="$currentNode"/>
35          <xsl:element name="uca:faultName">
36            <xsl:value-of select="."/>
37          </xsl:element>
38        </xsl:for-each>
39      </xsl:for-each>
40    </xsl:for-each>
41  </xsl:template>
42
43
44  <!-- Mutates faultName attributes by changing their value -->
45  <xsl:template match="*" mode="mutate-node">
46    <xsl:param name="extra"/>
47
48    <xsl:copy>
49      <xsl:copy-of select="@*[local-name() != 'faultName' and namespace-uri() =
           '']"/>
50      <xsl:attribute name="faultName">
51        <xsl:value-of select="$extra/text()"/>
52      </xsl:attribute>
53      <xsl:copy-of select="node()"/>
54    </xsl:copy>
55  </xsl:template>
56
57
58  <xsl:template match="bpel:throw" mode="collect-faultNames">
59    <xsl:sequence select="@faultName"/>
60  </xsl:template>
```

```
61
62    <xsl:template match="bpel:scope" mode="collect-faultNames"/>
63
64    <xsl:template match="*" mode="collect-faultNames">
65      <xsl:apply-templates mode="collect-faultNames"/>
66    </xsl:template>
67
68    <xsl:template match="text()" mode="collect-faultNames"/>
69
70
71    <!-- UTILITY FUNCTIONS -->
72
73    <!-- Lists faultNames which can replace a fault. These faultNames
74         are selected from the same scope the original fault was
75         defined. -->
76    <xsl:function name="uca:listFaultNames" as="xs:string*">
77      <xsl:param name="faultName" as="xs:string"/>
78      <xsl:param name="currentNode"/>
79
80      <xsl:variable
81          name="AllFaultNames"
82          select="fn:distinct-values(uca:listAllFaultNames($currentNode))"/>
83
84      <xsl:sequence select="fn:remove($AllFaultNames, fn:index-of($AllFaultNames,
            $faultName))"/>
85
86    </xsl:function>
87
88
89    <xsl:function name="uca:listAllFaultNames" as="xs:string*">
90      <xsl:param name="currentNode"/>
91
92      <xsl:apply-templates select="$currentNode/(ancestor::bpel:process|
            ancestor::bpel:scope)[last()]/*" mode="collect-faultNames"/>
93
94    </xsl:function>
95
96  </xsl:stylesheet>
```

En primer lugar, tenemos la declaración XML, indicando que se trata de un documento XML 1.0 codificado en UTF-8 (línea 1). Después tenemos al elemento raíz de toda hoja de estilos XSLT, `xsl:stylesheet`, bajo el espacio de nombres de XSLT. Estamos usando XSLT 2.0. (líneas 9 – 17).

El propio estándar XSLT define mecanismos para la herencia: a través del elemento `xsl:import` podemos importar todas las reglas de otro fichero XSLT, con menos precedencia que las actuales, de tal forma que podamos efectivamente especializar dicha hoja, redefiniendo y ampliando ciertos aspectos de forma parecida a la herencia del mundo orientado a objetos. Como podemos ver en la línea 19 estamos heredando de la hoja de estilos `op-base.xsl`.

A continuación, se describen los parámetros globales con los elementos `param`: `action` (línea 21) que indica la acción que se desea realizar y que es de tipo cadena, `operandIndex` (línea 22) que indica el operando concreto a mutar utilizando un entero, y `attribute` (línea 23) que indica el atributo y también debe ser un entero.

En las líneas 25 – 41 tenemos la primera plantilla de esta hoja. En XSLT, vamos básicamente recorriendo el árbol XML original a partir de la raíz, `/`, un nodo que representa al documento completo, y no a un elemento en particular. A cada paso aplicamos la plantilla `xsl:template` más específica cuya condición, indicada por el atributo `match`, se cumpla. Como su nombre indica esta plantilla definirá cómo se deben generar los operandos, que hereda de `op-base.xsl`.

Se recorrerá todo el documento XML de la composición WS-BPEL que se desea mutar. Para cada elemento `//bpel:throw` se realizan unas operaciones concretas, nótese que al utilizar el eje[2] `//`[3] se seleccionará aquellos elementos `throw` de WS-BPEL del nodo contexto y de sus sucesores.

Para cada nodo seleccionado se llevan a cabo las siguientes operaciones:

1. Se almacenará en una variable `currentNode` el nodo actual (línea 28).
2. A continuación se comprobará si el atributo `faultName` del nodo actual puede ser reemplazado por, al menos, otro atributo `faultName` de otro nodo (líneas 32 – 41). En caso afirmativo, se listarán estos atributos como operandos válidos para el nodo actual. Para ello se utiliza una función denominada `uca:listFaulNames` (líneas 73 – 86).

La función `uca:listFaulNames` recibe como parámetros el valor de un atributo `faulName` y el nodo actual, y devolverá una cadena con todos los `faulNames` distintos que pueden reemplazar al atributo del elemento `fault` de WS-BPEL original. Solamente se seleccionarán los `faultName` que se encuentren en el mismo ámbito en el que el `fault` original se ha definido.

Para asegurarnos que realmente sólo consideraremos operandos `faulName` pertenecientes al mismo ámbito del elemento `throw` que se desea mutar se utiliza una función auxiliar `uca:listAllFaultNames` (líneas 89 – 94) que devolverá estos operandos en una cadena y que recibe el nodo actual. Esta función utilizará las siguientes plantillas con el modo `collect-faulNames` (líneas 58 – 68). Cambiando entre modos podemos cambiar entre distintos conjuntos de plantillas fácilmente, y procesar varias veces el mismo nodo (véase §B.5). Se aplicará la plantilla automáticamente para todo elemento descendiente del ámbito al que pertenece el elemento `throw` (líneas 64 –

[2]Un eje en XPath indica la relación entre nodos. Se usa con la forma general `eje:testDeNodo[predicado]`. Una ruta en XPath está formada por 0 o más ejes separados por `/`. El `testDeNodo` indica el nombre del nodo que se desea seleccionar de todos los de la secuencia, es decir, es un filtro de los nodos de la secuencia. En los predicados se puede usar funciones que devuelvan un valor booleano que indica si el nodo pasa el filtro o no.

[3]Abreviatura del eje `/descendant-or-self/node()/`.

66), se cumple la condición `match="*"`, y se aplicarán las plantillas que tengan dicho modo. Como podemos comprobar estas plantillas están vacías (líneas 62 y 68), a excepción de la que tiene la condición `match="bpel:throw"` (líneas 58 – 60), que seleccionará los atributos `faultName` en una secuencia.

Con esta implementación, se recorrerán todos los elementos descendientes desde el `scope` del elemento `throw` que se desea mutar del documento de la composición WS-BPEL original, expresado como `$currentNode/(ancestor::bpel:process|ancestor::bpel:scope)[last()]/*`. Mientras que se encuentre algún elemento descendiente `bpel:throw` se seleccionará su atributo `faultName`. En el momento en que se recorra un elemento `bpel:scope` se aplicará la plantilla con `match="bpel:scope"` (línea 62) que estará vacía. Desde este momento, ya no se seguirán aplicando estas plantillas con modo `collect-faultNames` y tendremos únicamente los `faulName` pertenecientes al ámbito deseado.

La segunda plantilla (líneas 44 – 55) es la responsable de realizar la mutación específica del operador **XTF**. Se utilizará automáticamente al tener el atributo `match`. La condición de esta plantilla es `match="*"`, por tanto, coincide con cualquier elemento (que no nodo: excluye a los atributos y nodos de texto, por ejemplo) en el modo `mutate-node`.

La mutación recibirá el parámetro `extra` que lo utilizará para seleccionar la alternativa por la que se mutará.

Cabe destacar que la instrucción `xsl:copy` (líneas 48 – 54) copia el elemento contexto del documento fuente al documento resultante; no copia los hijos, descendientes o atributos del nodo contexto, únicamente copia el nodo contexto y su espacio de nombre (si es un elemento). Para una copia en profundidad se utilizará `xsl:copy-of`, es decir, al copiar un nodo, también se copiarán sus descendientes.

Con `@*[local-name() != 'faultName' and namespace-uri() == '']` se copian todos los atributos menos el que se llama `faultName` y no está en ningún espacio de nombres (el que queremos cambiar de valor).

6.6.5. Optimización de los operadores ASI, ISV y XTF

A continuación se presenta cuál era la situación actual y el motivo por el que se decidió optimizar los operadores **ASI**, **ISV** y **XTF**.

Para los operadores que tienen un rango variable en el atributo, puede que muchos individuos del algoritmo genético produzcan el mismo mutante. Por ejemplo, si **ISV** tiene 1 opción en un operando y 2 en otro, el rango máximo del atributo sería 2, pero (ISV, 1, 1) produciría el mismo mutante que (ISV, 1, 2), ya que en el operando 1 realmente sólo hay una opción.

Esto da lugar a que a partir de composiciones WS-BPEL de cierta complejidad se generen una gran cantidad de mutantes repetidos. Esto conlleva que tenga demasiado impacto en el tiempo de ejecución y, por ello, no deba ignorarse.

Para resolverlo, una propuesta que se llevó a cabo dentro del grupo fue calcular el resumen SHA1 del *.bpel* producido y no volver a ejecutar un mutante que tenga un SHA1 ya conocido. Se implementó en un guión Python que simula una prueba tradicional de mutaciones (classify-mutants) y se consiguió mejorar los tiempos.

Esta solución era un parche temporal y, por ello, el grupo decidió proponer una nueva para este problema. Las hojas XSLT producían una lista plana con operandos y valores adicionales que se decidió modificar. El resultado de esta modificación se recoge en §6.6.3.

Como consecuencia de esta optimización, el cálculo de los operandos en **ASI**, **ISV** y **XTF** ahora repetirá cada operando tantas veces como alternativas reales haya.

Además, se ha modificado el operador **ASI** para que sólo considere como operandos el intercambio de una actividad con una posterior.

6.6.6. Composición *TravelReservationService* adaptada

Tras llevar a cabo todas las modificaciones sobre la composición WS-BPEL original *TravelReservationService*, especificadas en la fase de diseño, hemos obtenido una composición a la que pueden aplicarse, al menos, los operadores **EMF**, **ASI**, **XMF**, **XMT**, **XTF**, **XER** y **XEE**.

La Figura 6.16 muestra gráficamente esta composición modificada. En la Figura 6.17 se detalla la actividad `if` denominada `HasAirline`. En la Figura 6.18 se detalla la actividad `if` denominada `HasVehicle` y en la Figura 6.19 la denominada `HasHotel`.

A continuación, se presentan los dos problemas más relevantes que hemos encontrado al modificar dicha composición.

La mayoría de estos problemas han surgido debido a la gran cantidad de restricciones estáticas que impone dicho lenguaje y las ambigüedades que presenta el estándar OASIS. Para profundizar más en estos detalles puede consultarse §A.

Introducción de un manejador de eventos

La mayoría de los ejemplos que se encontraron en la bibliografía de ejemplos de composiciones WS-BPEL sobre manejadores de eventos eran muy simples e implementaban estos manejadores de eventos de forma que una vez que recibían un mensaje salían abruptamente del proceso sin dar ninguna respuesta.

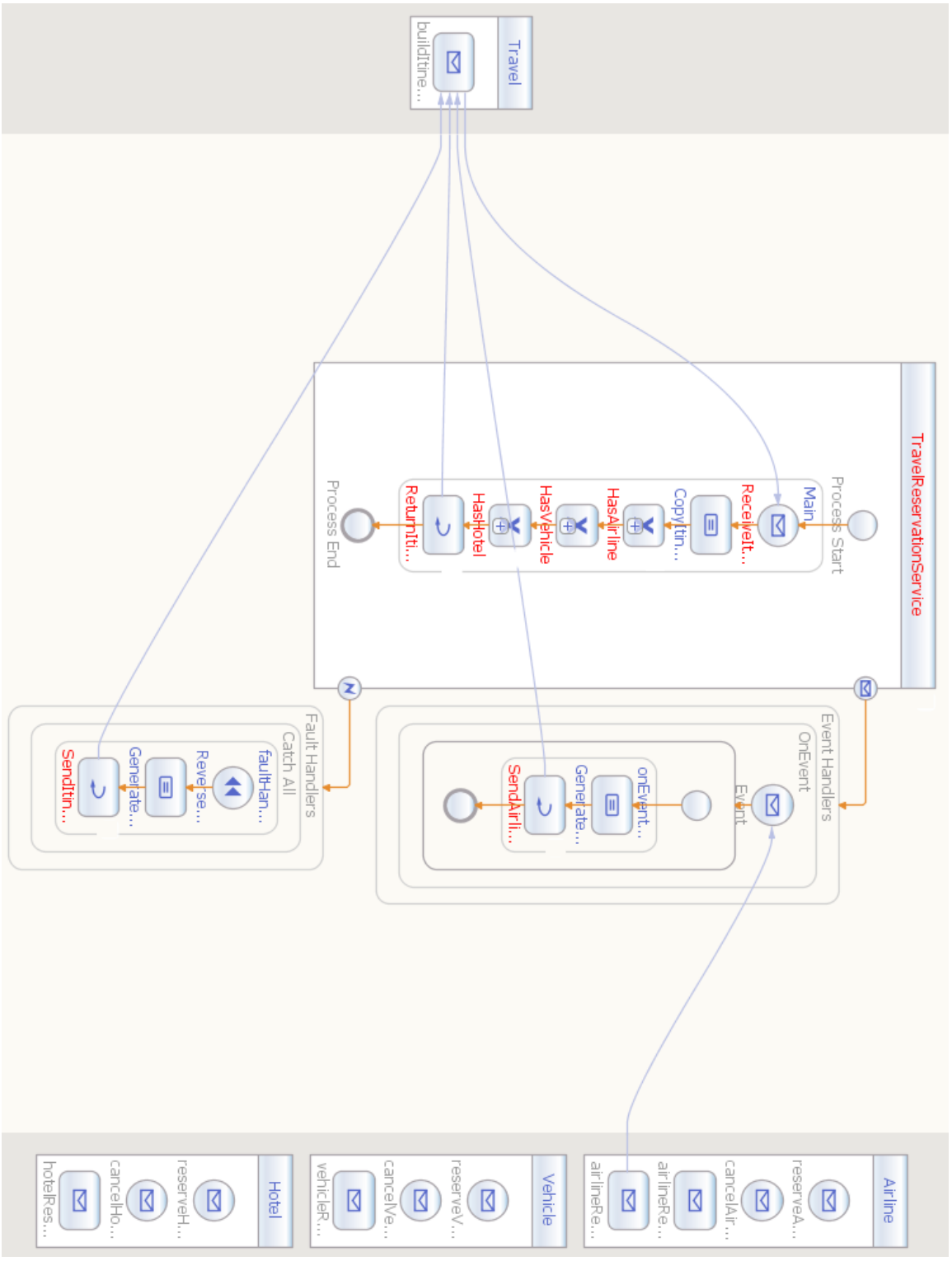

Figura 6.16: Vista gráfica del proceso *TravelReservationService*

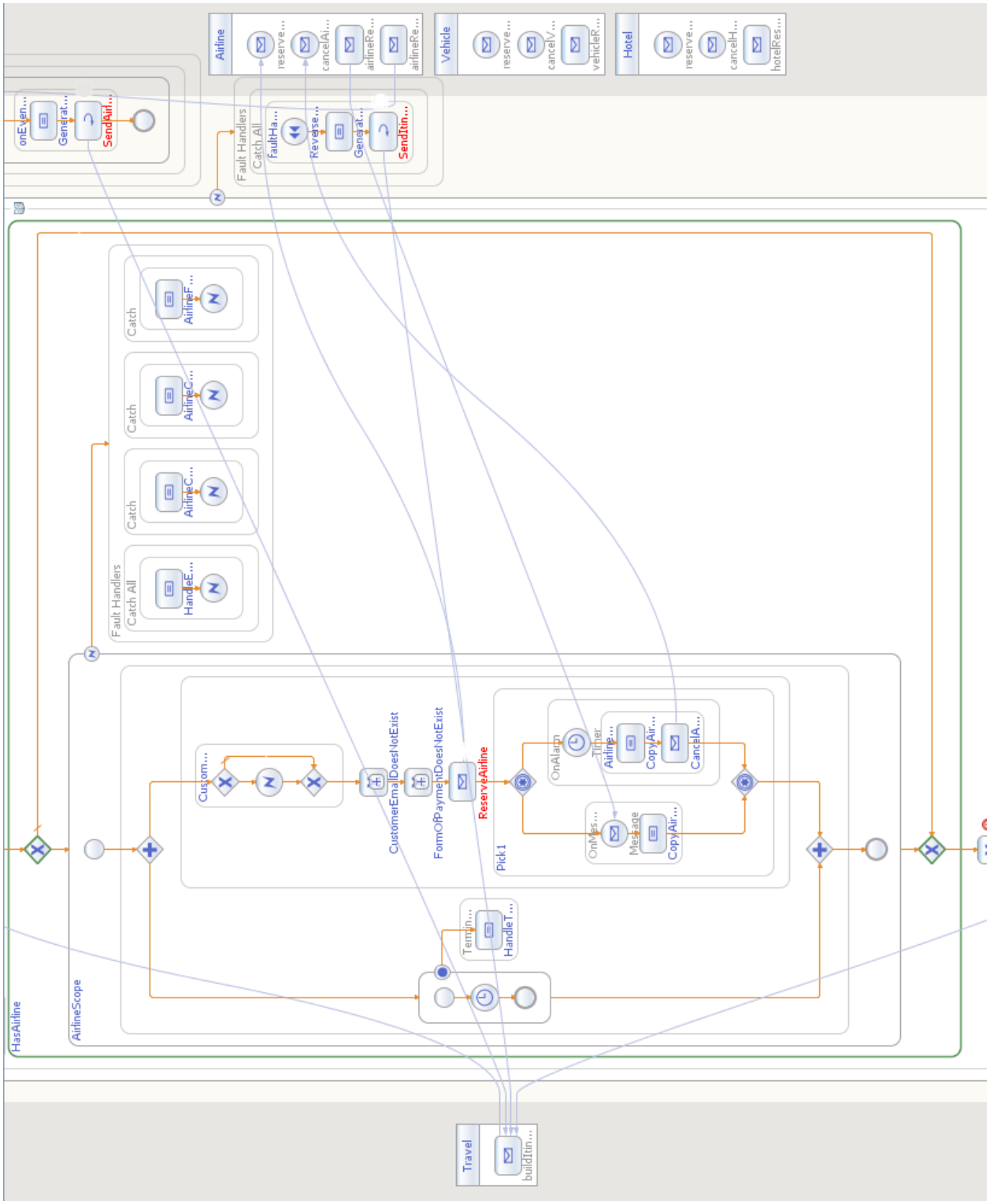

Figura 6.17: Vista gráfica de la actividad `if`, denominada `HasAirline`, de *TravelReservationService*

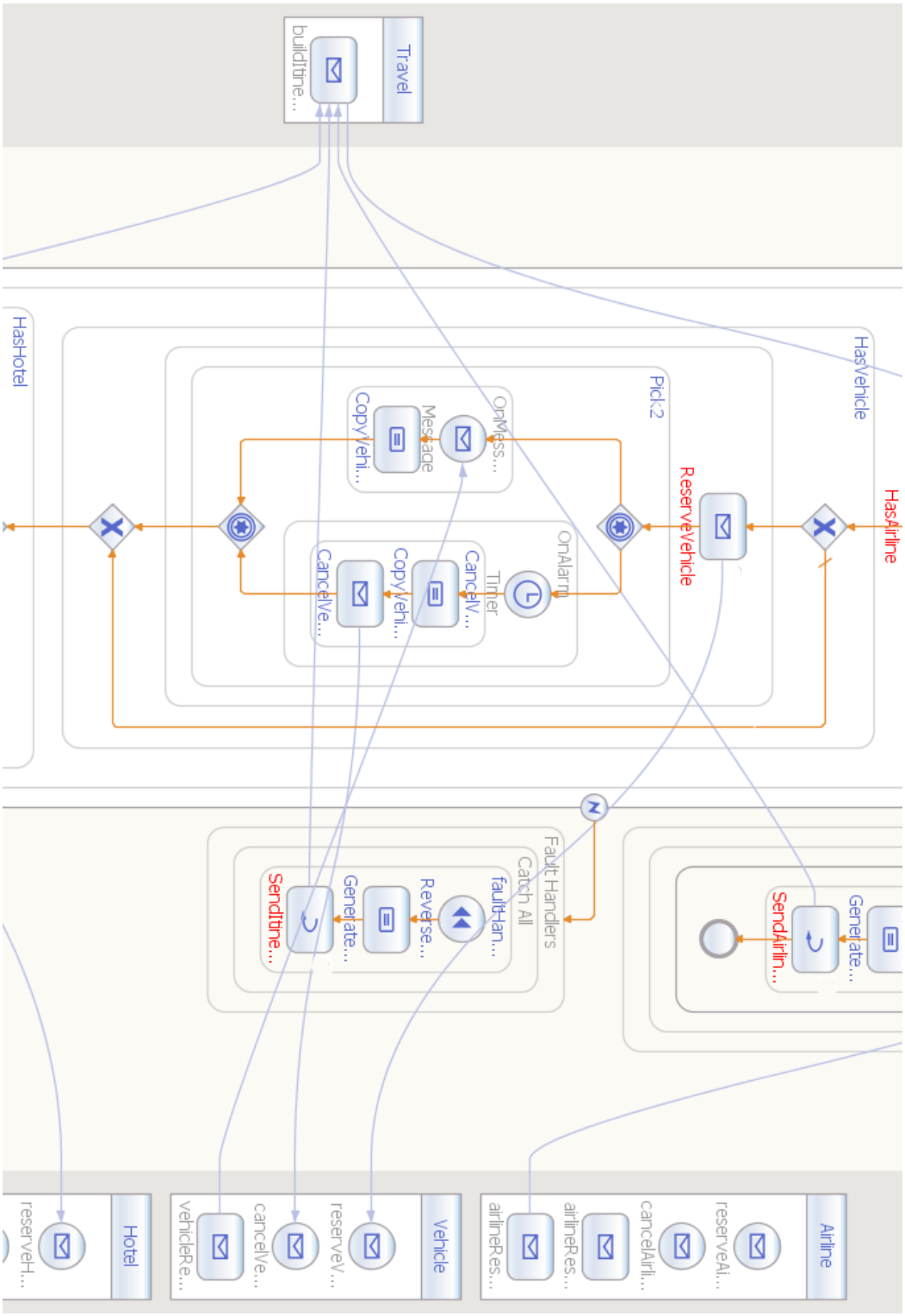

Figura 6.18: Vista gráfica de la actividad `if`, denominada `HasVehicle`, de *TravelReservationService*

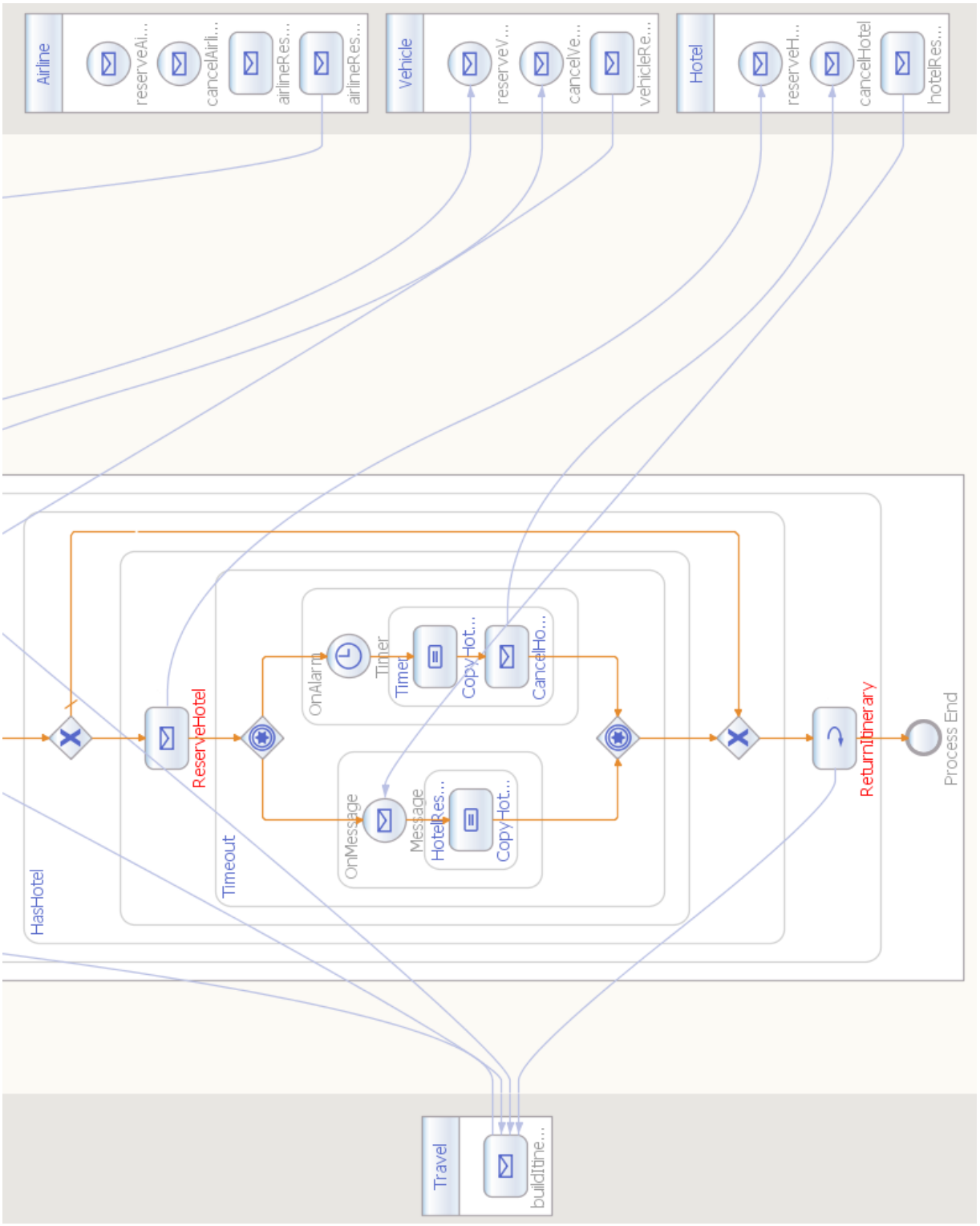

Figura 6.19: Vista gráfica de la actividad `if`, denominada `HasHotel`, de *TravelReservationService*

Básicamente, existen tres formas de implementar el *scope* o ámbito de un manejador de eventos:

1. Salir abruptamente de la composición. Esta opción se descartó puesto que la interacción es cliente-servidor y, por tanto, el cliente esperará una respuesta.
2. Lanzar una excepción dentro del ámbito del manejador de eventos y capturarla en el manejador de fallos del proceso. Esta opción complicaría inncesariamente la composición.
3. Responder al cliente con un fallo SOAP indicando el problema ocurrido, por ejemplo.

En primer lugar, se implementó la segunda opción pero no fue viable, ya que la composición se quedaba bloqueada sin hacer nada. Por tanto, se decidió implementar la tercera opción. Se enviará un mensaje al cliente, mediante una actividad `reply`, informándole que el vuelo no ha podido ser reservado. No debe confundirse con que la reserva haya sido cancelada, ya que esto se hace automáticamente si no se recibe una respuesta en 10 segundos.

Al introducir este manejador de eventos habrá que comprobar que la secuencias de mensajes es la siguiente:

1. El cliente envía el itinerario que quiere y se queda esperando.
2. El itinerario contiene una referencia a un vuelo, así que la composición WS-BPEL pide al servicio de reserva de vuelos que reserve el vuelo, a través de la operación `reserveAirline`. El *mockup* del servicio externo recibe el mensaje y sigue sin más.
3. El servicio de reserva de vuelos notifica a través del callback de la composición que la reserva no ha podido realizarse.
4. El manejador de eventos de la composición se activa y actúa enviando un mensaje de fallo al usuario indicándole que no ha podido hacerse la reserva.

A continuación citamos los dos problemas más importantes que surgieron a la hora de implementar el manejador de eventos:

> *[travel.bpr] [process.pdd] WARNING: Uncorrelated onEvent may not work correctly if more than one process can be waiting for the Partner Link. Add a correlation set or engine-managed correlation policy assertion.: in process.pdd*

En los manejadores de eventos es fundamental el concepto de correlación: si se tienen muchas instancias de una composición y llega un mensaje para un manejador de eventos, en principio no se sabrá a cuál de esas instancias se debe despachar.

Para saberlo, normalmente lo que se hará es poner en el mensaje entrante un identificador que indique con cuál instancia se corresponde ("se correla"). Para ello, se debe definir un `correlationSet` que emplea una propiedad WS-BPEL para obtener

el valor del identificador de correlación de cada uno de los tipos de mensajes "interesantes", y usarlo desde los `<receive>`, `<reply>`, `<onEvent>` y demás que lo necesiten. Estas propiedades y las expresiones XPath que sacan esos identificadores se declaran en el WSDL de la composición, con elementos `bpws:propertyAlias`. En esta composición se usa el elemento `ota:UniqueID`.

Por tanto, lo que se ha hecho ha sido correlar el `onEvent` usando el `correlationSet` que ya existía en la composición original, añadiendo un nuevo `propertyAlias` del mensaje de fallo de la reserva de avión, que ahora envía el identificador de correlación.

Los conjuntos de correlación han de ser iniciados por alguna actividad. Hay que decirle al motor que no lo inicia el `onEvent` (que daría fallo de validación y además no tendría sentido), sino el `<receive>` que recibe el itinerario. Para ello, se ha de usar el atributo `initiate` de la forma apropiada: "no" en el `<onEvent>` y "yes" en el `<receive>` inicial.

El siguiente problema fue:

> *[travel.bpr] [process.pdd] The reply activity has no corresponding receive with partner link 'Travel', operation 'buildItinerary', and message exchange ".: in process.*

Para solucionar esto tuvimos que recurrir al estándar WS-BPEL e intentar concretar las ambigüedades que éste presenta:

> *"If a <reply>activity cannot be associated with an open IMA by matching the tuple partnerLink, operation, and messageExchange then a WS-BPEL processor MUST throw a bpel:missingRequest fault on the <reply>activity. Since conflicting requests are rejected at the time the IMA begins execution there cannot be more than one corresponding IMA at the time a <reply>activity is executed."* (véase [43, 10.4.1])

De aquí deducimos que a todo `<reply>` en una operación petición-respuesta (se identifican porque tienen `<input>` y `<output>` en su WSDL) le tiene que corresponder su actividad de mensaje entrante ("Inbound Message Activity"). ActiveBPEL intenta con esta validación estática que no se produzcan fallos `bpel:missingRequest` en mitad de la ejecución del proceso.

Normalmente no tenemos problemas en este sentido, ya que con los identificadores de intercambios de mensajes generados por omisión es suficiente.

Sin embargo, según el párrafo siguiente, estos identificadores generados por omisión parecen ser distintos internamente para cada instancia de un `<scope>` de `onEvent` y de un `<forEach>` paralelo y para el ámbito implícito de `<process>`:

> *"If the messageExchange attribute is not specified on an IMA or <reply>then the activity's messageExchange is automatically associated with a default messageExchange with no name. Default messageExchange's are implicitly declared by the <process>and the immediate child scopes of <onEvent>and the parallel form of <forEach>. Other occurrences of <scope>activities do not provide a default messageExchange. Default messageExchange instances, just like non-default messageExchange elements, are created each time the scope declaring the default messageExchange is executed. For example each time an <onEvent>is executed (i.e. when a new message arrives for processing) it creates a new default messageExchange instance associated with each <onEvent>instance. This allows a request-response <onEvent>event handler to receive messages in parallel without faulting or explicitly specifying a messageExchange. Similarly it allows the use of <receive>-<reply>or <onMessage>-<reply>pairs in the parallel form of <forEach>without the need to explicitly specify a messageExchange."*

Por tanto, hay que definir manualmente un `messageExchange` (al que hemos llamado "M1") y asignárselo a los `<receive>` y `<reply>` de la operación `buildItinerary` del partner link `Travel`. De esta forma, se soluciona el problema y se despliega correctamente.

Introducción de un manejador de terminación

Para lanzar un manejador de terminación, se tiene que activar un manejador de fallos, según el siguiente párrafo. El manejador de fallos desactiva los manejadores de eventos y termina de manera forzada las actividades que estuvieran ocurriendo en ese momento. El manejador de terminación del usuario se ejecutará en ese momento.

> *"The behavior of a fault handler for a scope C begins by disabling the scope's event handlers and implicitly terminating all activities enclosed within C that are currently active (including all running event handler instances)."* (véase [43, 12.6])

Sin embargo, más adelante, el estándar dice lo siguiente:

> *"Forced termination for a scope applies only if the scope is in normal processing mode. If the scope has already invoked fault handling behavior, then the termination handler is uninstalled, and the forced termination has no effect. The already active fault handling is allowed to complete. If the fault handler itself throws a fault, this fault is propagated to the next enclosing scope."*

Es decir, que si el `scope` se mete en un manejador de fallos (aunque sea el que viene por defecto) no ejecutará su propio manejador de terminación. Esto implica que se debe lanzar el fallo desde un `scope` de nivel superior y capturarlo, de forma que una

de las actividades que termine sea otro `scope` con el `terminationHandler` que se desee invocar.

Para solucionar esto, hemos introducido un `<wait>` dentro de un `<flow>` paralelo al código normal. Este `<wait>` está dentro de un `<scope>` con el `terminationHandler` que se desea ejecutar.

6.7. Pruebas y validación

6.7.1. Plan de pruebas

Alcance

Se han probado tantos los operadores de mutación para WS-BPEL implementados, como los resultados obtenidos al ejecutar los casos de pruebas definidos para la composición WS-BPEL, denominada *TravelReservationService*, sobre la composición original y sus mutantes.

Tiempo y lugar

Las pruebas se han realizado durante el desarrollo del PFC. Conforme se implementaba un operador y se adaptaba dicha composición al nuevo operador se ejecutaban las pruebas.

A partir de los resultados de estas pruebas se realizaban nuevas iteraciones, en el caso de que éstas fallasen o, en caso contrario, se comenzaba la implementación de un nuevo operador.

Finalmente, se volvieron a ejecutar todas las pruebas automáticas para comprobar que tanto los operadores implementados como la adaptación de la composición se había llevado a cabo con éxito.

Naturaleza de las pruebas

Como se ha comentado en varias ocasiones, la totalidad de las pruebas usadas son pruebas de caja blanca.

6.7.2. Definición de casos de prueba unitarias para los operadores

Para las pruebas unitarias se usa una fábrica especializada *TestOperatorFactory* que genera versiones especializadas para pruebas de los operadores de mutación.

Se ha implementado *TestXSLTStylesheet*, que extiende *XSLTStylesheet*, que recoge todos los errores, errores graves y avisos en listas, y permite que el usuario los consulte. Los errores y los errores graves son lanzados como *TransformerExceptions*. Sólo se almacenan los errores, errores graves y avisos de la última transformación, ya que sus listas se borran en cada transformación.

Se ha creado una clase en Java denominada *BPELMutationTestCase* que define las funciones necesarias para definir las pruebas unitarias de cada composición WS-BPEL. Cada composición deberá definir una clase Java que extenderá a *BPELMutationTestCase* y que contendrá los distintos métodos de esta clase, que se evaluarán para comprobar si el funcionamiento de cada uno de ellos se comporta como se espera.

Los casos de prueba se han definido utilizando el framework JUnit[4]. Estos casos de prueba comprueban que los operadores se han implementado correctamente, atendiendo a:

1. Número de operadores detectados en la composición.
2. Modificaciones introducidas al ejecutar los operadores.
3. Comprobaciones de que las salidas de la composición original y la de los mutantes son diferentes.

A continuación, mostramos en el Listado 6.2 los casos de pruebas unitarios definidos para el operador **XTF** implementado. El resto de casos de prueba podrán obtenerse del CD adjunto.

Listado 6.2: Casos de prueba unitarios definidos para **XTF**

```
1  public class TravelReservationServiceProcessTest extends BPELMutationTestCase {
2
3      private final static String TEST_SUITE = "test_processes/
           travelReservationService/travelTest.bpts";
4
5      public TravelReservationServiceProcessTest() {
6          super("test_processes/travelReservationService/
               TravelReservationService.bpel");
7      }
8
9
10     /* XTF TESTS */
11
12     @Test
```

[4] JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck para hacer pruebas unitarias de aplicaciones Java.

```
13      public void xtfOperandCount() throws Exception {
14          assertOperandCount(6, getOp("XTF"));
15      }
16
17      @Test
18      public void xtfAttributeRange() throws Exception {
19          assertMaximumDistinctAttributeValueIs(getOp("XTF"), 1);
20      }
21
22      @Test
23      public void xtfMutateCustomerPersonNameFaultName() throws Exception {
24          applyAndAssertEqualsXPath(getOp("XTF"), 1, 1, "(//bpel:throw)[1]/@faultName", "
                AirlineCustomerEmailDoesNotExist");
25          applyAndAssertEqualsXPath(getOp("XTF"), 1, 1, "(//bpel:throw)[2]/@faultName", "
                AirlineCustomerEmailDoesNotExist");
26          applyAndAssertEqualsXPath(getOp("XTF"), 1, 1, "(//bpel:throw)[3]/@faultName", "
                AirlineFormOfPaymentDoesNotExist");
27      }
28
29      @Test
30      public void xtfCustomerPersonNameFaultNameMutantIsDifferent() throws Exception {
31          assertSuccessfulMutationIsDifferent(TEST_SUITE, getOp("XTF"), 1, 1);
32      }
33
34  }
```

En la línea 1 puede verse cómo se ha creado una clase denominada *TravelReservationServiceProcessTest* que extiende a *BPELMutationTestCase*.

A continuación se especifica la ruta en la que se encuentra el conjunto de casos de prueba definidos para la composición WS-BPEL (línea 3). En las líneas 4 – 7 se crea el constructor de esta clase.

A partir de aquí, deben definirse los casos de prueba para cada uno de los operadores. Desde la línea 10 en adelante tenemos los casos de prueba que se han definido para el operador **XTF**.

El caso de prueba `xtfOperandCount()` se encarga de comprobar si el número de operadores detectados en la composición es 6. Es decir, hemos contado que existen 6 operandos posibles para el operador **XTF** sobre la composición *TravelReservationService.bpel* y, por tanto, si la hoja XSLT de dicho operador está implementada correctamente el número de operandos que devuelva será el mismo que el que esperamos.

El caso de prueba `xtfAttributeRange()` comprueba si el máximo valor del atributo de este operador es 1.

El caso de prueba `xtfMutateCustomerPersonNameFaultName()` comprueba si son correctas las modificaciones introducidas al ejecutar este operador, es decir, si se realizan correctamente las mutaciones tal y como se espera.

En este caso en concreto, se comprueba que al aplicar el operador **XTF** con valor del operando 1 y valor del atributo 1 (valor por defecto del atributo), debe mutarse el nombre del fallo a lanzar `CustomerPersonName` de la actividad `bpel:throw` por `AirlineCustomerEmailDoesNotExist`. Con este caso de prueba verificaremos que realmente se modificará el nombre de fallo de la primera actividad `bpel:throw` y que, sin embargo, el resto de actividades `bpel:throw` no se habrán modificado.

El último caso de prueba denominado `xtfCustomerPersonNameFaultName MutantIsDifferent()` comprueba que la salida de la composición WS-BPEL original, `TravelReservationService`, y la del mutante, en el que se ha modificado el nombre de fallo comentado anteriormente, es diferente.

6.7.3. Definición de casos de prueba para la composición WS-BPEL

La definición de los casos de prueba para las composiciones WS-BPEL se lleva a cabo utilizando BPELUnit[5].

Para cada conjunto de casos de prueba que se define para una composición WS-BPEL concreta se creará un fichero *.bpts*, un fichero XML que define qué proceso se va a probar y cómo. Habrá que comprobar si contiene los casos de prueba suficientes para matar a todos los mutantes no equivalentes que produce la composición WS-BPEL original.

En general, las pruebas unitarias han de comprobar tanto que funciona bien cuando tenga que ir bien, como que falle de la forma esperada cuando algo vaya mal.

En este caso se ha modificado el fichero *travelTest.bpts* para la composición WS-BPEL denominada *TravelReservationService*. A continuación, mostraremos los aspectos más importantes de la implementación de estos casos de prueba. El fichero completo podrá obtenerse del CD adjunto.

Listado 6.3: Estructura de un fichero *.bpts*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tes:testSuite
3      xmlns:trs="http://www.sun.com/javaone/05/TravelReservationService"
4      xmlns:ota="http://www.opentravel.org/OTA/2003/05"
5      xmlns:tes="http://www.bpelunit.org/schema/testSuite"
6      xmlns:ves="http://www.sun.com/javaone/05/VehicleReservationService"
7      xmlns:aes="http://www.sun.com/javaone/05/AirlineReservationService"
8      xmlns:hes="http://www.sun.com/javaone/05/HotelReservationService">
9
10   <tes:name>TravelReservationService</tes:name>
11   <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
12
```

[5] BPELUnit es un framework para realizar pruebas de unidad de composiciones WS-BPEL creado por Philip Mayer [38].

```
13    <tes:deployment>
14      <tes:put name="TravelReservationService" type="activebpel">
15        <tes:wsdl>TravelReservationService.wsdl</tes:wsdl>
16        <tes:property name="BPRFile">travel.bpr</tes:property>
17      </tes:put>
18      <tes:partner name="Airline" wsdl="AirlineReservationService.wsdl"/>
19      <tes:partner name="Hotel" wsdl="HotelReservationService.wsdl"/>
20      <tes:partner name="Vehicle" wsdl="VehicleReservationService.wsdl"/>
21    </tes:deployment>
22
23    <tes:testCases>
24     ...
25     ...
26    </tes:testCases>
27
28  </tes:testSuite>
```

Los ficheros con extensión `bpts` son ficheros XML. Como todos los documentos XML, incluye un elemento raíz donde se especifican los espacios de nombre (líneas 1 – 8). En este caso, el prefijo `tes` está asociado al espacio de nombres de BPELUnit. Los demás son prefijos propios de esta composición.

El primer bloque de información (líneas 10 – 21) incluirá datos sobre el proceso que será desplegado: nombre, URL, *partners* (servicios a los que se conecta), etc.

tes:name Un nombre identificativo para el proceso.

tes:baseURL La URL sobre la que se desplegará. Normalmente, el valor del ejemplo es el que siempre se deberá usar.

test:deployment La información sobre el despliegue del proceso.

- **tes:put** El proceso a desplegar. Se especifica su nombre, el motor que lo ejecutará, el fichero WSDL que describe el servicio, y el fichero `bpr` que contiene la composición.
- **tes:partner** Los servicios externos a los que se conectará, junto a su descripción WSDL. Estos servicios externos pueden sustituirse por *mockups* cuando se especifique el caso de prueba. El atributo `name` debe identificar unívocamente al *partner*, ya que será el nombre que se usará después para sustituirlo por uno simulado.

El último bloque (líneas 23 – 26) son los casos de prueba. Se especifican como un elemento raíz `<tes:testCases>` en el que se anidan elementos `<tes:testCase>` para cada caso de prueba.

A continuación se describen los elementos que aparecen en los casos de prueba:

tes:testCase El elemento para definir un caso de prueba. El atributo `name` es un identificador único del caso de prueba. Un caso de prueba puede basarse en uno

anterior, definido con el atributo `basedOn`. Si el atributo `abstract` tiene el valor "yes", entonces el caso de prueba no puede ser ejecutado. Por último, `vary` especifica si se debe ejecutar más de una vez el mismo caso de prueba cambiando tiempos de respuesta, etc.

tes:clientTrack Define la petición que se enviará desde el cliente simulado. Cada caso de prueba debe contener un `clientTrack` y un número de `partnerTrack`.

tes:partnerTrack Se utiliza para definir un *mockup*. La estructura es prácticamente idéntica al `tes:clientTrack`. El atributo `name` define qué servicio externo está siendo reemplazado. Cada caso de prueba debe contener un único `partnerTrack` por cada *partner*.

Tanto los `clientTrack` como los `partnerTrack` pueden contener una secuencia de actividades soportadas por BPELUnit:

tes:sendOnly Esta actividad envía un mensaje simple.

tes:receiveOnly Esta actividad espera por un mensaje simple y lo verifica.

tes:sendReceive En el caso de `clientTrack` el cliente del proceso mandará una petición síncrona. Los atributos especifican a qué servicio y puerto se enviará la petición, y qué operación se usará. En el bloque `tes:send` los datos incluidos dentro de `tes:data` se enviarán al proceso. Estos datos seguirán la estructura definida por el propio proceso. El atributo `fault` a "false" indicará que no debe simularse un fallo, y a "yes" que sí. El bloque `tes:receive` se evaluará cuando el proceso responda al cliente. Con `tes:condition` pueden comprobarse si ciertas condiciones esperadas en el mensaje de respuesta se cumplen, si esta condición no se evalúa como verdadera para la actividad no pasará la prueba.

tes:receiveSend Espera un mensaje simple, lo verifica, y envía una respuesta de vuelta sincronizada.

tes:receiveSendAsynchronous Se reciben primero los datos, y se responde de forma asíncrona, por tanto, el proceso cliente no espera la respuesta.

tes:sendReceiveAsynchronous Envía un mensaje simple, espera una respuesta asíncrona, y verifica la respuesta.

Caso de prueba *OnlyAirlineReservationFailed*

El Listado 6.4 muestra el caso de prueba definido que matará a los mutantes no equivalentes que produce la composición WS-BPEL *TravelReservationService* al aplicarle el operador XEE.

Este caso de prueba, denominado *OnlyAirlineReservationFailed*, no está basado en otro caso de prueba, tampoco es abstracto y no se desea que se repita varias veces (línea 1).

Como se ha comentado anteriormente, `tes:clientTrack` define la petición que se simulará desde el cliente simulado (línea 2). El cliente mandará una petición síncrona, `<tes:sendReceive>`. Para ello, los atributos especifican a qué servicio (`trs:TravelReservationSoapService`) y puerto (`TravelReservation SoapHttpPort`) se enviará la petición, y qué operación se usará (`buildItinerary`) (líneas 3 – 6).

Los datos que se enviarán al proceso se encuentran incluidos dentro de `tes:data` en `tes:send`. Estos datos seguirán la estructura definida por el propio proceso (`OTA _TravelItinerary.xsd`). El atributo fault a "false" indicará que no debe simularse un fallo (líneas 8 – 106). En este caso de prueba podemos observar que los datos que se enviarán son los datos pertenecientes al itinerario del viaje que el cliente desea reservar. Es importante destacar que el itinerario contiene una referencia a un vuelo (líneas 11 – 13). Además, contiene información personal del cliente (líneas 14 – 35), información sobre el itinerario (líneas 36 – 92), el coste del viaje (líneas 93 – 100) y la información de actualización de la reserva (líneas 101 – 103).

`tes:send` se evaluará cuando el proceso responda al cliente (líneas 108 – 114). El atributo `fault` a "yes", indica que se espera recibir un fallo SOAP desde la composición WS-BPEL. `tes:condition` se utiliza para comprobar si ciertas condiciones esperadas en el mensaje de respuesta se cumplen, si esta condición no se evalúa como verdadera para la actividad no pasará la prueba. En este caso, se espera que el aserto XPath `//trs:ItineraryFault` contenga el valor *'Airline reservation has failed'*, informando al cliente que la reserva de vuelo ha fallado.

A continuación, se definen los *mockup* de los servicios externos. En las líneas 117 – 135 se define el *mockup* del servicio de reserva de vuelos. Mediante `tes:receiveSendAsynchronous` se reciben primero los datos, y se responde de forma asíncrona, por tanto, el proceso cliente no espera la respuesta. En este caso, el *mockup* del servicio externo recibe el mensaje indicando que se desea reservar el vuelo, a través de la operación `reserveAirline` (líneas 119 – 122). El servicio de reserva de vuelos notifica a través del `callback` de la composición que la reserva no ha podido realizarse (líneas 124 – 133); para identificar cuál es el vuelo que ha fallado se utilizará el identificador único de la reserva, especificado en `tes:data`.

Los *mockups* de los servicios externos de vehículo y hotel no serán invocados en este caso de prueba (líneas 137 – 143).

Listado 6.4: Caso de prueba *OnlyAirlineReservationFailed*

```
1   <tes:testCase name="OnlyAirlineReservationFailed" basedOn="" abstract="false"
        vary="false">
2     <tes:clientTrack>
3       <tes:sendReceive
4           service="trs:TravelReservationSoapService"
5           port="TravelReservationSoapHttpPort"
6           operation="buildItinerary">
7
```

```
8           <tes:send fault="false">
9             <tes:data>
10              <ota:TravelItinerary>
11                <ota:ItineraryRef Type="14" ID="M839LW">
12                  <ota:UniqueID>M839LWAA</ota:UniqueID>
13                </ota:ItineraryRef>
14                <ota:CustomerInfos>
15                  <ota:CustomerInfo RPH="01">
16                    <ota:Customer>
17                      <ota:PersonName>
18                        <ota:NamePrefix>Mr.</ota:NamePrefix>
19                        <ota:GivenName>Robert</ota:GivenName>
20                        <ota:MiddleName>Anthony</ota:MiddleName>
21                        <ota:Surname>Jones</ota:Surname>
22                      </ota:PersonName>
23                      <ota:Telephone PhoneNumber="2241630" AreaCityCode="302" Extension
                            ="5574"/>
24                      <ota:Email>rajones@somewhere.org</ota:Email>
25                      <ota:Address FormattedInd="true">
26                        <ota:StreetNmbr PO_Box="P.O. Box 77">1452 S. 13th St. N.W.</
                              ota:StreetNmbr>
27                        <ota:CityName>Westfinster</ota:CityName>
28                        <ota:PostalCode>90210</ota:PostalCode>
29                        <ota:StateProv>NY</ota:StateProv>
30                        <ota:CountryName>US</ota:CountryName>
31                      </ota:Address>
32                    </ota:Customer>
33                    <ota:AgencyAcctNumber Type="5" ID="UOTWT0122321"/>
34                  </ota:CustomerInfo>
35                </ota:CustomerInfos>
36                <ota:ItineraryInfo>
37                  <ota:ReservationItems ChronoOrdered="true">
38                    <ota:Item RPH="01" ItinSeqNumber="01">
39                      <ota:Air DepartureDateTime="2003-03-03T08:00:00" FlightNumber="
                            942" ResBookDesigCode="Y">
40                        <ota:DepartureAirport LocationCode="DEN"/>
41                        <ota:ArrivalAirport LocationCode="ORD"/>
42                        <ota:MarketingAirline>UA</ota:MarketingAirline>
43                        <ota:Seats>
44                          <ota:Seat Number="07A" Characteristic="Window" CustomerRPH="01
                                "/>
45                        </ota:Seats>
46                      </ota:Air>
47                    </ota:Item>
48                    <ota:Item RPH="02" ItinSeqNumber="02">
49                      <ota:Air DepartureDateTime="2003-03-03T13:00:00" FlightNumber="
                            3545" ResBookDesigCode="Y">
50                        <ota:DepartureAirport LocationCode="ORD"/>
51                        <ota:ArrivalAirport LocationCode="YUL"/>
52                        <ota:OperatingAirline>AC</ota:OperatingAirline>
53                        <ota:MarketingAirline>UA</ota:MarketingAirline>
54                        <ota:Seats>
```

```
55                    <ota:Seat Number="08B" Characteristic="Aisle" CustomerRPH="01"
                          />
56                  </ota:Seats>
57                </ota:Air>
58              </ota:Item>
59              <ota:Item RPH="03" ItinSeqNumber="05">
60                <ota:Air DepartureDateTime="2003-03-10T08:00:00" FlightNumber="
                      5568">
61                  <ota:DepartureAirport LocationCode="YUL"/>
62                  <ota:ArrivalAirport LocationCode="BOS"/>
63                  <ota:OperatingAirline>AC</ota:OperatingAirline>
64                  <ota:MarketingAirline>UA</ota:MarketingAirline>
65                  <ota:Seats>
66                    <ota:Seat Number="3D" Characteristic="Window" CustomerRPH="01"
                          />
67                  </ota:Seats>
68                </ota:Air>
69              </ota:Item>
70              <ota:Item RPH="04" ItinSeqNumber="06">
71                <ota:Air DepartureDateTime="2003-03-10T10:30:00" FlightNumber="
                      5542">
72                  <ota:DepartureAirport LocationCode="ORD"/>
73                  <ota:ArrivalAirport LocationCode="DEN"/>
74                  <ota:MarketingAirline>UA</ota:MarketingAirline>
75                  <ota:Seats>
76                    <ota:Seat CustomerRPH="01" Number="23B"/>
77                  </ota:Seats>
78                </ota:Air>
79              </ota:Item>
80            </ota:ReservationItems>
81            <ota:Ticketing TicketType="eTicket" eTicketNumber="30465876954325"
                  PlatingCarrier="UAL"/>
82            <ota:ItineraryPricing ItemRPH_List="01 02 03 04">
83              <ota:Cost AmountAfterTax="745.00" CurrencyCode="USD"/>
84            </ota:ItineraryPricing>
85            <ota:SpecialRequestDetails>
86              <ota:SpecialServiceRequests>
87                <ota:SpecialServiceRequest FlightRefNumberRPHList="01"
                      TravelerRefNumberRPHList="01" SSRCode="VGML">
88                  <ota:Airline Code="UA">United Airlines</ota:Airline>
89                </ota:SpecialServiceRequest>
90              </ota:SpecialServiceRequests>
91            </ota:SpecialRequestDetails>
92          </ota:ItineraryInfo>
93          <ota:TravelCost>
94            <ota:FormOfPayment>
95              <ota:PaymentCard CardNumber="3151002645983754" ExpireDate="1205"
                    CardType="MC">
96                <ota:CardHolderName>ROBERT A. JONES</ota:CardHolderName>
97              </ota:PaymentCard>
98            </ota:FormOfPayment>
99            <ota:CostTotals AmountAfterTax="1957.92" CurrencyCode="USD"/>
```

```
100                </ota:TravelCost>
101                <ota:UpdatedBy>
102                  <ota:Access ID="U09932147"/>
103                </ota:UpdatedBy>
104              </ota:TravelItinerary>
105            </tes:data>
106          </tes:send>
107
108          <tes:receive fault="true">
109            <tes:condition>
110              <tes:expression>//trs:itineraryFault</tes:expression>
111              <tes:value>'Airline reservation has failed.'</tes:value>
112            </tes:condition>
113          </tes:receive>
114        </tes:sendReceive>
115      </tes:clientTrack>
116
117      <tes:partnerTrack name="Airline">
118        <tes:receiveSendAsynchronous>
119          <tes:receive service="aes:AirlineReservationSoapService"
120                       port="AirlineReservationSoapHttpPort"
121                       operation="reserveAirline"
122                       fault="false"/>
123
124          <tes:send service="aes:AirlineReservationCallbackService"
125                    port="AirlineReservationCallbackSoapHttpPort"
126                    operation="airlineReservationFailed"
127                    fault="false">
128            <tes:data>
129              <ota:ItineraryRef Type="14" ID="M839LW">
130                <ota:UniqueID>M839LWAA</ota:UniqueID>
131              </ota:ItineraryRef>
132            </tes:data>
133          </tes:send>
134        </tes:receiveSendAsynchronous>
135      </tes:partnerTrack>
136
137      <tes:partnerTrack name="Vehicle">
138        <!-- Not invoked -->
139      </tes:partnerTrack>
140
141      <tes:partnerTrack name="Hotel">
142        <!-- Not invoked -->
143      </tes:partnerTrack>
144
145    </tes:testCase>
```

Caso de prueba *AirlineDataWithoutCustomerPersonName*

La implementación de este caso de prueba puede obtenerse del CD adjunto.

Este caso de prueba tiene un `tes:clientTrack` con `tes:sendReceive`. Los datos que se enviarán al proceso, incluidos en `tes:send`, son similares a los enviados anteriormente, excepto en que el cliente no proporcionará su nombre personal.

En `tes:send` el atributo `fault` tiene valor "yes", que indica que se espera recibir un fallo SOAP desde la composición WS-BPEL. Se espera que el aserto XPath `//trs:ItineraryFault` contenga el valor *'The following FCT activities have occurred:Termination - reserveAirline invoke activity is terminated.Fault - Airline Data without customer person name.'*, informando al cliente que han ocurrido las siguientes actividades de Fallo, Compensación y/o Terminación: de Terminación porque ha terminado la actividad de invocación de la reserva, y de Fallo ya que se ha lanzado un fallo al no encontrarse el nombre personal del cliente que realiza la reserva.

Sin embargo, no será invocado ni el `tes:partnerTrack` *Airline*, ni el *Vehicle* ni el *Hotel*.

Caso de prueba *AirlineDataWithoutCustomerEmail*

La implementación de este caso de prueba puede obtenerse del CD adjunto.

Este caso de prueba es análogo al anterior. También tiene un `tes:clientTrack` con `tes:sendReceive`. Los datos que se enviarán al proceso son similares a los enviados anteriormente, excepto en que el cliente, esta vez, sí proporcionará su nombre personal pero, sin embargo, no proporciona su correo electrónico, fundamental para llevar a cabo la reserva con éxito.

En `tes:send` el atributo `fault` también tiene valor "yes". Se espera que el aserto XPath `//trs:ItineraryFault` contenga el valor *'The following FCT activities have occurred:Termination - reserveAirline invoke activity is terminated.Fault - Airline Data without customer email.'*, informando al cliente que han ocurrido las siguientes actividades de Fallo, Compensación y/o Terminación: de Terminación porque ha terminado la actividad de invocación de la reserva, y de Fallo ya que se ha lanzado un fallo al no encontrarse el correo electrónico del cliente que realiza la reserva.

No será invocado ni el `tes:partnerTrack` *Airline*, ni el *Vehicle* ni el *Hotel*.

Caso de prueba *AirlineDataWithoutFormOfPayment*

La implementación de este caso de prueba puede obtenerse del CD adjunto.

Este caso de prueba es análogo al anterior. También tiene un `tes:clientTrack` con `tes:sendReceive`. Los datos que se enviarán al proceso son similares a los enviados anteriormente, excepto en que el cliente, esta vez, sí proporcionará su correo electrónico pero, sin embargo, no proporciona la forma de pago con la que efectuará la reserva.

En `tes:send` el atributo `fault` también tiene valor "yes". Se espera que el aserto XPath `//trs:ItineraryFault` contenga el valor *'The following FCT activities have occurred:Termination - reserveAirline invoke activity is terminated.Fault - Airline Data without form of payment.'*, informando al cliente que han ocurrido las siguientes actividades de Fallo, Compensación y/o Terminación: de Terminación porque ha terminado la actividad de invocación de la reserva, y de Fallo ya que se ha lanzado un fallo al no encontrarse la forma de pago con la que el cliente realizará la reserva.

No será invocado ni el `tes:partnerTrack` *Airline*, ni el *Vehicle* ni el *Hotel*.

Caso de prueba *OnlyAirline*

La implementación de este caso de prueba puede obtenerse del CD adjunto.

Este caso de prueba se encargará de realizar una reserva únicamente de avión, de ahí su nombre.

El `tes:clientTrack` también contiene una actividad `tes:sendReceive`. En el bloque `tes:send` se encuentran los datos que se enviarán al proceso. El bloque `tes:receive` se encarga de comprobar si el identificador único del itinerario del viaje, indicado por la expresión `ota:TravelItinerary/ota:ItineraryRef/ota:UniqueID`, es el valor `'M839LWAA'`.

En cuanto a los *partnerTrack* el único que se invocará será el de avión. Éste contiene una actividad `tes:receiveSendAsynchronous`. En este caso, en el bloque `tes:receive` se recibe un mensaje que solicita que se reserve un avión con la operación `reserveAirline` y se envía de forma asíncrona los datos de la reserva realizada con éxito, indicada con la operación `airlineReserved`.

Caso de prueba *OnlyAirlineCancelled*

La implementación de este caso de prueba puede obtenerse del CD adjunto.

Este caso de prueba se basa en el caso de prueba anterior, *OnlyAirline*, por tanto, el `tes:clientTrack` será el mismo.

Sin embargo, el `partnerTrack` del avión contiene dos actividades. La primera actividad `tes:receiveOnly` en la que se recibe un mensaje que desea realizar la reserva de avión, indicado con la operación `reserveAirline`.

La segunda actividad es `tes:receiveSend` que llevará a cabo la cancelación de esta reserva, la operación se denomina `cancelAirline`. Esta actividad contiene un bloque `tes:receive` vacío, puesto que no recibe ningún mensaje, y un bloque `tes:send` que enviará un mensaje comunicando que la reserva de avión se ha cancelado.

Caso de prueba *OnlyHotel*

Este caso de prueba es análogo al caso de prueba *OnlyAirline*, pero en lugar de realizar una reserva de avión se realizará una reserva de hotel.

Caso de prueba *OnlyHotelCancelled*

Este caso de prueba es análogo al caso de prueba *OnlyAirlineCancelled*, pero en lugar de llevar a cabo una cancelación de avión se hará de un hotel.

Caso de prueba *OnlyVehicle*

Este caso de prueba es análogo al caso de prueba *OnlyAirline*, pero en lugar de realizar una reserva de avión se realizará una reserva de vehículo.

Caso de prueba *OnlyVehicleCancelled*

Este caso de prueba es análogo al caso de prueba *OnlyAirlineCancelled*, pero en lugar de llevar a cabo una cancelación de avión se hará de un vehículo.

6.7.4. Pruebas manuales

Comprobar que la composición está bien formada y es válida

Una vez modificada la composición *TravelReservationService* es necesario comprobar si es "sintácticamente correcta". Para ello, hay que:

1. Comprobar si es un documento XML bien formado. Esto puede comprobarse abriendo el documento con el navegador *Firefox*, por ejemplo, o bien utilizando una herramienta XML de línea de comandos denominada *xmllint* utilizando el siguiente comando: `xmllint fichero.xml >/dev/null`. Cualquier problema con el código XML como tal debería aparecer por pantalla.
2. Comprobar si es un proceso WS-BPEL válido y que sigue funcionando sin problemas. Por tanto, se ejecuta con el BPTS original y se comprueba que se superan todas las pruebas. Para ello, se deben realizar los siguientes pasos:
   a) Arrancar ActiveBPEL con el siguiente comando: `ActiveBPEL.sh start`
   b) Ejecutar el proceso modificado con: `mutationtool run ruta-a.bpts ruta-a.bpel >salida.xml`.

c) Abrir el fichero *salida.xml* y comprobar que el conjunto de casos de prueba tiene valor `PASSED`, y no `FAILURE` o `ERROR`.

Comprobar que el mutante se despliega correctamente

Para comprobar que el mutante se despliega correctamente y que se ejecuta de la forma esperada debe consultarse los ficheros `~/AeBpelAdmin/deployment-logs` y `~/AeBpelAdmin/process-logs`. En caso de que no se despliegue, habrá que observar qué fallo SOAP es el que está enviando el mutante (debe obtenerse de los resultados de BPELUnit). En https://neptuno.uca.es/redmine/wiki/sources-fm/RecomendacionesDepuracion se enumeran algunas recomendaciones para la depuración.

Comprobar la calidad del conjunto de casos de prueba definido

Lo que determina la calidad de los casos del *.bpts* definidos, es si podemos distinguir con ellos entre el original y un mutante no equivalente. Si el mutante es realmente equivalente, es decir, es el mismo comportamiento pero escrito de otra forma, no hay nada que hacer.

El problema es que la única herramienta que sabe si un mutante es realmente equivalente o simplemente resistente es la "inspección manual" que, hasta la fecha, nadie ha podido automatizar aún. Éste es el problema del oráculo. A continuación presentamos un ejemplo de un mutante equivalente:

Programa *p*:

```
for (int i = 0; i < 5; i++)
{
   ... (el valor de i no cambia)
}
```

Mutante equivalente *m*:

```
for (int i = 0; i != 5; i++)
{
   ... (el valor de i no cambia)
}
```

Para comprobar la calidad de los casos de prueba, se estudiarán los resultados de la matriz de ejecución (§6.6.1). Se debe comprobar dos cosas: que el original supera la prueba, y que el mutante se distingue del original.

7 Resumen

En este PFC se han implementado algunos de los operadores de mutación definidos para el lenguaje WS-BPEL 2.0 que permiten modelar los fallos comunes que pueden cometer los programadores a la hora de escribir un programa en este lenguaje, en concreto: **EMF**, **ASI**, **XMF**, **XMT**, **XER** y **XEE**.

Se ha realizado una búsqueda bibliográfica de ejemplos de composiciones WS-BPEL. Puesto que ninguno de los encontrados se adaptaban a nuestras necesidades, se tomó la decisión de modificar la composición *TravelReservationService*.

Además, se han definido los casos de prueba, para comprobar que los operadores se han implementado correctamente, y el conjunto de casos de prueba para dicha composición.

Por otro lado, se ha estudiado y analizado, por primera vez, las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL y los definidos para otros lenguajes, que ha sido redactado en un artículo [4], y aceptado en el V Taller de PRIS 2010.

8 Conclusiones y trabajo futuro

En esta sección presentamos las conclusiones a las que se han llegado tras desarrollar este PFC y el trabajo futuro.

8.1. Valoración

Este PFC ha sido desarrollado para ayudar en las tareas de investigación del grupo SPI&FM. Éste ha sido el principal motivo por el que este PFC ha podido llevarse a cabo con éxito y en un tiempo razonable.

Se han cumplido todos los objetivos propuestos. Además, esta memoria de Proyecto Fin de Carrera es el resultado de un buen trabajo de documentación sobre el proyecto MutationAnalysis. Hasta ahora, no existía mucha documentación al respecto. Esta memoria permitirá que cualquier persona que desee trabajar con los operadores de mutación tenga una referencia sobre cómo hacerlo.

Destacar la parte más específica de investigación sobre los operadores de mutación realizada durante este período. Hemos presentado un artículo [4] que ha sido aceptado en el V Taller de PRIS 2010.

Algunos de los conocimientos más destacables que he adquirido durante la realización de este Proyecto han sido sobre los siguientes temas:

- Prueba y análisis de mutaciones, especialmente sobre operadores de mutación.
- Algoritmos genéticos.
- Servicios web.
- Lenguajes: WS-BPEL, WSDL, XSLT, XPath, XML Schema.
- Tecnologías: SOAP, JUnit, BPELUnit.
- Herramientas: Ant, Subversion, Eclipse, TkDiff, XMLEye.

Además de adquirir conocimientos, este PFC también me ha ayudado a madurar como Ingeniero Informático y he aprendido a que:

- El trabajo en equipo es fundamental al trabajar en temas de investigación. Normalmente la bibliografía disponible tanto en soporte papel como electrónico ha sido escasa, y ha sido necesaria la opinión y los conocimientos de los miembros de este grupo para optar por la mejor solución en cada caso.
- El correo electrónico es un buen medio para contactar con los directores, pero el contacto personal que hemos tenido durante los seminarios celebrados aproximadamente cada semana me ha ayudado no sólo a resolver mis dudas, sino también a tener una actitud positiva, de superación y de responsabilidad ante las dificultades presentadas.
- Es necesario utilizar un SCV (Sistema de Control de Versiones) para desarrollar proyectos de cierta complejidad.

8.2. Trabajo futuro

Como para cualquier proyecto de investigación, este PFC presenta unas líneas de trabajo futuro.

En primer lugar, continuar la investigación sobre operadores de mutación. Hasta el momento se han estudiado de forma exhaustiva cuáles son las equivalencias existentes entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes de programación, en concreto, para algunos lenguajes de programación no orientados a objetos y para otros orientados a objetos.

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

Además, otra posible ampliación de este estudio puede ser incluir los operadores de mutación definidos para los lenguajes de programación orientados a aspectos, así como otros operadores de mutación que no intervienen en la prueba de mutaciones de programa sino en la prueba de mutaciones de especificación [25], entre los que caben destacar los operadores relacionados con la prueba de mutaciones de servicios Web.

A partir de las conclusiones que se obtengan durante este período de investigación, podrán definirse nuevos operadores de mutación para WS-BPEL 2.0, por ejemplo, ope-

radores que tengan en cuenta criterios de cobertura, que habrá que implementar. Esta memoria servirá de ayuda para esta labor.

9 Agradecimientos

- A Inmaculada Medina Bulo, por darme la oportunidad de colaborar en este proyecto y por animarme en todo momento.
- A Antonio García Domínguez, por todo el conocimiento que me ha aportado y por su ayuda tan eficaz.
- A todos los miembros del grupo de investigación SPI&FM, por su apoyo.
- A mi familia y amigos, por su comprensión, en estos meses, de mi gran dedicación al trabajo y al estudio, y por su apoyo.

10 Manual de instalación

En esta sección se presenta un breve manual de instalación de GAmera. Se recomienda disponer de una distribución Ubuntu 9.10 o openSUSE 11.0.

10.1. Instrucciones de instalación de GAmera

Para instalar GAmera realice los siguientes pasos:

1. Descargar en cualquier directorio el guión de instalación de: https://neptuno.uca.es/redmine/projects/sources-fm/repository/changes/trunk/scripts/install.sh.
2. Ejecutar desde ese directorio el siguiente comando: `bash install.sh gamera`
3. Seguir las instrucciones para completar la instalación.

Este guión instalará las dependencias comunes a Takuan y GAmera: Java 6, Tomcat 5.5, ActiveBPEL 4.1, XMLBeans 2.1.0, BPELUnit 1.4 y Saxon-B 9.1.0.1. Además, instalará lo específico a GAmera: mutationtool 1.0.7, galib 2.4.7, JUnit 4 y GAmera en sí.

11 Manual del usuario

En esta sección se presenta un breve manual de usuario de la herramienta **mutationtool**.

Es aconsejable que el usuario tenga algunos conocimientos sobre la prueba de mutaciones de programa para que pueda comprender la utilidad de esta herramienta.

Para poder utilizar **mutationtool** debe previamente haber instalado GAmera. Siga las instrucciones que se indican en §10.1.

mutationtool es un guión que forma parte de MutationAnalysis, y que hace de envoltorio para lanzar mutationtool.jar, el JAR que reúne el código compilado de MutationAnalysis y sus dependencias (InstrumentadorProcesoBPEL, AnalizadorXPath, ConversorXPath y Saxon-B 9.1.0.1, el motor XSLT 2.0).

Para ejecutar este programa escriba en la terminal el siguiente comando:

```
mutationtool (argumentos)
```

11.1. Analizar una composición WS-BPEL

Consiste en identificar qué instrucciones o elementos del programa original pueden mutarse. La sintaxis es:

```
mutationtool analyze bpel
```

El resultado de este análisis será un listado que contiene para cada operador de mutación el número de las instrucciones en las que el operador puede ser aplicado y el máximo valor del atributo.

> **NOTA:** Se puede indicar que enumere cada operador. De esta forma podrá también indicarse el operador por número, en lugar de por nombre.

```
mutationtool analyze bpel | nl
```

Analizar la composición *TravelReservationService*

Para analizar la composición *TravelReservationService*:

```
mutationtool analyze TravelReservationService.bpel
```

La salida que se obtiene es:

Listado 11.1: Salida del análisis de la composición *TravelReservationService*

```
1  ISV 0 1
2  EAA 0 4
3  EEU 0 1
4  ERR 6 5
5  ELL 0 1
6  ECC 24 1
7  ECN 6 4
8  EMD 4 2
9  EMF 0 2
10 ACI 0 1
11 AFP 0 1
12 ASF 13 1
13 AIS 0 1
14 AIE 0 1
15 AWR 0 1
16 AJC 0 1
17 ASI 33 1
18 APM 0 1
19 APA 3 1
20 XMF 5 1
21 XMC 0 1
22 XMT 1 1
23 XTF 6 1
24 XER 4 1
25 XEE 1 1
```

Si nos fijamos, por ejemplo, en el operador **XMF** que elimina un elemento `catch` o el elemento `catchAll` de un manejador de fallos, podemos observar que el número de operandos es 5 y el número máximo de atributo es 1 (valor por omisión).

Esto quiere decir, que en esta composición WS-BPEL existen 5 elementos, `catch` o `catchAll`. Por tanto, podremos obtener 5 mutantes distintos al aplicar este operador a cada uno de estos operandos.

En este caso, el valor del atributo máximo no es de utilidad. Si tomamos, por ejemplo, el operador **ERR** que sustituye un operador relacional por otro del mismo tipo, tiene 6 operandos para esta composición y un valor máximo de atributo de 5 (véase la Tabla 6.1), es decir, cada uno de los 6 operadores relacionales que aparecen en esta composición puede sustituirse por uno de los 5 valores válidos del atributo: <, >, >=, <=, =, !=.

11.2. Aplicar un operador a la composición WS-BPEL original

Consiste en aplicar un operador a la composición WS-BPEL original para generar un mutante. La sintaxis es:

```
mutationtool apply bpel operator operand attribute
```

Para ello, deberá indicarse el operador, el operando y, en algunos casos, también el valor del atributo.

> **NOTA:** El operador podrá indicarse tanto por nombre como por el número que tenga asociado (coincide con el orden de aparición en la lista resultante del análisis).

El resultado será la composición WS-BPEL mutada que se mostrará por la salida estándar. Podrá redirigirse a un fichero para conservar este mutante.

Aplicar el operador XMF a la composición *TravelReservationService*

El resultado del análisis para este operador ha sido:

```
XMF 5 1
```

Tenemos la posibilidad de obtener 5 mutantes distintos. En este ejemplo, decidimos mutar el primer elemento `catch` o `catchAll` que aparezca en la composición. En este caso es un elemento `catchAll` que se encuentra dentro del manejador de fallos del proceso. Por tanto, por la definición de este operador, además de eliminar este elemento también eliminará por completo dicho manejador.

Para aplicar este operador:

```
mutationtool apply TravelReservationService.bpel XMF 1 1
   > mutante1-xmf.bpel
```

Como se ha comentado anteriormente, el valor del atributo es indiferente para este operador, por eso se especifica el valor 1 por omisión.

Veamos el mutante que se ha generado. Para comprobar si la mutación se ha realizado correctamente puede comprobarse manualmente o utilizar alguna herramienta, como TkDiff, que indique automáticamente cuáles son las diferencias existentes entre la composición *TravelReservationService* original y el mutante. Para ello, utilizamos la línea de órdenes:

```
tkdiff TravelReservationService.bpel mutante1-xmf.bpel
```

Se recomienda seleccionar en este programa que se ignore el espaciado en blanco, en caso contrario se mostrarán demasiadas diferencias. Como puede observarse en la Figura 11.1 se ha obtenido el mutante que se esperaba. No obstante, siempre es recomendable realizar la normalización de la composición original antes de compararla[1] (véase §11.8).

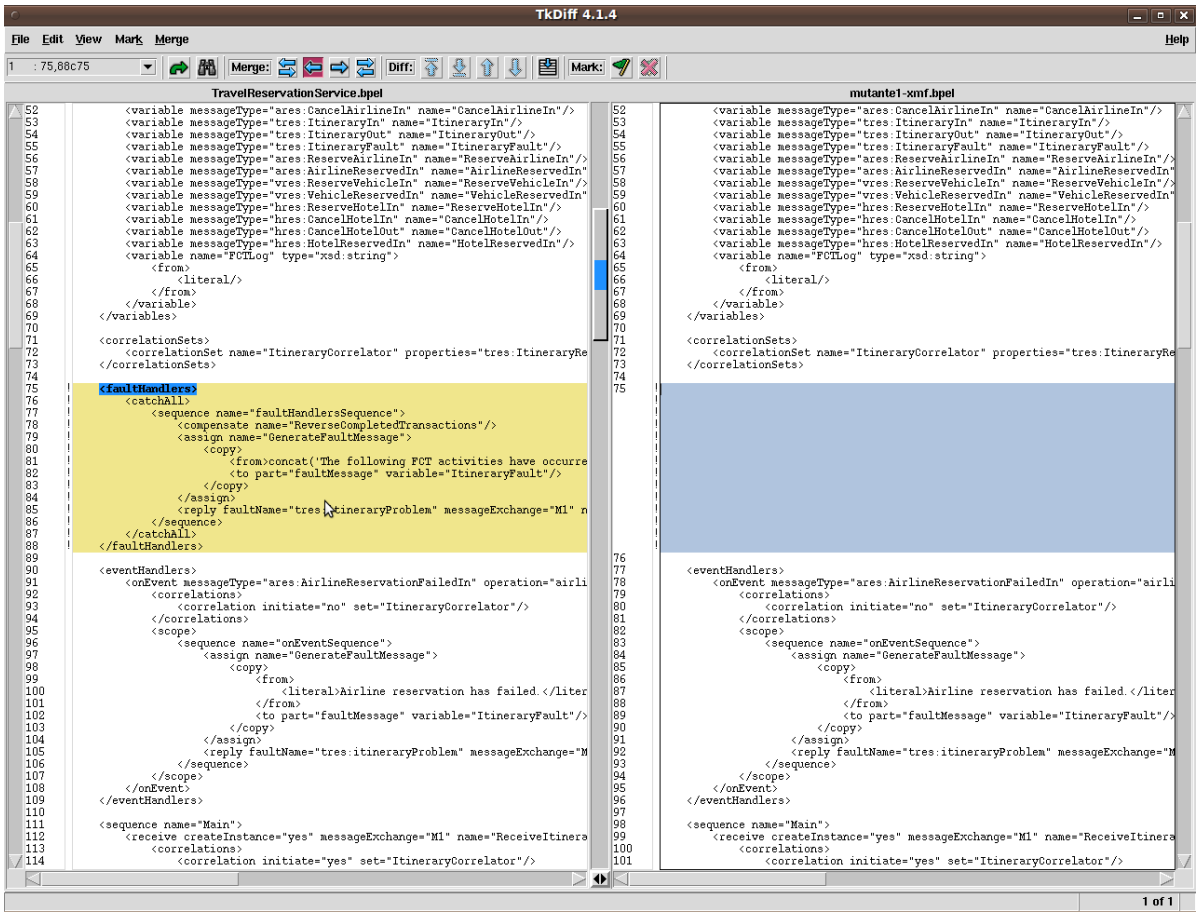

Figura 11.1: Diferencias entre la composición *TravelReservationService* y un mutante

11.3. Aplicar todos los operadores a la composición WS-BPEL original

Esta opción es una generalización de la anterior. En lugar de especificar qué operador, operando y valor de atributo aplicar, se aplican, de forma automática, todos los ope-

[1] No es necesario normalizar un mutante, ya que `apply` lo imprime normalizado.

radores a la composición WS-BPEL original para generar todos los mutantes posibles. La sintaxis es:

```
mutationtool applyall bpel
```

Se obtendrá un fichero distinto con cada mutante generado. El formato del nombre de cada fichero es: *mxx-yy-zz.bpel*, donde *m* indica que se trata de un mutante, *xx* indica el operador que se le ha aplicado (valor comprendido entre 0 y el número total de operadores menos 1), *yy* indica el operando y *zz* el valor del atributo con el que se ha generado ese mutante.

Aplicar todos los operadores a *TravelReservationService*

Para ello:

```
mutationtool applyall TravelReservationService.bpel
```

Por la salida estándar se mostrarán qué mutante se está generando en cada momento:

Listado 11.2: Salida de la aplicación de todos los operadores a *TravelReservationService*

```
1 Generating (ERR=4,1,1)
2 Generating (ERR=4,1,2)
3 Generating (ERR=4,1,3)
4 Generating (ERR=4,1,4)
5 Generating (ERR=4,1,5)
6 Generating (ERR=4,2,1)
7 Generating (ERR=4,2,2)
8  ...
9  ...
```

Una vez terminado este proceso, se obtendrán los ficheros con los mutantes, nombrados como se ha comentado anteriormente.

11.4. Ejecutar la composición WS-BPEL sobre el conjunto de casos de prueba

Consiste en ejecutar la composición WS-BPEL sobre el conjunto de casos de prueba, para obtener si ha pasado con éxito o no cada uno de estos casos. La sintaxis es:

```
mutationtool run bpts bpel
```

El fichero *fichero.bpts* contendrá el conjunto de casos de prueba sobre el que se ejecutará la composición WS-BPEL.

> **NOTA:** `mutation run` ejecuta todos los casos de prueba, puesto que tiene que producir los resultados XML completos. Sólo se debe utiilizar esta opción con la composición original. Para los mutantes utilizar la opción `compare` (véase §11.5).

Antes de llevar acabo esta ejecución, es necesario levantar el motor ActiveBPEL, que se ejecuta dentro del servidor de aplicaciones Apache Tomcat. Para ello:

```
ActiveBPEL.sh start
```

Si se han iniciado correctamente se obtendrá la siguiente salida:

```
Starting Tomcat (remote debugging OFF)... done.
Waiting until ActiveBPEL is ready... done.
```

La sintaxis de ActiveBPEL.sh es:

```
ActiveBPEL.sh start [-debug]|pause|stop|restart|ls|kill|
              killall|status
```

start Inicia Tomcat y ActiveBPEL.

pause Realiza una pausa.

stop Para ActiveBPEL.

restart Para y reinicia ActiveBPEL.

ls Lista PID de los procesos en ejecución.

kill PID Mata al proceso con el PID indicado.

killall Mata a todos los procesos.

status Indica el estado de ActiveBPEL.

Ejecutar la composición *TravelReservationService* sobre el conjunto de casos de prueba

Para ello:

```
mutationtool run travelTest.bpts TravelReservationService.bpel
   > salida-original.xml
```

El documento XML resultante puede abrirse en cualquier visor. Se puede utilizar, por ejemplo, Firefox:

Listado 11.3: Salida de la ejecución de *TravelReservationService*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tes:testResult name="Test Suite TravelReservationService" result="PASSED"
3 message="Passed" xmlns:tes="http://www.bpelunit.org/schema/testResult">
4     <tes:state name="Status Code">PASSED</tes:state>
5     <tes:state name="Status Message">Passed</tes:state>
6     <tes:testCase name="Test Case OnlyAirlineReservationFailed" result="PASSED"
7 message="Passed">
8  ...
9  ...
```

En el Listado 11.3 puede observarse que la composición *TravelReservationService* original ha superado el conjunto de los casos de prueba (línea 2).

Al igual que en la línea 6, en este documento se indicará si para cada caso, en concreto, se ha superado o no la prueba. De esta forma, si el conjunto de casos de prueba no pasa la prueba, podrá localizarse cuáles son los casos de pruebas que fallan.

11.5. Ejecutar los mutantes y comparar hasta que se encuentre la primera diferencia

Consiste en ejecutar los mutantes y comparar hasta que se encuentre la primera diferencia entre la salida de la composición original y las salidas de las ejecuciones de las mutaciones. La sintaxis es:

```
mutationtool compare bpts bpel xml bpel1...
```

Donde *fichero.bpts* es el conjunto de casos de prueba, *fichero.bpel* es la composición original, *fichero.xml* es la salida de ejecución de la composición original y *fichero.bpelN* es cada uno de los mutantes que se desea comparar (separados por espacios).

Para cada mutante, la salida será una fila con tantos números como casos de pruebas existan en el *fichero.bpts* (véase §6.6.1). Si el número es 0 indica que es la misma salida para el caso de prueba correspondiente, si es 1 indica que es distinta y si es 2 indica un *stillborn*.

Como este proceso se parará en el primer caso de prueba que muestre diferencias o cuando compruebe que no se ha desplegado correctamente el servicio, sólo mostrará el primer 1 de la fila: el resto estará todo a 0. Si el mutante no es válido, se pondrá toda la fila a 2.

Ejecutar un mutante y compararlo hasta encontrar la primera diferencia con *TravelReservationService*

Para ello:

```
mutationtool compare travelTest.bpts
   TravelReservationService.bpel
   salida-original.xml mutante1-xmf.bpel
```

La salida obtenida es:

```
0 1 0 0 0 0 0 0 0 0
```

Es decir, el mutante *mutante1-xmf.bpel* ha sido matado por el caso de prueba que aparece en segundo lugar en el fichero *travelTest.bpts*, denominado `AirlineDataWithoutCustomerPersonName`. Al encontrar la primera diferencia, se termina el proceso de comparación.

Esto es correcto. En el mutante *mutante1-xmf.bpel* se había eliminado por completo el manejador de fallos del proceso principal. Por tanto, teniendo en cuenta la definición del proceso *TravelReservationService*, el cliente no recibirá el mensaje de fallo adecuado, que indica el problema ocurrido durante la reserva de vuelo.

11.6. Ejecutar todos los mutantes y comparar las salidas

Ejecutar todos los mutantes y comparar la salida de la composición original con las salidas de las ejecuciones de las mutaciones, es decir, comprobar cómo se comporta cada caso de prueba con los mutantes. La sintaxis es:

```
mutationtool comparefull bpts bpel xml bpel1...
```

El funcionamiento de la opción *comparefull* es igual a la de *compare* a excepción de que *comparefull* no detiene su ejecución cuando encuentre la primera diferencia entre la salida de ejecución de la composición original y la de un mutante.

Ejecutar un mutante y comparar la salida con la de *TravelReservationService*

Para ello:

```
mutationtool comparefull travelTest.bpts
   TravelReservationService.bpel
   salida-original.xml mutante1-xmf.bpel
```

La salida obtenida es:

```
0 1 1 1 0 0 0 0 0 0
```

Como puede observarse, esta vez el mismo mutante *mutante1-xmf.bpel* ha sido matado no sólo por el caso de prueba que aparece en segundo lugar en el fichero *travelTest.bpts*, denominado `AirlineDataWithoutCustomerPersonName`, sino también por el caso de prueba `AirlineDataWithoutCustomerEmail` y `AirlineDataWithout FormOfPayment`. Al utilizar la opción *comparefull* en lugar de *compare* no se termina el proceso de comparación al encontrar la primera diferencia.

Se puede comprobar de igual modo, que la salida obtenida es la esperada.

> **NOTA:** Como es evidente, la opción *comparefull* necesitará más tiempo de cómputo que *compare*.

11.7. Comparar dos salidas de ejecución de una composición WS-BPEL

Consiste en comparar dos ficheros que contengan dos salidas de ejecución de una composición WS-BPEL, ya sea la composición original o un mutante de dicha composición, sobre el conjunto de casos de prueba. La sintaxis es:

```
mutationtool compareout xml1 xml2
```

En este caso, no se realiza ninguna ejecución. Simplemente se comparan las dos salidas proporcionadas: *fichero.xml1* y *fichero.xml2*.

> **NOTA:** El número de casos de prueba de los dos ficheros debe ser el mismo.

11.8. Normalizar una composición WS-BPEL

Consiste en normalizar una composición WS-BPEL, es decir, obtener una forma canónica de ésta. La sintaxis es:

```
mutationtool normalize bpel
```

> **NOTA:** Se recomienda redirigir la salida estándar a un fichero para poder utilizar esta composición normalizada en las otras opciones de **mutationtool**.

12 Manual del desarrollador

En esta sección se presenta un breve manual para el desarrollador.

Para ello, deberá descargar el proyecto MutationAnalysis con el siguiente comando:

```
svn co https://neptuno.uca.es/svn/sources-fm/trunk/
              src/MutationAnalysis
```

12.1. Implementar nuevos operadores de mutación para WS-BPEL 2.0

A continuación, se presentan unas intrucciones breves sobre los pasos a realizar en el caso de que se definan nuevos operadores de mutación para WS-BPEL 2.0 y sea necesario implementarlos:

1. Implementar el código XSLT de cada operador, que realice la mutación adecuada del código original escrito en WS-BPEL. Si el operador afecta a una expresión XPath tendrá que heredar de `xpath-op-base.xsl`, si elimina actividades o elementos de la definición de proceso WS-BPEL tendrá que heredar de `delete-op-base.xsl`, si no heredará de `op-base.xsl` (véase §6.6.3). Esta hoja XSLT estará alojada en el directorio `MutationAnalysis/src/es/uca/webservices/mutants/operators`.
2. Definir los casos de prueba, utilizando el framework JUnit, para comprobar que los operadores se han implementado correctamente (véase §6.7.2). El fichero se localizará en `MutationAnalysis/test/es/uca/webservices/mutants`. Los casos de prueba se definirán atendiendo a:
   a) Número de operadores detectados en la composición.
   b) Modificaciones introducidas al ejecutar los operadores.
   c) Comprobaciones de que las salidas de la composición original y la de los mutantes son diferentes.

3. Adaptar un ejemplo de composición WS-BPEL 2.0 o crear uno nuevo para que le sea aplicable, al menos, los operadores implementados. En el caso de que se cree una nueva composición se creará un nuevo directorio con el nombre de la composición correspondiente en `MutationAnalysis/test_processes`. Este directorio contendrá, al menos, los ficheros con la composición WS-BPEL (*composicion.bpel*), el conjunto de casos de prueba (*conjunto-casos-prueba.bpts*), definiciones de los servicios web (*nombre-servicio.wsdl*) y el fichero para describir la estructura y las restricciones de los contenidos de los documentos XML (*tipos.xml*).
4. Crear el conjunto de casos de prueba BPTS y comprobar que contiene los casos de prueba suficientes para matar a todos los mutantes no equivalentes que produce la composición WS-BPEL adaptada.

Para que los cambios surtan efecto, hay que regenerar el .jar (que contiene el código Java y los *.xsl*) y colocarlo en el directorio en el que esté el **mutationtool** que se desea ejecutar. Para ello, se debe ejecutar el siguiente comando desde el directorio `trunk/src/MutationAnalysis`:

```
ant dist
```

Para probar el programa habrá que ejecutarlo desde el **mutationtool** que está en ese directorio, y su .jar:

```
./mutationtool (argumentos)
```

12.2. Configurar la ejecución de los casos de prueba de los operadores de mutación

Para comprobar si un operador que se ha implementado es correcto y pasa todos los casos de prueba unitarias definidas mediante el framework JUnit existen dos opciones:

1. Utilizar Eclipse para hacer el desarrollo, ejecutando sólo las pruebas directamente relacionadas. En este caso, hay que:
   a) Importar los proyectos Eclipse de AnalizadorXPath, ConversorXPath, InstrumentadorBPEL y MutationAnalysis.
   b) Definir la biblioteca de usuario BPELUnit con los jar que hay en `/bin/bpelunit/lib`.

   c) Cuando se tengan todos los proyectos compilando sin errores se puede pulsar el botón derecho en una de las clases de pruebas (las que acaban en Test) y ejecutar sólo sus pruebas de unidad seleccionando `Run As >JUnit Test`.

2. Ir al directorio principal `src/MutationAnalysis` y ejecutar `ant test`. Esto ejecutará todos los casos de prueba unitarias, e informará cuáles fallan y cuáles van bien.

Se recomienda utilizar la primera opción, Eclipse, durante el desarrollo puesto que permite incluso ejecutar una prueba determinada o depurar su ejecución, haciéndolo más rápido y cómodo. El inconveniente es que requiere bastante configuración.

La segunda opción se aconseja cuando se ha terminado el desarrollo y se desea comprobar que efectivamente todas las pruebas se ejecutan sin fallos.

A WS-BPEL 2.0

WS-BPEL es un lenguaje formal para expresar un proceso de negocio como una composición de servicios Web. Según los objetivos establecidos al diseñar este lenguaje, intenta unificar "los conceptos de modelado de proceso. . . codificados en una plataforma neutral y portable".

WS-BPEL 2.0 fue elegido estándar oficial por OASIS en Abril de 2007.

Puesto que WS-BPEL es un lenguaje formal se supone que la semántica es inequívoca. Por tanto, la codificación es esencial para lograr dicha portabilidad. Por esta razón, WS-BPEL se basa en el lenguaje de marcado extensible (XML), que es diseñado especialmente para ofrecer la portabilidad a través de una codificación de plataforma neutral.

La definición de WS-BPEL se compone de dos partes: un conjunto de documentos XML Schema, la definición de la sintaxis del lenguaje, y un documento, denominada *especificación estándar*, la definición de la semántica y algunas restricciones de la sintaxis.

XML Schema es un lenguaje formal, por lo que es adecuado para una definición formal. La especificación estándar, sin embargo, está escrito en inglés plano. Existe un peligro al definir un lenguaje formal usando un lenguaje natural, dando lugar a definiciones y requisitos ambiguos o inconsistentes.

En [22] se ha realizado un estudio a fondo de la especificación estándar. Se ha concluido que esta especificación contiene requisitos innecesarios, ambiguos o contradictorios, y se proponen algunas inconsistencias con las tecnologías relacionadas. Se demuestra que el uso de actividades implícitas y las extensiones en WS-BPEL dan lugar a relaciones *oscuras* entre la definición de proceso y su semántica. Por ello, se propone una colección de plantillas para transformar cualquier descripción de proceso WS-BPEL en uno simplificado pero equivalente a nivel semántico, definido usando sólo un subconjunto del lenguaje, denominado *Simplified BPEL* o *S-BPEL*. Además, se presenta una colección de plantillas para transformar el WS-BPEL XML Schema en el esquema propuesto S-BPEL.

Un documento WS-BPEL se relaciona con al menos un documento WSDL, la definición de las operaciones de servicio Web proporcionadas y utilizadas por el proceso. Además, se deben utilizar las definiciones XML Schema para definir los tipos o elementos utilizados por las variables y se deben importar las definiciones de referencia.

La especificación estándar utiliza las siguientes especificaciones: WSDL, XML Schema, XPath y XSLT, y recomienda WS-I (Web Services Interoperability Organization) Basic Profile 1.1, por tanto, recomienda indirectamente SOAP.

A.1. Un ejemplo sencillo

Un documento WS-BPEL describe un proceso, utilizando únicamente los servicios Web para comunicarse con el mundo exterior. Para ejecutar una instancia del proceso, la descripción debe ser desplegada en un servidor. Un proceso es instanciado a la llegada de un mensaje, tal como se define en la descripción del proceso.

Para introducir la estructura y los conceptos básicos de una descripción de proceso se presenta una ejemplo sencillo, pero a su vez completo, descrito en [22]. En este ejemplo se recibirá un mensaje que contiene una cantidad y una moneda, ya sea dólares o euros, devolviendo un mensaje con la cantidad en DKK[1].

El Listado A.1 muestra el documento WS-BPEL completo. El elemento raíz es el elemento `process`, la definición de las importaciones (líneas 5 – 7), los agentes externos o *partner links* (líneas 9 – 13), las variables (líneas 15 – 18) y la actividad principal (líneas 20 – 56) del proceso.

Listado A.1: Un proceso de negocio sencillo

```
1  <process xmlns="http://docs.oasisopen.org/wsbpel/2.0/process/executable"
2     name="exchange" targetNamespace="http://beepell.com/tests/exchange/bpel"
3     xmlns:exc="http://beepell.com/tests/exchange/wsdl">
4
5     <import importType="http://schemas.xmlsoap.org/wsdl/"
6        namespace="http://beepell.com/tests/exchange/wsdl"
7        location="simple.wsdl" />
8
9     <partnerLinks>
10       <partnerLink name="exchangePartnerLink"
11          partnerLinkType="exc:exchangePartnerLinkType"
12          myRole="exchangeProviderRole" />
13    </partnerL inks >
14
15    <variables>
16       <variable name="request" messageType="exc:exchangeMessage" />
17       <variable name="response" messageType="exc:exchangeMessage" />
18    </variables>
19
20    <sequence>
21       <receive
22          partnerLink="exchangePartnerLink"
23          operation="exchangeOperation"
```

[1] ISO 4217 Currency Abbreviations: United States Dollar (USD), Euro (EUR), Danish Krone (DKK).

```
24          variable="request" createInstance="yes"/>
25
26        <if>
27          <condition>$request.currency = 'USD'</condition>
28          <assign>
29            <copy>
30              <from>$request.amount * 4.7850</from>
31              <to variable="response" part="amount" />
32            </copy>
33          </assign>
34          <elseif>
35            <condition>$request.currency = 'EUR'</condition>
36            <assign>
37              <copy>
38                <from>$request.amount * 7.4627</from>
39                <to variable="response" part="amount" />
40              </copy>
41            </assign>
42          </elseif>
43        </if>
44
45        <assign>
46          <copy>
47            <from><literal>DKK</literal></from>
48            <to variable="response" part="currency" />
49          </copy>
50        </assign>
51
52        <reply
53          partnerLink="exchangePartnerLink"
54          operation="exchangeOperation"
55          variable="response"/>
56    </sequence>
57  </process>
```

Regresamos a las importaciones y a los agentes externos. En este proceso se definen dos variables: `request` (de solicitud) y `response` (de respuesta), ambas de tipo de mensaje `exchangeMessage`. Este tipo es un mensaje WSDL definido en el documento WSDL importado, que se muestra en el Listado A.2. Utilizamos estas variables para almacenar el mensaje recibido y preparamos el mensaje para enviar como respuesta.

Un proceso se compone de **actividades**. La especificación estándar distingue entre actividades **básicas** y **estructuradas**. Las actividades estructuradas contienen otras actividades y su semántica define el flujo de control del proceso. Las actividades básicas son las operaciones básicas, por ejemplo recibir un mensaje, asignar un valor a una variable o invocar un servicio Web. Las actividades básicas no pueden contener otras actividades.

La actividad principal de este proceso es un actividad de sequencia, definida por el elemento `<sequence>` (líneas 20 – 56). La actividad de sequencia es una actividad

estructurada, que define una lista de actividades que se ejecutan en el orden léxico en el que aparecen[2]. En este proceso la secuencia contiene las actividades `<receive>` (líneas 21 – 24), `<if>` (líneas 26 – 43), `<assign>` (líneas 45 – 50) y `<reply>`.

La actividad `<reply>` (líneas 52 – 55) está a la espera de recibir un mensaje, que especifica una variable para almacenar el mensaje (línea 24). Al atributo `createInstance` se le asigna el valor "yes", causando una nueva instancia del proceso que se creará a la llegada del mensaje esperado.

Una vez recibido el mensaje, la actividad `<if>` se ejecuta. Tiene una condición principal (línea 27) para ejecutar su actividad principal (líneas 28 – 33). La condición es una expresión XPath booleana:

$$request.currency =' USD'$$

El lado izquierdo de la expresión es una referencia de variable a la parte `currency` de la variable de mensaje `request`. Un mensaje WSDL está construido por partes, cada una con su propio tipo y valor.

Si la condición es cierta, la actividad `<assign>` (líneas 28 – 33) se ejecuta. De lo contrario, se considera la cláusula `elseif` (líneas 34 – 42).

Una actividad `<assign>` contiene una o más operaciones `<copy>`. Cada copia contiene un elemento `from` y un elemento `to`. En ambas asignaciones de la actividad `<if>`, se copia una expresión, el cálculo de la cantidad convertida, a la parte `amount` de la variable `response`.

Tras la actividad `<if>` se ejecuta la actividad `<assign>` que copia el valor literal "DKK" a la parte `currency` de la variable `response` (líneas 45 – 50).

Por último, una actividad `<reply>` envía la variable `response` en respuesta al mensaje recibido (líneas 52 – 55).

Listado A.2: Documento WSDL importado (simple.xsd)

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <wsdl:definitions
4     targetNamespace="http://beepell.com/tests/exchange/wsdl"
5     xmlns:exc="http://beepell.com/tests/exchange/wsdl"
6     xmlns:plnk="http://docs.oasisopen.org/wsbpel/2.0/plnktype"
7     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
8     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
9     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
10
11    <plnk:partnerLinkType name="exchangePartnerLinkType">
12       <plnk:role name="exchangeProviderRole" portType="exc:exchangePortType"/>
13    </plnk:partnerLinkType>
```

[2]La especificación estándar usa la palabra "léxico" refiriéndose al orden de aparición, no al orden alfabético.

```
14
15    <wsdl:message name="exchangeMessage">
16       <wsdl:part name="currency" type="xsd:token" />
17       <wsdl:part name="amount" type="xsd:decimal" />
18    </wsdl:message >
19
20    <wsdl:portType name="exchangePortType">
21       <wsdl:operation name="exchangeOperation">
22          <wsdl:input message="exc:exchangeMessage" />
23          <wsdl:output message="exc:exchangeMessage" />
24       </wsdl:operation>
25    </wsdl:portType >
26
27    <wsdl:binding name="exchangeBinding" type="exc:exchangePortType">
28       <soap:binding style="rpc"
29          transport="http://schemas.xmlsoap.org/soap/http"/>
30
31       <wsdl:operation name="exchangeOperation">
32          <soap:operation soapAction="http://beepell.com/tests/exchange/wsdl"/>
33          <wsdl:input>
34             <soap:body use="literal"
35                namespace="http://beepell.com/tests/exchange/wsdl" />
36          </wsdl:input>
37          <wsdl:output>
38             <soap:body use="literal"
39                namespace="http://beepell.com/tests/exchange/wsdl" />
40          </wsdl:output>
41       </wsdl:operation>
42    </wsdl:binding >
43
44    <wsdl:service name="exchangeService">
45       <wsdl:port name="exchangePort" binding="exc:exchangeBinding">
46          <soap:address
47          location="http://localhost:9090/samples/proxy/definitions/exchange"/>
48       </wsdl:port>
49    </wsdl:service>
50
51 </wsdl:definitions>
```

La Figura A.1 muestra el árbol de ejecución de cómo fue ejecutada la instancia del proceso. Podemos comprobar que la actividad `<if>` sólo tiene un nodo hijo `<assign>`, porque sólo se ha ejecutado una de las asignaciones.

A.2. Estructura básica

Como ya se ha comentado, WS-BPEL es un lenguaje basado en XML. Su espacio de nombres es:

```
1 http://docs.oasis-open.org/wsbpel/2.0/process/executable
```

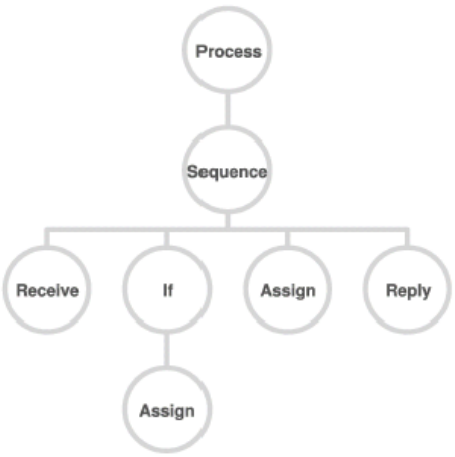

Figura A.1: Árbol de ejecución del proceso de negocio

El elemento raíz es `<process>`, y debe definirse, como mínimo, el atributo `name`: el nombre del proceso. Adicionalmente, pueden especificarse los atributos:

queryLanguage Especifica el lenguaje de consulta que utiliza el proceso para la selección de nodos en las asignaciones. El valor por defecto es `"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"`, es decir, XPath 1.0.

expressionLanguage Determina el lenguaje de las expresiones usadas en el elemento `<process>`. El valor por defecto es `"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"`, que, de nuevo, representa XPath 1.0.

suppressJoinFailure Especifica si el error `joinFailure` será suprimido para todas las actividades. El valor por defecto es "no". Puede sobrecargarse por cada actividad.

exitOnStandardFault Si el valor de este atributo es "yes", entonces el proceso deberá terminar inmediatamente (como si se alcanzara una actividad `<exit>`), cuando ocurra un fallo estándar distinto de `bpel:joinFailure`. Si el valor es "no", entonces el proceso podrá manejar un fallo estádar usando un manejador de fallos. El valor por defecto es "no".

En el caso de que se estén empleando extensiones, podría añadirse un elemento `<extensions>` al proceso dentro del cual se especifiquen tantas `<extension>` como sea necesario. Estos elementos deben tener dos atributos:

namespace El espacio de nombres de la extensión.

mustUnderstand Especifica si el motor de ejecución debe entender la extensión, o puede ignorarla.

Además de las extensiones, un proceso puede contener los siguientes elementos en su primer nivel:

<import> Importación de otros ficheros: declaraciones WSDL, XSD...

<partnerLinks> Declaración de los enlaces a otros servicios (agentes externos).

<variables> Declaración de variables globales.

<correlationSets> Vincula un mensaje con una instancia concreta del proceso.

<faultHandlers> Manejadores de fallos.

<eventHandlers> Manejadores de eventos.

Actividad La actividad principal del proceso.

La mayoría de estos elementos se detallan a continuación. Para obtener una información más completa sobre su sintaxis, consulte la especificación estándar [43].

A.3. Actividades

El ejemplo de la Sección A.1 muestra cómo un proceso de negocio se compone de actividades. La especificación estándar especifica las veinte actividades, ocho estructuradas y doce básicas.

Las **actividades básicas** son:

<assign> Se usa para asignar valores a las variables.

<compensate> Se usa sólo dentro de un manejador de fallos, de compensación o de terminación para comenzar la compensación.

<compensateScope> Se usa sólo dentro de un manejador de fallos, de compensación o de terminación para comenzar la compensación de un actividad `<scope>` específica.

<empty> No hace nada, pero puede utilizarse para sincronizar.

<exit> Provoca la salida inmediata del proceso.

<invoke> Invoca un servicio Web.

<receive> Espera la llegada de un mensaje en una operación específica.

<reply> Devuelve un mensaje en respuesta a un mensaje recibido previamente.

<rethrow> Se usa sólo dentro de un manejador de fallos para relanzar el fallo.

<throw> Lanza un fallo.

<validate> Valida una o más variables.

<wait> Provoca la parada del proceso durante un periodo de tiempo específico o hasta que se alcanza una fecha límite.

Las **actividades estructuradas** son:

<flow> Proporciona concurrencia mediante la ejecución de todas sus actividades hijas en paralelo. También introduce enlaces de sincronización para hacer que una actividad espere por otra, o salte su ejecución por completo.

<forEach> Es un constructor repetible especial para ejecutar múltiples instancias de la misma actividad `<scope>`.

<if> Proporciona un comportamiento condicional. Se toma la primera rama cuya `<condition>` se evalúa a *true*, y se ejecuta la actividad que contiene. Si no se toma una rama con una condición, entonces se toma la rama `<else>` si existe.

<pick> Permite elegir uno de varios eventos (mensajes entrantes o alarmas temporizadas) y continuar por la ruta de ejecución seleccionada.

<repeatUntil> Proporciona la ejecución repetida de una actividad contenida. La actividad contenida se ejecuta hasta que la `<condition>` booleana se evalúe a *true*.

<scope> Permite la declaración de los recursos locales, como las variables y los agentes externos, y una actividad principal a ejecutar en este entorno local. También incluye manejadores de compensación, de fallos, de terminación y de eventos.

<sequence> Contiene una o más actividades que son ejecutadas secuencialmente, en el orden léxico en el que aparecen dentro de esta actividad.

<while> Proporciona la ejecución repetida de una actividad contenida. La actividad contenida se ejecuta mientras que la `<condition>` booleana se evalúa a *true* al comienzo de cada iteración.

A.3.1. Empty

La actividad `<empty>` se utiliza para no hacer nada. También se utiliza para establecer un punto de sincronización en una actividad `<flow>`.

- **Condición de finalización**: la actividad `<empty>` se completa inmeditamente tras ejecutar el estado entero.
- **Condiciones de fallo**: no deben ser lanzadas en el estado de ejecución.
- **Comportamiento de terminación**: se debe permitir que la actividad `<empty>` se complete, cuando se indique la terminación.

A.3.2. Validate

La actividad `<validate>` puede utilizarse para asegurarse que los valores de las variables son válidas a partir de su definición de datos XML asociada, incluyendo el tipo

simple de XML Schema, tipo complejo, definición de elemento y definiciones XML de partes WSDL.

- **Condición de finalización**: la actividad `<validate>` se completa cuando todas las variables son validadas.
- **Condiciones de fallo**:
  - `bpel:invalidVariables` Si una (o más) de las variables listadas en los atributos de variables es inválida.
- **Comportamiento de terminación**: la especificación estándar no describe un comportamiento de terminación para la actividad `<validate>`.

A.3.3. Wait

La actividad `<wait>` especifica un retardo por un periodo de tiempo determinado o hasta que una fecha límite determinada es alcanzada. Si el valor de duración especificado en `<for>` es cero o negativo, o una fecha límite especificada en `<until>` ya ha sido alcanzada o pasada, entonces la actividad `<wait>` se completa inmeditamente.

- **Condición de finalización**: la actividad `<wait>` se completa cuando el hilo de ejecución ha esperado la duración especificada o cuando la fecha se alcanza.
- **Condiciones de fallo**:
  - `bpel:invalidExpressionValue` Si la expresión `for` o `until` no devuelve una duración válida o una fecha respectivamente.
  - `bpel:subLanguageExecutionFault` Si la evaluación de la expresión falla.
- **Comportamiento de terminación**: la actividad `<wait>` debe ser interrumpida de forma prematura.

A.3.4. Exit

La actividad `<exit>` se utiliza para finalizar inmediatamente la instancia del proceso de negocio. Todas las actividades ejecutadas actualmente deben ser finalizadas inmediatamente sin involucrar ningún manejador de terminación, de fallo o comportamiento de compensación.

- **Condición de finalización**: el propósito de esta actividad es finalizar todas las actividades en ejecución, incluida ésta. Por definición nunca será completada.
- **Condiciones de fallo**: esta actividad no puede lanzar fallos.
- **Comportamiento de terminación**: esta actividad no puede ser terminada.

A.3.5. Throw

La actividad `<throw>` se utiliza cuando un proceso de negocio necesita lanzar un fallo interno explícitamente.

- **Condición de finalización**: no se completa, siempre causa un fallo.
- **Condiciones de fallo**: Siempre causa un fallo.
- **Comportamiento de terminación**: esta actividad nunca puede ser terminada.

A.3.6. Rethrow

La actividad `<rethrow>` se utiliza en los manejadores de fallo para relanzar el fallo que han capturado. Sólo puede ser utilizado dentro de un manejador de fallos.

- **Condición de finalización**: no se completa, siempre causa un fallo.
- **Condiciones de fallo**: Siempre causa un fallo.
- **Comportamiento de terminación**: esta actividad nunca puede ser terminada.

A.3.7. Invoke

La actividad `<invoke>` se utiliza para llamar a Servicios Web ofrecidos por proveedores de servicios. El uso típico es invocar una operación de un servicio, que se considera una actividad básica. Las operaciones pueden ser operaciones *request-response* o *one-way*, correspondiendo a las deficiones de operación WSDL.

La `inputVariable` se refiere a una variable de mensaje que se envía a la `operation` del `partnerLink`. Si es una operación *request-response*, el mensaje de respuesta se almacena en la `outputVariable`.

La `inputVariable` o `outputVariable` podría omitirse si el mensaje WSDL correspondiente no contiene ninguna parte.

La semántica de la correlación podría provocar que un mensaje no se envíe.

- **Condición de finalización**: la actividad `<invoke>` es una actividad bloqueante. Para una operación *request-response* la actividad no se completa hasta que un mensaje de respuesta sea recibido. Para una operación *one-way*, se completa cuando el mensaje se envía. Éste no es claramente un estado en la especificación estándar, y para operaciones *one-way* que no tienen un mensaje de respuesta, el bloqueo no es un requisito necesario.
- **Condiciones de fallo**:

- `bpel:correlationViolation` Si las restricciones de correlación son violadas.
- `bpel:uninitializedPartnerRole` Si una actividad `<invoke>` se utiliza en un `partnerLink` cuyo `partnerRole EPR` no es inicializado.
- *`fault messages`* Si una invocación *request-response* devuelve un mensaje de fallo WSDL, esto da lugar a un fallo identificado en WS-BPEL por el *namespace* objetivo del correspondiente tipo de puerto y el nombre de fallo.

- **Comportamiento de terminación**: cada actividad `<invoke>` debe interrumpirse y terminarse de forma prematura. Cuando un `<invoke>` *request-response* se interrumpe y termina de forma prematura, la respuesta (si es recibida), por ejemplo una actividad terminada, debe ignorarse.

A.3.8. Receive

Una actividad `<receive>` especifica el `partnerLink` que contiene el `myRole` utilizado para recibir mensajes y la `operation` que espera el *partner* para invocar.

La actividad `<receive>` es una de las tres actividades de mensajes de entrada, IMA (Inbound Message Activity). Las otras dos son las actividades estructuradas `<pick>` y `<onEvent>`. La última no es realmente una actividad, sino parte del manejador de eventos.

Los atributos `partnerLink`, `operation`, `variable` y `messageExchange` son comunes a todos los IMAs. El `myRole` y `partnerLinkType` del `partnerLink` declarado en un `<scope>`, identifica el tipo de puerto WSDL que declara la operación.

El atributo `variable` nombra la variable usada para almacenar el mensaje recibido. Éste debe ser una variable de tipo mensaje del mismo tipo que el correspondiente mensaje de entrada WSDL. Éste debe ser omitido si el mensaje no contiene ninguna parte.

El atributo `createInstance` indica si un mensaje de entrada debe crear una nueva instancia del proceso. El valor por defecto es `no`. Existen muchas restricciones sobre el uso de `createInstance="yes"`.

Si la operación es una operación *request-response*, entonces la actividad `<reply>` debe ser utilizada para enviar el mensaje de respuesta. El atributo `messageExchange` se utiliza para eliminar la ambigüedad de la relación entre un IMA y un `<reply>`, donde múltiples IMAs y actividades `<reply>` usan el mismo `partnerLink` y `operation`. Los nombres usados en los atributos `messageExchange` se declaran en la actividad `<scope>`.

- **Condición de finalización**: `<receive>` es una actividad bloqueante que no se completará hasta que la instancia del proceso reciba un mensaje relacionado.

- **Condiciones de fallo**.
- **Comportamiento de terminación**: la actividad `<receive>` debe ser interrumpida y terminada de forma prematura.

A.3.9. Reply

La actividad `<reply>` se utiliza para enviar una respuesta a una petición previamente aceptada a través de una actividad de mensaje de entrada, por ejemplo la actividad `<receive>`.

Como todas las actividades de mensajes, `<reply>` es altamente dependiente de la implementación. La especificación estándar no describe cómo se envía una actividad `<reply>`, en qué punto debe considerarse que el mensaje de respuesta se ha entregado o cómo se maneja la terminación de forma prematura. Algunas de éstos dependen del protocolo de mensaje y algunos de la implementación.

- **Condición de finalización**: la especificación estándar no define claramente esta condición para esta actividad.
- **Condiciones de fallo**:
  - `bpel:correlationViolation` Si las restricciones de correlación son violadas.
  - `bpel:uninitializedVariable` Si una variable usada para la correlación o la variable de mensaje para enviar no es inicializada.
- **Comportamiento de terminación**: la actividad `<reply>` debe ser interrumpida y terminada de forma prematura.

A.3.10. Sequence

La actividad `<sequence>` contiene una o más actividades que son ejecutadas secuencialmente, en el orden léxico en el que aparecen dentro de la `<sequence>`.

- **Condición de finalización**: la actividad `<sequence>` se completa cuando la última actividad en la sequencia se ha completado.
- **Condiciones de fallo**: la actividad `<sequence>` no puede fallar.
- **Comportamiento de terminación**: los constructores `<sequence>` y `<flow>` deben terminarse terminando su comportamiento y aplicando la terminación a todas las actividades anidadas y activas actualmente dentro de ellos.

A.3.11. If

La actividad <if> proporciona un comportamiento condicional. Se toma la primera rama cuya <condition> se evalúa a *true*, y se ejecuta la actividad que contiene. Si no se toma una rama con una condición, entonces se toma la rama <else> si existe.

- **Condición de finalización**: la actividad <if> se completa cuando la actividad contenida en la rama seleccionada se completa, o inmediatamente cuando la <condition> no se evalúa a *true* y no se especifica la rama <else>.
- **Condiciones de fallo**.
- **Comportamiento de terminación**: si una actividad <if> o <pick> ha seleccionado una rama, entonces la terminación debe ser aplicada a la actividad de la rama seleccionada. Si ninguna de estas actividades han seleccionado una rama, entonces la actividad <if> o <pick> debe ser terminada inmediatamente.

A.3.12. While

La actividad <while> proporciona la ejecución repetida de una actividad contenida. La actividad contenida se ejecuta mientras que la <condition> booleana se evalúa a *true* al comienzo de cada iteración.

- **Condición de finalización**: la actividad <while> se completa cuando la condición se evalúa a *false*.
- **Condiciones de fallo**: fallos de expresiones estándar.
- **Comportamiento de terminación**: la iteración de <while>, <repeatUntil> y <forEach> en serie debe interrumpirse y la terminación debe aplicarse al bucle del cuerpo de la actividad.

A.3.13. RepeatUntil

La actividad <repeatUntil> proporciona la ejecución repetida de una actividad contenida. La actividad contenida se ejecuta hasta que la <condition> booleana se evalúe a *true*. La condición se comprueba después de cada iteración del cuerpo del bucle. Al contrario de la actividad <while>, el bucle <repeatUntil> ejecuta el contenido de la actividad al menos una vez.

- **Condición de finalización**: la actividad <repeatUntil> se completa cuando la condición se evalúa a *true*.
- **Condiciones de fallo**.

- **Comportamiento de terminación**: la iteración de `<while>`, `<repeatUntil>` y `forEach` en serie debe interrumpirse y la terminación debe aplicarse al bucle del cuerpo de la actividad.

A.3.14. Pick

La actividad `<pick>` espera por la ocurrencia de exactamente un evento de un conjunto de eventos, entonces ejecuta la actividad asociada con ese evento. Después de que un evento ha sido seleccionado, este `<pick>` no aceptará los otros eventos.

- **Condición de finalización**: la actividad `<pick>` se completa cuando la actividad contenida por el evento seleccionado se completa.
- **Condiciones de fallo**:
  - `bpel:correlationViolation` Si las restricciones de correlación son violadas.
  - `bpel:InvalidExpressionValue` Si una propiedad de variable que usa la correlación no devuelve un valor consistente con su tipo. O si fallan las expresiones de fecha límite o duración de los eventos `alarm`.
  - `bpel:Selection` Si una propiedad de variable que usa la correlación o las expresiones de fecha límite o duración de los eventos `alarm` no devuelven un solo nodo.
- **Comportamiento de terminación**: la actividad `<pick>` debe terminarse inmediatamente.

A.3.15. Flow

Una actividad `<flow>` crea un conjunto de actividades concurrentes directamente anidadas dentro de ella. Permite dependencias de sincronización entre actividades que son anidadas dentro de ella a cualquier profundidad. El constructor `<link>` se utiliza para expresar estas dependencias de sincronización. Las declaraciones de `<link>` son contenidas en una actividad `<flow>`.

- **Condición de finalización**: una actividad `<flow>` se completa cuando todas las actividades contenidas en `<flow>` se han completado. Si su condición se evalúa a *false* entonces se salta una actividad y también se considera completada.
- **Condiciones de fallo**: la actividad `<flow>` no puede fallar.
- **Comportamiento de terminación**: los constructores `<sequence>` y `<flow>` deben terminarse terminando su comportamiento y aplicando la terminación a todas las actividades anidadas y activas actualmente dentro de ellos.

A.3.16. ForEach

La actividad <forEach> es un constructor repetible especial para ejecutar múltiples instancias de la misma actividad <scope>. Se ejecuta en uno de los dos modos: paralelo o en serie. En modo paralelo es como una actividad <flow> y en modo en serie como una actividad <while>.

El atributo counterName nombra un contador especial que será implícitamente declarado en todas las instancias de la actividad <scope>, pero con diferentes valores desde startCounterValue a finalCounterValue.

La actividad <forEach> ejecutará su actividad <scope> contenida N+1 veces, donde N es igual a <finalCounterValue> menos <startCounterValue>.

El atributo parallel indica si las intancias N+1 de la actividad <scope> se ejecutarán concurrentemente, cuando se establece a "yes", o como una secuencia en serie, cuando se establece a "no".

La completionCondition se usa para parar la ejecución antes de que las N+1 se hayan completado. La expresión en los elementos branches define el número de <scope> que deben ser completados. De esta forma, es posible comenzar, por ejemplo, diez instancias del <scope>, definiendo que se han terminado cuando cinco de ellas se han completado. Las instancias que quedan serán terminadas. El atributo successfulBranchesOnly indica si los <scope> fallidos cuentan como completados (*no*) o sólo se cuentan los <scope> con éxito (*yes*).

- **Condición de finalización**: si la completionCondition es realizada o cuando todas las instancias N+1 se han completado.
- **Condiciones de fallo**: fallos de expresiones de evaluación.
- **Comportamiento de terminación**: la iteración de <while>, <repeatUntil> y <forEach> en serie debe interrumpirse y la terminación debe aplicarse al bucle del cuerpo de la actividad. Para un <forEach> paralelo, la terminación debe aplicarse a todas las ramas que se ejecutan en paralelo.

A.3.17. Assign

La actividad <assign> se utiliza para copiar datos de una variable a otra, y también para construir e insertar nuevos datos usando expresiones. El uso de estas expresiones viene motivado por la necesidad de realizar cálculos simples. Pueden operar sobre variables, propiedades, y literales.

Otra posibilidad de las asignaciones es copiar referencias desde y hacia agentes externos. Incluso es posible incluir operaciones definidas como elementos de extensión pertenecientes a espacios de nombres diferentes al de WS-BPEL. Si dicha extensión no es

reconocida por el procesador WS-BPEL, y no está sujeta a `mustUnderstand="yes"`, se ignorará.

A.4. Agentes externos y tipos

La Figura A.2, propuesta en [22], muestra cómo la descripción de un proceso no referencia directamente a tipos de puerto en las definiciones WSDL.

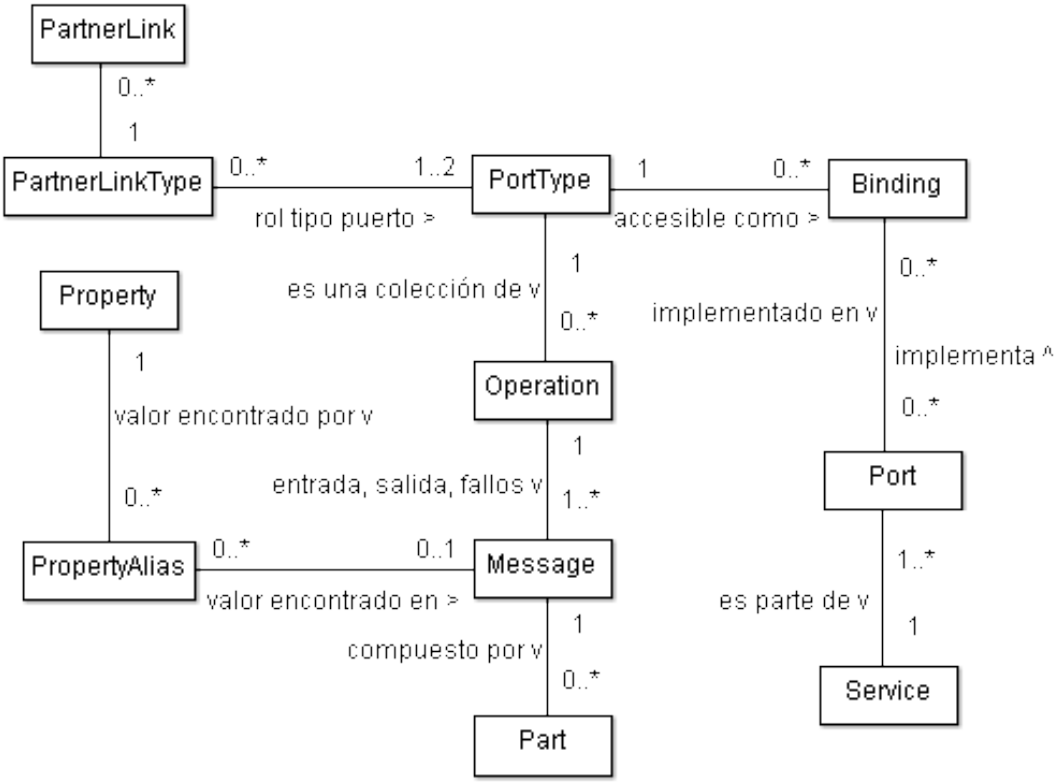

Figura A.2: Relación entre WS-BPEL y WSDL

A.4.1. Tipos de agentes externos

WS-BPEL utiliza el concepto de agente externo y roles para expresar la relación entre dos socios “de negocio”. La relación se expresa en términos de qué operaciones un socio debe exponer para que el otro las invoque.

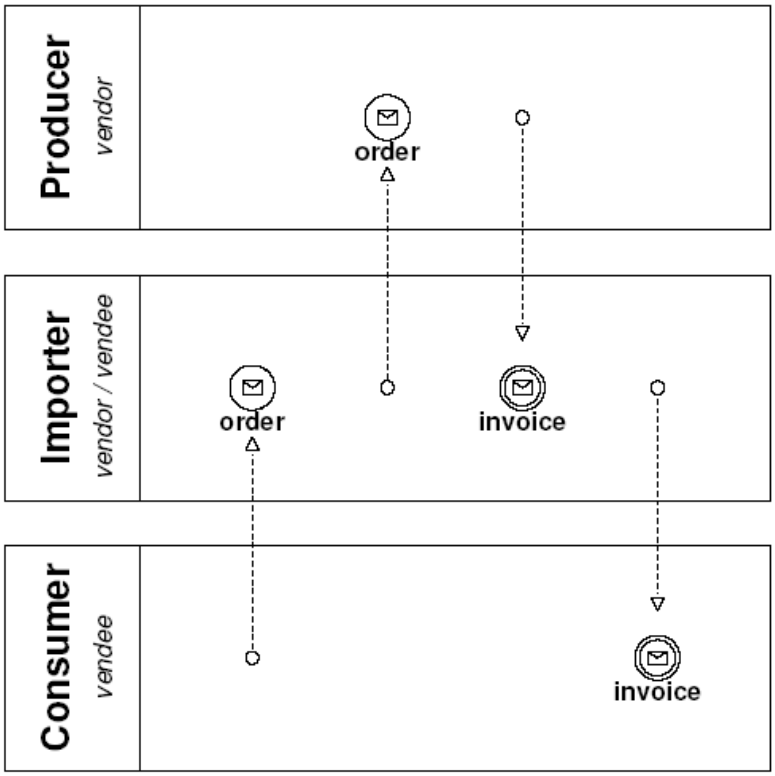

Figura A.3: Roles vendor y vendee

La Figura A.3, propuesta en [22], muestra la relación inter-proceso entre tres participantes. El *Consumer* (consumidor) juega el papel *vendee* (el que compra algo) y el *Producer* (productor) juega el papel de *vendor* (proveedor). El *Importer* (el que importa) juega ambos roles, comprar al productor y vender al consumidor.

El Listado A.3 muestra un fragmento WSDL con un agente externo denominado “trade” (comercio), que define la relación utilizando los roles “vendor” (línea 10) y “vendee” (línea 11). Un rol es definido por una referencia a un tipo de puerto WSDL.

Listado A.3: Fragmento WSDL

```
1  <wsdl:portType name="A">
2     <wsdl:operation name="order" ... />
3  </wsdl:portType>
4
5  <wsdl:portType name="B">
6     <wsdl:operation name="invoice" ... />
7  </wsdl:portType>
8
9  <plnk:partnerLink Type name="trade">
10    <plnk:role name="vendor" portType="tns:A" />
11    <plnk:role name="vendee" portType="tns:B" />
12 </plnk:partnerLink Type>
```

Un rol sólo describe las operaciones que un socio debe exponer al otro socio. No se describe el orden de invocación ni requiere que todas las operaciones expuestas sean utilizadas. Además, el rol “vendee” no incluye el envío de una petición, como el rol “vendor” no incluye el envío de la invocación.

El tipo de agente externo liga los dos roles juntos, que describe la interacción entre dos socios. Es posible que un tipo de agente externo tenga sólo un rol, cuando sólo uno de los socios debe exponer las operaciones.

A.4.2. Agentes externos

Un tipo de agente externo es abstracto en el sentido de que define los roles, pero no quién los juega. Ésta es la razón por la que los tipos de agentes externos se denominan “tipos”. Los “actores” con los roles que juegan se definen dentro de la descripción del proceso mediante agentes externos concretos.

Una descripción del proceso debe declarar al menos un agente externo para cada tipo de agente externo y rol asociado. Por tanto, el *Consumer* y el *Producer* sólo juegan un rol en un tipo de agente externo y necesita declarar sólo un agente externo. El *Importer* se ocupa de ambos roles y necesita declarar dos agentes externos.

Utilizamos el proceso *Importer* de la Figura A.3 para explicar el uso de los agentes externos. El Listado A.4 muestra una descripción del proceso, la implementación del proceso *Importer*. Los agentes externos se declaran en la sección `partnerLinks` del elemento `process` o en un `scope` local.

Listado A.4: Uso de agentes externos para especificar quién juega qué rol

```
1  <process name="importer">
2     ...
3     <partnerLinks >
4        <partnerLink name="consumer"
5           partnerLinkType="abc:trade"
6           myRole="vendor"
7           partnerRole="vendee"
8            initializePartnerRole ="yes" />
9
10       <partnerLink name="producer"
11          partnerLinkType="abc:trade"
12          myRole="vendee"
13          partnerRole="vendor"
14           initializePartnerRole ="yes" />
15    </partnerLinks>
16    ...
17    <sequence>
18       <receive partnerLink="consumer"
19          operation="order"
20          variable="purchase-order"
21          createInstance="yes" />
```

```
22        ...
23        <invoke partnerLink="producer"
24           operation="order"
25           inputVariable="back-order">
26           <correlations>
27              <correlation set="back-order-payment"
28                 initiate ="yes" pattern="request" />
29           </correlations>
30        </invoke>
31        ...
32        <receive partnerLink="producer"
33           operation="invoice"
34           variable="producer-invoice">
35           <correlations>
36              <correlation set="back-order-payment" initiate="no" />
37           </correlations>
38        </receive>
39        ...
40        <invoke partnerLink="consumer"
41           operation="invoice"
42           inputVariable="consumer-invoice" />
43     </sequence>
44     ...
45  </process>
```

El primer `partnerLink` (líneas 4 – 8) es el "enlace" entre este proceso (*Importer*) y el *Consumer*. Él es de tipo de agente externo "trade" (línea 5), define los dos roles *vendor* y *vendee*. A continuación declaramos quién juega qué rol: `myRole` es "vendor" (línea 6), el rol desempeñado por este proceso. El `partnerRole` es "vendee" (línea 7), el papel que debe desempeñar nuestro socio, el `Consumer`.

El segundo `partnerLink` es para el *Producer*. Es del mismo tipo, pero podemos observar cómo el rol se intercambia.

Los `partnerLink` se referencian desde las actividades de mensaje, en este caso `<receive>` e `<invoke>`. El primer `<receive>` (líneas 18 – 21) está recibiendo una petición de la operación "order" del agente externo "consumer". Cuando se recibe, está implícito que es el rol definido por `myRole` en el `partnerLink` referenciado, que utilizamos para buscar el tipo de puerto WSDL correspondiente.

Lo mismo ocurre en el `<invoke>` (líneas 23 – 30): invocamos a una operación de "order" en el `partnerLink` "producer". A diferencia de `<receive>` es el rol del "partner" el que se utiliza para buscar el tipo de puerto WSDL.

Hemos mostrado cómo declarar agentes externos e invocar operaciones sobre ellos. Sin embargo, para enviar y recibir mensajes concretos necesitamos conocer el protocolo de paso de mensajes que se utiliza, y la dirección "endpoint" actual.

A.5. Inicialización de *Endpoint*

De las definiciones de servicios Web sabemos qué partes debe contener un mensaje cuando lo enviamos para invocar una operación. El “partner” debe tener una dirección donde podemos enviar mensajes cuando se invocan operaciones, y el rol de “my” debe tener una dirección en la que recibimos mensajes. Llamamos a esta dirección “endpoint” o “endpoint address”.

El *endpoint* del rol “partner” es una variable, se le puede asignar un valor mediante la actividad `<assign>`. El *endpoint* del rol “my” es constante, sólo puede ser leído por una operación de asignación y su valor se determina en el despliegue.

Si el atributo `initializePartnerLink` de un agente externo se establece a “yes”, entonces el *endpoint* del rol *partner link* debe inicializarse en el despliegue, y si se establece a “no” debe ser inicializado por asignación.

El atributo `initializePartnerLink` es opcional y cuando se omite su implementación depende de si el *endpoint* se inicializa en el despliegue. Esto afecta a la portabilidad de las descripciones de procesos.

El valor de *endpoint* es un elemento `sref:service-ref`, un elemento contenedor con contenido dependiente de la implementación. WS-BPEL no está obligado a utilizar SOAP para el intercambio de mensajes, puede utilizarse cualquier protocolo. Por lo tanto, no es posible para la especificación estándar imponer restricciones sobre el contenido de dicho elemento `service-ref`.

Parece razonable suponer que el *endpoint* en el modelo de WS-BPEL se enlaza al agente externo y no al tipo de agente externo. Por lo tanto, cambiar el *endpoint* en uno de los agentes externos no tiene repercusiones sobre otros agentes externos con el mismo tipo de agente externo y rol. Sin embargo, esto no se especifica directamente, y como los *endpoints* en el modelo de WSDL son enlazados a tipos de puerto, cambiar un *endpoint* en un agente externo podría verse como cambiar el *endpoint* para el tipo de puerto.

En la Figura A.3, el *Importer* debe declarar dos agentes externos diferentes, pero del mismo tipo: “trade”. El agente externo “consumer” dispone de un tipo de puerto *A* como rol “my” y el *parter link* “producer” dispone de un tipo de puerto *A* como rol “partner”.

Cambiar el *endpoint* del rol “partner” en el agente externo “producer” no puede afectar al *endpoint* de rol “my” en el agente externo “consumer”, ya que es un valor constante. Por lo tanto, se especifica indirectamente que el *endpoint* se enlaza al agente externo, aunque puede ser inicializado desde un código común, como un documento WSDL.

A.6. Variables

En la especificación se distinguen tres clases de declaraciones de variables: elemento XML Schema, tipo (simple o complejo) de XML Schema y mensaje WSDL. Sin embargo, las diferencias entre las variables de tipo simple y complejo son tan significativas, que pueden considerarse como dos clases diferentes: se inicializan y se acceden de distinta forma cuando se usan en expresiones.

Las variables se declaran en una actividad `<scope>`. El elemento `process` tiene ámbito global. El acceso a las variables tiene un ámbito léxico.

Hay tres propiedades que son comunes para las cuatro clases de variables:

- El nombre de la variable WS-BPEL (`name`): el nombre utilizado para acceder a la variable. Es una cadena de texto que se ajuste al tipo XML Schema `NCName`, con la excepción de que los períodos no se les permite en cualquier parte del nombre. El nombre debe ser único dentro del ámbito al que pertenece.
- La definición del contenido de la variable (`messageType`, `type`, `element`): un nombre que se refiere a la definición del contenido de la variable. En función de la clase de la variable ésta debe ser una definición simple, compleja, de elemento o mensaje.
- Un booleano "inicializado" en tiempo de ejecución: en su ciclo de vida, una variable comienza sin estar iniciliazada. Si se intenta leer una variable que no está inicializada dará lugar a un "fallo de variable sin inicializar". En algún momento, la variable se inicializa al asignarle contenido.

A.7. Scopes

La actividad `<scope>` proporciona un contexto de comportamiento para sus actividades contenidas en ella, ofreciendo variables locales, agentes externos, mensajes de intercambio y conjunto de correlaciones, mediante manejadores de eventos, de fallos, de compensación y terminación.

La declaración de los manejadores de eventos dentro de una actividad `<scope>` permite la ejecución paralela basada en eventos, o un evento temporizado o un mensaje de entrada. El manejador de fallos puede capturar fallos desde las actividades contenidas y actividades ejecutadas. El manejador de compensación se activa cuando se le pide al `<scope>` completado la compensación de sus acciones. El manejador de terminación se activa para permitir una limpieza cuando se fuerza su terminación.

A.7.1. Inicialización

> "La inicialización de un <scope> consiste en instanciar e inicializar las variables y los agentes externos del <scope>, instanciar el conjunto de correlaciones, e instanciar los manejadores de fallos, el manejador de terminación y los manejadores de eventos. Una vez que la inicialización del <scope> se completa, la primera actividad del <scope> se ejecuta y los manejadores de eventos son instalados en paralelo con cada una." [43, sección 12.1]

Esto es una contradicción en la descripción de la inicialización del <scope>: instalar los manejadores de eventos es, en primer lugar, mencionado como parte de la inicialización del <scope> y, más tarde, se dice que debe ser hecho después de que la inicialización del <scope> se ha completado. Para clarificar esto necesitamos comprender por qué el estándar distingue si los manejadores de eventos son instalados antes o en paralelo con la ejecución de la primera actividad.

> "Cualquier agente externo definido en el <scope> debe ser establecido antes de la definición de las variables en el mismo <scope> cuya lógica de inicialización se refiere a estos agentes externos." [43, sección 12.1]

Como la inicialización de los agentes externos, durante la inicialización del <scope>, no depende de variables este requisito puede asegurarse inicializando los agentes externos antes de las variables. Por tanto, no hay requisitos para el orden de la inicialización.

Si se produce un fallo durante esta inicialización se lanza el fallo scopeInitializationFailure al <scope> padre.

A.7.2. Aplicando manejadores

La especificación estándar usa el término "instalar" sobre los manejadores, sin establecer una definición clara de lo que significa exactamente. Sin embargo, parece ser que un manejador puede ser instalado y desinstalado.

En la Tabla A.1 podemos comprobar qué conocemos de la especificación estándar sobre la temporización de la instalación y desinstalación de los manejadores.

Esta tabla nos muestra que los manejadores de fallos y terminación sólo se aplican en los estados en ejecución. El manejador de compensación sólo se aplica en el estado completado. El manejador de eventos empieza nuevos eventos mientras se encuentra en estado en ejecución, esto se debe a que el manejador de eventos es "instalado" en paralelo con la ejecución de la actividad principal.

Hay un requisito especial sobre el <scope> que contiene la actividad inicial de comienzo, que el manejador de eventos no debe ser instalado antes de que la actividad inicial de comienzo se haya completado.

Tabla A.1: Instalación y desinstalación de manejadores

	Fallo	**Terminación**	**Compensación**	**Evento**
Instalado	Inicialización del <scope>	- Inicialización del <scope>. - "Terminación forzada. . . se aplica sólo... en modo de procesamiento normal" [43, sección 12.6]	Al finalizar con éxito el único <scope>	- En paralelo con la ejecución de la actividad principal. - Cuando la actividad de inicio inicial se ha completado.
Desinstalado	- Cuando un manejador de fallo está en ejecución, otros manejadores de fallo están desinstalados. - Cuando otro manejador FCT (Fault-Compensation-Termination) está activo.	- Cuando un manejador de fallo está en ejecución. - Cuando otro manejador FCT está activo.	- Cuando el control de la actividad de la actividad de compensación falla o se termina. - En la terminación del manejador FCT contenido. - Cuando falla un manejador de fallo. - En la terminación del manejador de compensación del <scope> padre. - Cuando otro manejador FCT está activo.	- Cuando la actividad principal del <scope> ha terminado. - A los manejadores de eventos que están ejecutándose se les permite finalizar, y el <scope> espera su terminación. - En terminación forzada. - En la activación del manejador de fallo.
Cuando se desinstala		Sin efecto	Cuando se ejecuta una actividad <empty>.	No es posible

A.7.3. Manejador de compensación

Una actividad <scope> debe especificar una actividad de compensación para "deshacer" sus acciones. Por ejemplo, un <scope> cuya actividad principal reservará entradas para el teatro, debería tener una compensación de actividad para cancelar estas reservas.

La actividad de compensación se indica en el manejador de compensación. Dicho manejador sólo puede ser invocado una vez, en un <scope> "completado satisfactoriamente", y desde el manejador de fallos, de compensación o terminación de un <scope> contenido inmediatamente en la jerarquía de <scope>.

Un manejador de compensación se invoca usando la actividad <compensate> para compensar todos los <scope> contenidos inmediatamente, o la actividad <compensateScope> para compensar un <scope> específico.

Un <scope> no puede compensarse a sí mismo y las dos actividades de compensación citadas anteriormente podrían usarse sólo dentro un manejador de fallos, compensación o terminación.

A.7.4. Manejador de fallos

Un fallo tiene un nombre que lo identifica y, opcionalmente, contiene algunos datos del fallo. El manejador de fallos permite capturar un fallo por su nombre, tipo de datos o ambos, y ejecuta la actividad adecuada.

El mecanismo de captura es trivial. Como vemos en el Listado A.5, un manejador de fallos captura el fallo no-estándar x:fatal provocando la salida inmediata del proceso, mientras otro manejador de fallo captura x:ignorable y usa una actividad <empty> para ignorarlo. Finalmente, el <catchAll> captura los otros fallos e inicia la compensación del propio <scope>.

Listado A.5: Ejemplo de una actividad Scope

```
1  <scope name="transfer">
2    <faultHandlers>
3      <catch faultName="x:fatal">
4        <exit />
5      </catch>
6      <catch faultName="x:ignorable">
7        <empty />
8      </catch>
9      <catchAll>
10       <compensate />
11     </catchAll>
12   </faultHandlers>
13
14   <sequence>...</sequence>
```

```
15 </scope>
```

Los fallos no-estándar no son necesarios declararlos anteriormente.

A.7.5. Manejador de eventos

Existen dos tipos de eventos que pueden aparecer en una actividad `<scope>`: eventos de alarma (`onAlarm`) y eventos de mensaje (`onEvent`). Los eventos de alarma son eventos temporizados triviales que se ejecutarán después de un periodo de tiempo especificado o cuando se alcance una fecha límite, opcionalmente repetido varias veces.

Los eventos de mensaje son más complicados. El `<onEvent>` contiene una actividad `<scope>` que se ejecutará cada vez que se reciba el mensaje especificado. Puede recibir varios mensajes durante su tiempo de vida, por tanto, ejecuta varias instancias de su `<scope>` contenido y debe usar agente externo, mensaje de intercambio y conjunto de correlación desde dentro de dicha actividad `<scope>`. Además, el `<onEvent>` dispone de `variable` y atributos `messageType/element` para declarar implícitamente una variable para el mensaje recibido dentro de este `<scope>`.

La variable es explícitamente declarada en el `<scope>`, y se declara una actividad `<receive>` para recibir el mensaje.

A.8. Paso de mensajes

El paso de mensajes no es solamente la única forma de comunicación disponible para los procesos sino que es, además, la única forma de instanciar un proceso. Por tanto, la interfaz del mecanismo del paso de mensajes tiene un gran importancia.

A.8.1. Ligar mensajes

> "Todas las implementaciones WS-BPEL deben ser configurables pudiendo participar en interacciones conforme a *Basic Profile 1.1*" [43, sección 3]

El WS-I proporciona una orientación sobre el intercambio de mensajes entre servicios Web, de forma interoperable. WS-I Basic Profile 1.1 proporciona un conjunto de requisitos concretos para implementaciones de servicios Web.

The Basic Profile se basa en SOAP 1.1, por tanto, se convierte indirectamente en el protocolo recomendado para paso de mensajes en WS-BPEL.

Las variables de mensaje que usamos en WS-BPEL se declaran usando las definiciones de mensajes de WSDL. Estas definiciones no proporcionan ningún detalle sobre cómo es un mensaje concreto, sólo qué tipo de información contienen. SOAP define los mensajes concretos que podemos usar y recibir entre servicios Web. "Ligar" o *binding* es la definición de cómo construir un mensaje concreto desde la definición de mensaje de WSDL.

Basado en el concepto de RPC (Remote Procedure Call), un mensaje en ligadura RPC/literal se construye como una invocación donde las partes del mensaje son argumentos que se pasan a una operación.

El Listado A.6 muestra un fragmento de una definición de servicio Web con una ligadura SOAP RPC/literal. El estilo se declara en `soap:binding` (líneas 19 – 20), pero también podría haberse declarado en `soap:operation` (líneas 22 – 23). Todos los elementos `soap:body` deben tener un atributo `use` con el valor "literal" (líneas 25 – 26, 29 – 30).

Listado A.6: Fragmento WSDL que define un tipo de puerto con ligadura SOAP

```
1  <wsdl:message name="ConversionRateRequest">
2     <wsdl:part name="from" type="xsd:token" />
3     <wsdl:part name="to" type="xsd:token" />
4  </wsdl:message>
5
6  <wsdl:message name="ConversionRateResponse">
7     <wsdl:part name="rate" type="xsd:decimal" />
8  </wsdl:message>
9
10 <wsdl:portType name="CurrencyConvertorPortType">
11    <wsdl:operation name="ConversionRateOperation">
12       <wsdl:input message="tns:ConversionRateRequest" />
13       <wsdl:output message="tns:ConversionRateResponse" />
14    </wsdl:operation>
15 </wsdl:portType>
16
17 <wsdl:binding name="CurrencyConvertorBinding"
18       type="tns:CurrencyConvertorPortType">
19    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
20       style="rpc" />
21    <wsdl:operation name="ConversionRateOperation">
22       <soap:operation
23          soapAction="http://tim.hallwyl.dk/ConversionRateOperation" />
24       <wsdl:input>
25          <soap:body namespace="http://tim.hallwyl.dk/request"
26             use="literal" />
27       </wsdl:input>
28       <wsdl:output>
29          <soap:body namespace="http://tim.hallwyl.dk/response"
30             use="literal" />
31       </wsdl:output>
32    </wsdl:operation>
```

```
33 </wsdl:binding>
```

Listado A.7: Cuerpo de un mensaje de petición SOAP para la operación de conversión

```
1 <foo:ConversionRateOperation xmlns:foo="http://tim.hallwyl.dk/request">
2    <from>DKK</from>
3    <to>USD</to>
4 </foo:ConversionRateOperation>
```

Listado A.8: Cuerpo de un mensaje de respuesta SOAP para la operación de conversión

```
1 <ns1:ConversionRateOperationResponse
2     xmlns:ns1="http://tim.hallwyl.dk/response">
3    <rate>42.424</rate>
4 </ns1:ConversionRateOperationResponse>
```

Cuando se usa RPC/literal todas las partes del mensaje deben ser declaradas usando tipos *Schema*, usando el atributo `type` (líneas 2, 3, 7).

En el mensaje de petición, las partes del mensaje son envueltas en un elemento nombrado con el nombre de la operación. WS-I requiere que este elemento envuelto pertenezca a un espacio de nombre, usando el atributo `namespace` para estar presente en las declaraciones `soap:body`.

El cuerpo del mensaje de petición para la operación de tipo de conversión se muestra en el Listado A.7. Puede observarse cómo el elemento envuelto `ConversionRateOperation` se encuentra en el nombre de espacio definido por la ligadura `soap:body`. Los elementos `from` and `to`, denominados *parts accessor elements* no deben tener ningún espacio de nombres, porque se refieren a tipos, no a elementos. Su nombre local es el nombre de la parte que representan. En este caso, las partes son tipos simples. Si fuesen tipos complejos, entonces los elementos hijos de los *parts accessor elements* deberían pertenecer a un espacio de nombre.

El mensaje de respuesta también sigue las mismas reglas. El nombre local del elemento envuelto, sin embargo, es el nombre de la operación con un postfijo “Response”, y el nombre de espacio está declarado en el `soap:body` para la correspondiente ligadura `wsdl:output` (Listado A.6). El cuerpo del mensaje de respuesta puede observarse en el Listado A.8.

A.8.2. Correlación

La correlación es necesaria para enrutar las instancias de los mensajes de entrada a las actividades. La correlación también es útil para dotar de consistencia a los mensajes enviados y recibidos desde otros servicios Web.

La correlación es necesaria especialmente cuando se realiza un intercambio de un mensaje asíncrono. Veamos la Figura A.4, propuesta en [22].

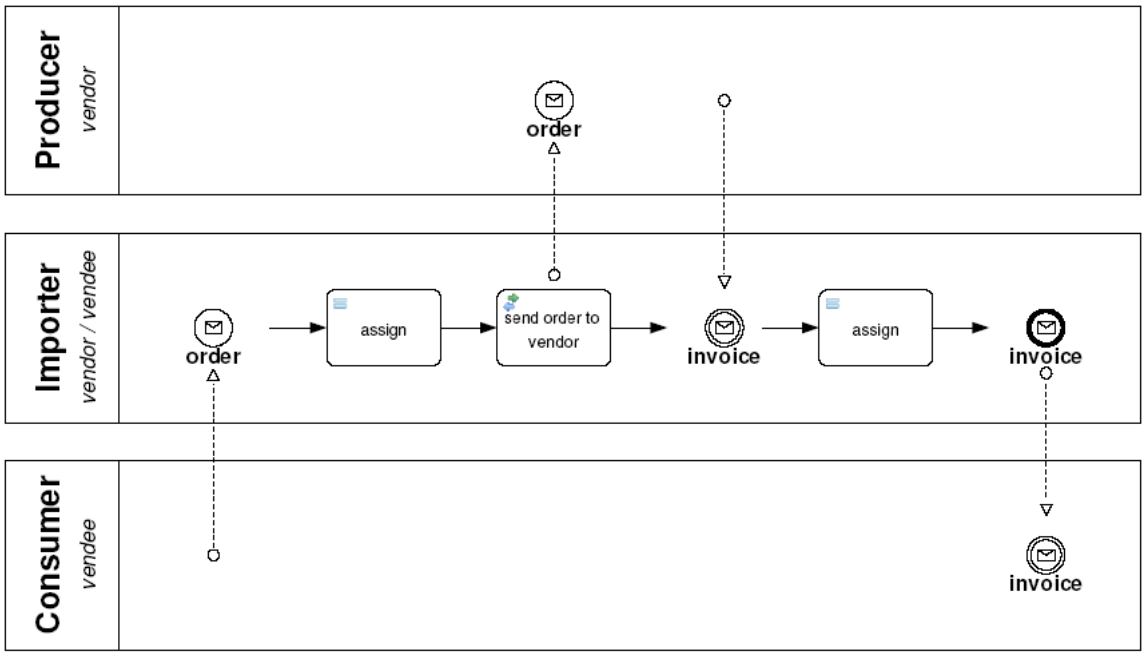

Figura A.4: Intercambio de mensaje asíncrono con correlación

Todas las invocaciones del servicio en este ejemplo son operaciones *one-way*. Consideramos la relación entre el *Importer* y el *Producer*. Primero el *Importer* envía un mensaje de petición al *Producer* invocando la operación `order`. El *Producer* no responde inmediatamente y el *Importer* debe esperar simplemente.

Cuando el *Producer* ha enviado la petición se envía una invocación al *Importer* invocando la operación `invoice`. Si el *Importer* tiene más de una petición con el *Producer*, entonces es necesario un mecanismo para enrutar los mensajes de invocación a la instancia correcta del proceso. Éste es el objetivo principal de la correlación.

Un `<correlationSet>` se declara en el `<process>` o en una actividad `<scope>`, como un conjunto de propiedades de variables de elementos o mensajes de tipo simple. Las propiedades de variable se definen en un documento WSDL importado con un conjunto de alias de propiedad.

El Listado A.9 muestra la definición de la propiedad `orderId` utilizada por el *Importer* (línea 3). La propiedad `orderId` aparece dentro de un mensaje de pedido y un mensaje de invocación, definidos por los dos alias de propiedad (líneas 4 – 14).

Listado A.9: Propiedades y alias

```
1 <wsdl:definitions  ... >
2     ...
3     <vprop:property name="orderId" type="xsd:integer" />
4     <vprop:propertyAlias
5         propertyName="tns:orderId"
6         messageType="tns:orderMessage" part="order">
```

```
7        <vprop:query>sch:id</vprop:query>
8     </vprop:propertyAlias>
9
10    <vprop:propertyAlias
11       propertyName="tns:orderId"
12       messageType="tns:invoiceMessage" part="invoice">
13       <vprop:query>sch:order</vprop:query>
14    </vprop:propertyAlias>
15    ...
16 </wsdl:definitions>
```

La descripción del proceso *Importer* se describe en el Listado A.10. Define dos `<correlationSet>`, cada uno con una única propiedad. Ambos utilizan la misma propiedad pero en diferentes mensajes.

Listado A.10: Descripción del proceso Importer utilizando correlaciones

```
1  <process name="importer" ...>
2     ...
3     <correlationSets>
4        <correlationSet name="orderMatch" properties="srv:orderId" />
5        <correlationSet name="orderinvoice" properties="srv:orderId" />
6     </correlationSets>
7
8     <sequence>
9        <receive ... createInstance="yes">
10          <correlations>
11             <correlation set="orderMatch" initiate="yes" />
12          </correlations>
13       </receive>
14       ...
15       <invoke ...>
16          <correlations>
17             <correlation set="orderinvoice" initiate="yes"
18                pattern="request" />
19          </correlations>
20       </invoke>
21
22       <receive ... >
23          <correlations>
24             <correlation set="orderinvoice" initiate="no" />
25          </correlations>
26       </receive>
27       ...
28       <invoke ...>
29          <correlations>
30             <correlation set="orderMatch" initiate="no" />
31          </correlations>
32       </invoke>
33    </sequence>
34 </process>
```

El conjunto `orderMatch` se utiliza para correlacionar el mensaje de petición recibida desde el *Consumer* con el mensaje de invocación enviado de vuelta al final del proceso. Se trata de una comprobación de consistencia pura, no se necesita, por ejemplo, de enrutamiento.

En el conjunto `orderinvoice` se utiliza el enrutamiento asegurándose de que el mensaje de invocación llega a la instancia correcta.

El conjunto de correlación en tiempo de ejecución se inicia con una actividad de mensaje. Esto quiere decir que almacena los valores de la propiedad actual utilizados en el mensaje que se envía o recibe (o ambos). El conjunto `orderMatch` se inicia con la primera actividad `<invoke>` (líneas 15 – 20) mediante el almacenamiento del `orderId` desde el mensaje de petición.

Cuando llega un mensaje de petición para las siguientes actividades `<receive>` (líneas 22 – 26) la propiedad `orderId`, que se encuentra en el mensaje de petición, se compara con el valor almacenado en el conjunto de correlación. La instancia con la correlación relacionada recibe el mensaje.

La **semántica de correlación** se basa en dos restricciones: la inicialización y las restricciones de consistencia. La restricción de la consistencia se define como:

> "Después de que un conjunto de correlación se inicializa, los valores de las propiedades de un conjunto de correlación deben ser idénticos para todos los mensajes de todas las operaciones que llevan el conjunto de correlación y se producen en el ámbito correspondiente hasta su finalización." [43, sección 9.2]

La restricción de inicialización define el comportamiento basado en el valor del atributo `initiate`. Existen tres opciones: cuando tiene valor "yes" el conjunto de correlación se debe inicializar o lanzar un fallo si ya se ha inicializado. Cuando tiene el valor "join" se intenta la inicialización de la correlación y sólo lanzará un fallo si ya se ha inicializado y la restricción de consistencia se ha violado. Cuando tiene el valor "no" el conjunto de correlación debe haber sido inicializado previamente y obedecer a la restricción de consistencia, en caso contrario lanzará un fallo.

A.8.3. Instanciación del Proceso

La única manera de instanciar un proceso es mediante la recepción de un mensaje. La especificación estándar define una "start activity" como una actividad `<receive>` o `<pick>`, anotada con un atributo `createInstance = "yes"` [43, sección 10.4]. Se permite tener múltiples actividades de inicio, pero se requiere al menos una. La actividad de inicio inicial se define como la actividad de inicio que provocó que una instancia de proceso particular fuese instanciada. [43, sección 10.4].

Hay algunas restricciones importantes sobre la declaración de actividades de inicio, que tienen influencia sobre cómo se gestionan los mensajes entrantes.

> "Las actividades de no-inicio, excepto `<scope>`, `<flow>`, `<sequence>` o las actividades `<extensionActivity>` deben tener una dependencia de control de una actividad de inicio". [43, sección 10.4]

La definición de una dependencia de control es:

> "Si una actividad A debe completarse antes de que la actividad B comience, como consecuencia de la existencia de una vía de control de A a B en la definición del proceso, entonces decimos que B tiene una dependencia de control sobre A". [43, sección 12.5.2]

Si vemos una descripción del proceso como una estructura de árbol, esto quiere decir que los antecesores de una actividad de inicio deben ser las actividades `<scope>`, `<flow>` o `<sequence>`.

Por tanto, una actividad de inicio es una actividad `<receive>` o `<pick>`. A excepción de las actividades `<scope>`, `<flow>` y `<sequence>`, las primeras actividades en la petición de ejecución de la descripción del proceso deben ser actividades de inicio. En un proceso con más de una actividad de inicio, la actividad de inicio inicial debe completarse antes de que otras actividades de inicio comiencen.

B XSLT 2.0

XSLT es un lenguaje que, de acuerdo con su especificación [27], es diseñado principalmente para transformar un documento XML en otro. Sin embargo, XSLT también permite transformar XML a HTML y a cualquier otro formato basado en texto. Por tanto, una definición más general de XSLT es:

> "XSLT es un lenguaje que permite transformar la estructura y el contenido de un documento XML." [28]

XML es una forma estándar y simple de intercambiar datos de texto estructurados entre programas de ordenador. Este lenguaje tiene tanto éxito porque también puede ser leído y escrito por los humanos, utilizando un simple editor de textos. XML satisface dos requisitos:

- **Separación de datos de la presentación**: la necesidad de separar la información de los detalles de la forma en la que se presenta en un disposito en particular.
- **Transmitir datos entre aplicaciones**: la necesidad para transmitir información, como peticiones, desde una organización a otra sin invertir en proyectos de integración única de software.

XSLT 2.0 fue publicado como una recomendación W3C en Enero de 2007. Esta versión aporta sobre la versión anterior, XSLT 1.0, las siguientes características:

- **Integración a través de la familia de estandares XML**
- **Extensión del ámbito de aplicación**
- **Ingeniería de software más robusta**, por ejemplo, en cuanto al manejador de fallos y el chequeo de tipos.
- **Mejora de la usabilidad**

B.1. Motores XSLT

La función principal de un motor XSLT es aplicar una hoja de estilos XSLT a un documento fuente XML y producir un documento resultante.

Algunos de los motores disponibles para XSLT 2.0 son:

- **Saxon**: Disponible con dos variantes, hasta la fecha, que corresponden con los dos niveles de conformidad definidos en la especificación W3C:
  - **Saxon-B 9.1**: una implementación de código abierto de un motor XSLT 2.0 básico (http://saxon.sf.net/).
  - **Saxon-SA 9.1**: una implementación comercial de un motor XSLT con esquemas (http://www.saxonica.com/).

  Las dos variantes están disponibles para las plataformas Java y .NET. **Saxon** ofrece una interfaz de línea de comandos y una API (Application Programming Interface) de Java o .NET. Existen un conjunto de herramientas que ofrecen interfaces de usuarios gráficas: el producto **Kernow** de código abierto (kernowforsaxon.sourceforge.net), el entorno de desarrollo **Stylus Studio** (www.stylusstudio.com) y el editor **oXygen XML** (www.oxygenxml.com). Estos dos últimos soportan depuración de Saxon paso a paso.
- **Altova** (www.altova.com): ha lanzado su propio motor XSLT 2.0, que está disponible como un componente COM con una interfaz de línea de comando y API de Windows, o construido en el entorno de desarrollo **XML Spy**. Como único componente este producto es gratis, aunque no es código abierto, y ofrece ambos motores con y sin esquemas.
- **Gestalt** es un motor XSLT 2.0 de código abierto escrito en el lenguaje Eiffel por Colin Adams (http://sourceforge.net/projects/gestalt). Los inconvenientes son que el producto está en progreso y la documentación es escasa.

B.2. El modelo de procesamiento XSLT

Podemos pensar que XSLT realiza un proceso de transformación porque la salida (el documento resultante) es del mismo tipo de objeto que la entrada (el documento fuente). Esto tiene beneficios intermedios: por ejemplo, es posible hacer una transformación compleja como una serie de transformaciones simples, y es posible hacer transformaciones en cualquier dirección utilizando la misma tecnología.

El nombre de hoja de estilos (*stylesheet*) parece ser no muy adecuado para el documento que define la transformación, puesto que XSLT se utiliza, a menudo, para tareas que no tienen nada que ver con el estilo. El nombre refleja la realidad de que un mismo tipo de transformación realizada con XSLT define un estilo para la información del documento fuente, por lo que el documento resultante contiene información del documento fuente con más información, dependiendo del disposito de salida que vaya a mostrar dicha información.

XSLT define sus operaciones en términos de un modelo de datos, denominado XDM (XML Data Model), en el que un documento XML se representa como un árbol (*tree*). El árbol es un tipo de datos abstracto. No hay definido una API ni tampoco la representación de datos, sólo un modelo conceptual que define los objetos en el árbol, sus propiedades, y sus relaciones. El árbol XDM es similar en concepto al W3C DOM, excepto en que el DOM tiene una API definida. De hecho, algunos implementadores utilizan el DOM como su estructura de árbol interna. Otros usan una estructura de datos que está más cerca de la especificación XDM, mientras algunos usan las estructuras de datos internas optimizadas que están poco relacionadas con este modelo. Se describe un modelo conceptual, no algo que necesariamente debe existir en una implementación.

El modelo de datos para los árboles XSLT es compartido con las especificaciones XPath y XQuery (XML Query), que aseguran que los datos pueden intercambiarse entre estos tres lenguajes. Es principal la relación entre XSLT y XPath porque XSLT siempre recupera los datos de un documento fuente ejecutando expresiones XPath.

Las reglas de conformidad formal dicen que un motor XSLT debe ser capaz de leer una hoja de estilos y utilizarla para transformar un árbol fuente en un árbol resultante. Ésta es la parte del sistema mostrado en la Figura B.1, extraída de [28], en el rectángulo con bordes redondeados. No existen requisitos "oficiales" para manipular las partes del proceso mostrado dentro del rectángulo, como la creación de un árbol fuente desde un documento XML fuente, conocido como *parsing*, o la creación de un documento XML resultante desde un árbol resultante, denominado *serialization*. En la práctica, la mayoría de los productos reales también manejan estas partes.

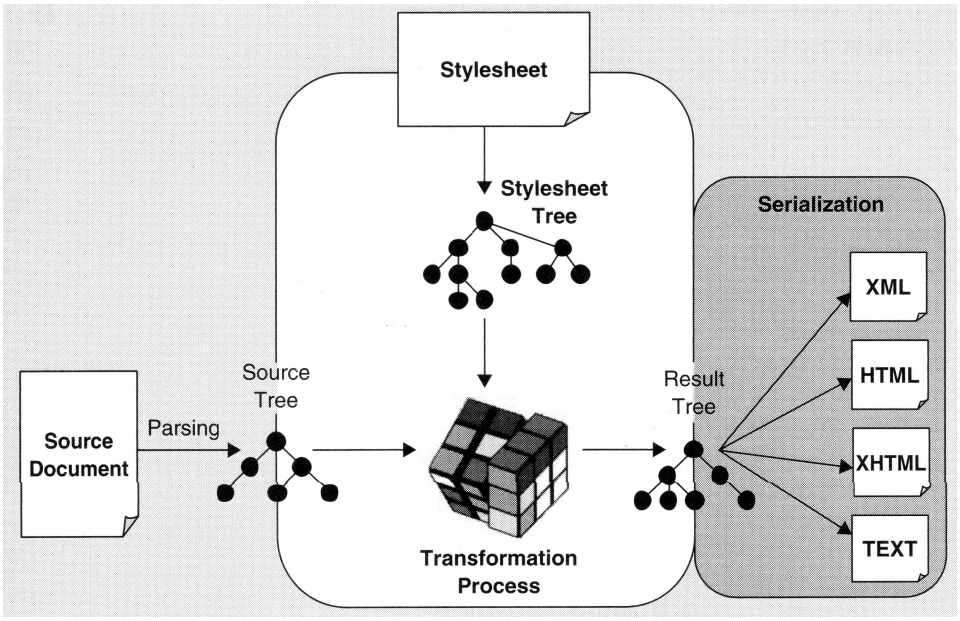

Figura B.1: Modelo de procesamiento XSLT

Aunque el proceso final de convertir el árbol resultante a un documento de salida está fuera de la conformidad de las reglas del estándar XSLT, esto no signifca que XSLT no tenga que decir nada al respecto.

El principal control sobre este proceso es el elemento `<xsl:output>`. Este elemento define cuatro formatos o métodos de salida: `xml`, `html`, `xhtml` y `text`. En cada caso un árbol resultante se escribe a un fichero de salida concreto.

Un árbol XDM está compuesto de nodos. Existen siete tipos de nodos. Los diferentes tipos de nodos corresponden con los componentes del documento XML fuente:

Document El nodo documento es un nodo simple, hay uno para cada documento. No debe confundirse el "nodo" documento con el "elemento" documento, que en un documento bien formado es el elemento que contiene a todos los demás. Un nodo documento nunca tiene un padre, por tanto, siempre es la raíz del árbol.

Element Un elemento es una parte de un documento limitado por etiquetas de comienzo y fin, o representado por una etiqueta de elemento vacío como `<etiqueta/>`.

Text El nodo texto es una secuencia consecutiva de caracteres en una parte `PCDATA` de un elemento. Los nodos texto siempre se construyen tan grandes como sea posible: nunca habrá dos nodos texto adyacentes en el árbol, porque siempre están fusionados en uno solo (ésta es la teoría, algunas implementaciones no siguen siempre esta regla).

Attribute El nodo atributo incluye el nombre y el valor de un atributo escrito dentro de una etiqueta de comienzo de elemento (o etiqueta de elemento vacío). Un atributo que no sea presentado en la etiqueta, pero que tenga un valor por defecto definido en el DTD (Document Type Definition) o Schema, también se representa como un nodo atributo en cada instancia de elemento separada. Una declaración de espacio de nombre (un atributo cuyo nombre es `xmlns` o cuyo nombre comienza con `xmlns:`) no es, sin embargo, representado por un nodo atributo en el árbol.

Comment Un nodo comentario representa un comentario escrito en el documento XML fuente entre los delimitadores `<!--` y `-->`.

Processing instruction Un nodo de instrucción de procesamiento representa una instrucción de procesamiento escrita en el documento XML fuente entre los delimitadores `<?` y `?>`. El `PITarget` desde el fuente XML se toma como el nombre del nodo y el resto del contenido como su valor. Destacar que la declaración XML `<?xml version="1.0"?>` no es una instrucción de procesamiento, incluso aunque aparezca sola no se representa como un nodo en el árbol.

Namespace Un nodo de espacio de nombre representa una declaración de espacio de nombre, excepto que se copia a cada elemento que se aplica. Por tanto, cada nodo elemento tiene un nodo de espacio de nombre para cada declaración de espacio de nombre que está en el ámbito para ese elemento. Los nodos de nombre de

espacio pertenecientes a un elemento son distintos de los que pertenecen a otro elemento, incluso cuando son derivados de la misma declaración de espacio de nombre en el documento fuente.

Existen siete posibles formas para clasificar estos nodos. Podemos distinguir los que pueden tener hijos (nodos elemento y documento), los que pueden tener un padre (todos, excepto el nodo documento), los que tienen un nombre (elementos, atributos, espacio de nombre e instrucciones de procesamiento) o los que tienen su propio contenido textual (atributos, texto, comentarios, instrucciones de procesamiento y nodos de espacio de nombre). Puesto que cada uno de estos criterios da una jerarquía de clase posible diferente, el modelo XDM deja la jerarquía plana, y define todas estas características para todos los nodos. Donde una característica no es aplicable a un tipo de nodo particular, XDM define generalmente su valor como una secuencia vacía, aunque a veces cuando se accede a la propiedad desde una expresión XPath real lo que se obtiene sea una cadena de longitud cero.

Por tanto, si utilizamos una jerarquía de clases en notación UML, obtendremos el diagrama simplificado mostrado en la Figura B.2.

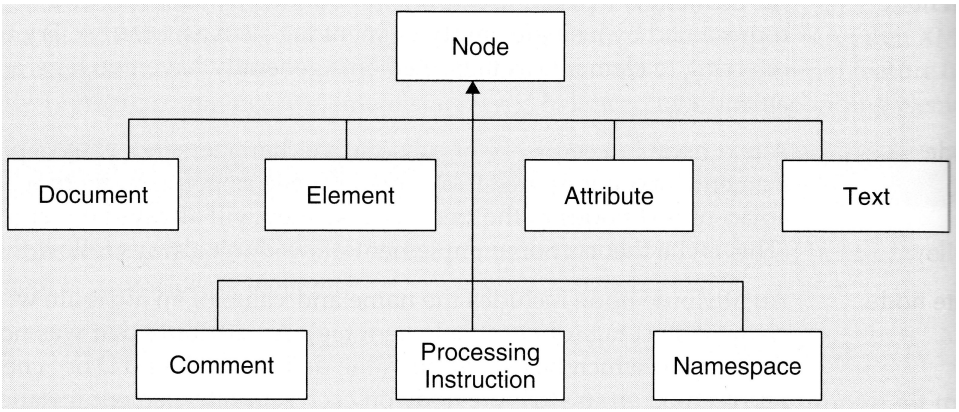

Figura B.2: Jerarquía de clases en UML de nodos en el Modelo de Árbol

En este diagrama, las cajas representan las clases o tipos, y la flecha representa una relación *es-un-tipo-de*, por ejemplo, un comentario *es-un-tipo-de* nodo.

B.3. Procesamiento *push*

La forma más simple de procesar un árbol fuente consiste en escribir un regla de plantilla (*template*) para cada tipo de nodo que puede encontrarse, y para esa regla de plantilla para producir cualquier salida requerida, además de llamar a `<xsl:apply-templates>` para procesar los hijos de ese nodo.

El siguiente ejemplo extraído de [28] demuestra la técnica de procesamiento *push*: una hoja de estilos basada en reglas en la que hay una regla de plantilla para procesar cada tipo de nodo diferente.

El Listado B.1 muestra el documento fuente, `books.xml`, que es un simple catálogo de libros.

Listado B.1: `books.xml`

```
<?xml version="1.0"?>
<books>
  <book category="reference">
      <author>Nigel Rees</author>
      <title>Sayings of the Century</title>
      <price>8.95</price>
   </book>
   <book category="fiction">
      <author>Evelyn Waugh</author>
      <title>Sword of Honour</title>
      <price>12.99</price>
   </book>
   <book category="fiction">
      <author>Herman Melville</author>
      <title>Moby Dick</title>
      <price>8.99</price>
   </book>
   <book category="fiction">
      <author>J. R. R. Tolkien</author>
      <title>The Lord of the Rings</title>
      <price>22.99</price>
   </book>
</books>
```

Supongamos que deseamos mostrar los datos del catálogo de libros de una forma concreta: una lista de libros enumerada secuencialmente. Para ello, debemos utilizar una hoja de estilos, `books.xsl`, como la que mostramos en el Listado B.2.

Listado B.2: `books.xsl`

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="books">
  <html><body>
    <h1>A list of books</h1>
    <table width="640">
      <xsl:apply-templates/>
    </table>
  </body></html>
</xsl:template>

<xsl:template match="book">
  <tr>
```

```
15      <td><xsl:number/></td>
16      <xsl:apply-templates/>
17    </tr>
18  </xsl:template>
19
20  <xsl:template match="author | title | price">
21    <td><xsl:value-of select="."/></td>
22  </xsl:template>
23
24  </xsl:stylesheet>
```

En este caso, no hay una regla de plantilla para el nodo documento, por tanto, se invoca la plantilla por omisión. Ésta procesa todos los hijos del nodo documento.

Hay un único hijo del nodo documento, el elemento `<books>`. Por tanto, la regla de plantilla para el elemento `<book>` se evalúa. Ésta crea algunos elementos HTML estándar en el árbol resultante, y eventualmente llama a `<xsl:apply-templates/>` para provocar que sus propios hijos sean procesados. Estos hijos son todos elementos `<book>`, por ello, son todos procesados por la regla de plantilla cuya patrón de relación es `match="book"`. Esta regla de plantilla produce un elemento `<tr>` HTML, y dentro un elemento `<td>`, que se rellena con la ejecución de la instrucción `<xsl:number/>` cuyo efecto es obtener el número de secuencia del nodo actual (el elemento `<book>`) dentro de su elemento padre. Entonces llama a `<xsl:apply-templates/>` otra vez para procesar los hijos del elemento `<book>` en el árbol fuente.

Los hijos del elemento `<book>` en el documento fuente son todos elementos `<author>`, `<title>` o `<price>`; por tanto, todos están relacionados con la regla de plantilla cuyo patrón de relación es `match="author|title|price"` (puede leerse `|` como "or"). Esta regla de plantilla produce un elemento `<td>` HTML que se rellena con la ejecución de una instrucción `<xsl:value-of select="."/>`. Esta instrucción evalúa una expresión XPath, y escribe su resultado (una cadena) como texto en el árbol resultante. La expresión es `.`, que devuelve el valor cadena del nodo actual, que es el contenido textual del elemento actual `<author>`, `<price>` o `<title>`.

Esta plantilla no llama a `<xsl:apply-templates>`, por tanto, sus hijos no serán procesados, y el control se devuelve hacia arriba.

En el Listado B.3 podemos comprobar la salida que se ha obtenido al aplicar la hoja de estilos `books.xsl` al documento fuente `books.xml`.

Listado B.3: `output.html`

```
1  <html>
2    <body>
3      <h1>A list of books</h1>
4      <table width="640">
5        <tr>
6          <td>1</td>
```

```
7         <td>Nigel Rees</td>
8         <td>Sayings of the Century</td>
9         <td>8.95</td>
10      </tr>
11      <tr>
12        <td>2</td>
13        <td>Evelyn Waugh</td>
14        <td>Sword of Honour</td>
15        <td>12.99</td>
16      </tr>
17      ...
18    </table>
19  </body>
20 </html>
```

Este estilo de procesamiento se llama procesamiento *push*. Es dirigido por la instrucción `<xsl:apply-templates>`, como si el motor estuviese empujando los nodos fuera de la puerta, preguntando "si hay alguien interesado en relacionarse con éste".

B.4. Controlando qué nodos procesar

El procesamiento *push* funciona muy bien cuando los datos de salida tendrán la misma estructura y secuencia que los datos de entrada, y todo lo que queremos hacer es añadir o eliminar algunas etiquetas o modificar algunos valores.

El ejemplo anterior no habría funcionado tan bien si las propiedades de cada libro fuesen menos predecibles, por ejemplo, si algunos de los libros no tuviesen precio, o si el título y el autor puedieran aparecer en cualquier orden. En este caso la tabla de HTML que habríamos obtenido no hubiese sido la esperada.

Ante estas circunstancias existen dos opciones:

- Ser más preciso sobre qué nodos procesar, en lugar de procesar todos los hijos del nodo actual.
- Ser más preciso sobre cómo procesarlos, en lugar de elegir la mejor regla de plantilla.

Veamos la primera opción con el ejemplo definido en [28]. Podemos tener más control sobre *qué* nodos serán procesados cambiando la plantilla `<book>` en `books.xsl`, como puede observarse en el Listado B.4.

Listado B.4: Controlando la secuencia de procesamiento

```
1 <xsl:template match="book">
2   <tr>
3     <td><xsl:number/></td>
4     <xsl:apply-templates select="author, title, price"/>
```

```
5   </tr>
6 </xsl:template>
```

En lugar de seleccionar todos los elementos hijos y encontrar la regla de plantilla apropiada para cada uno, ahora éste selecciona explícitamente el primer elemento hijo `<author>`, entonces el elemento hijo `<title>` y, a continuación, el elemento hijo `<price>`.

Esta forma de procesamiento es más robusta que la anterior. No obstante, producirá una tabla irregular si hay algunos elementos `<book>` sin un `<author>`, o con más de uno.

El operador coma utilizado en la expresión `select="author, title, price"` es nuevo en XPath 2.0. Simplemente concatena varias secuencias (que deben ser elementos simples, pero también podría estar vacía, o contener elementos múltiples) en una secuencia, en el orden especificado.

Como deseamos una salida con una estructura regular y conocemos bastante sobre la estructura del documento fuente, probablemente en esta situación sea mejor definir todos los procesamientos en la plantilla `<book>`, en lugar de depender de reglas de plantilla para relacionar cada uno de sus elementos hijos.

Podemos conseguir mayor control sobre *cómo* los nodos serán procesados escribiendo la regla de plantilla `<book>` como puede verse en el Listado B.5.

Listado B.5: Seleccionando nodos explícitamente

```
1 <xsl:template match="book">
2   <tr>
3     <td><xsl:number/></td>
4     <td><xsl:value-of select="author"/></td>
5     <td><xsl:value-of select="title"/></td>
6     <td><xsl:value-of select="price"/></td>
7   </tr>
8 </xsl:template>
```

Algunos llaman a esto procesamiento *pull*, porque en lugar de que la plantilla empuja los nodos fuera de la puerta para que sean recogidos por otra plantilla, este procesamiento consite en "tirar de" los nodos y manejarlos.

El estilo de relación de patrones (o `push`) de procesamiento es el más característico de XSLT, y funciona muy bien en aplicaciones donde se necesita describir la manipulación de cada tipo de nodo en el documento fuente independientemente. Sin embargo, existen otras técnicas. Dentro de una regla de plantilla que procesa un nodo particular, las alternativas principales si deseamos acceder a la información de otros nodos son las siguientes:

- Llamar a `<xsl:apply-templates>` para procesar estos nodos utilizando sus reglas de plantilla apropiadas.

- Llamar a <xsl:apply-templates> en un *modo* en particular para procesar estos nodos utilizando las reglas de plantilla para el modo relevante.
- Llamar a <xsl:value-of> para extraer la información requerida desde los nodos directamente.
- Llamar a <xsl:for-each> para realizar procesamiento explícito de cada uno de los nodos por turno.
- Llamar a <xsl:call-template> para invocar una plantilla específica por nombre, en lugar de depender de la relación de patrón para decidir qué plantilla invocar.

B.5. Modos

A veces queremos procesar el mismo nodo del árbol fuente más de una vez, de diferentes maneras. El ejemplo clásico es producir una tabla de contenidos. Cuando se genera la tabla de contenidos, deseamos manejar todas las cabeceras de sección de una forma, y cuando producimos el cuerpo del documento, queremos tratarlas de diferente manera.

Una forma de solucionar este problema es usar un procesamiento *push* en uno de estos pasos a tráves de los datos, y procesamiento *pull* en las demás ocasiones. Sin embargo, esto podría estar muy restringido. En vez de esto, podemos definir diferentes modos de procesamiento, uno para cada paso a través de los datos. Podemos nombrar el modo de procesamiento cuando llamamos a <xsl:apply-templates>, y las únicas reglas de plantilla que se tendrán en cuenta serán aquellas que especifican el mismo modo. Por ejemplo, si especificamos:

```
<xsl:apply-templates select="heading-1" mode="table-of-contents"/>
```

Entoces la regla de plantilla seleccionada deberá ser una que esté definida como:

```
<xsl:template match="heading-1" mode="table-of-contents">
...
</xsl:template>
```

B.6. Reglas de plantilla por omisión

¿Qué ocurre cuando se invoca <xsl:apply-templates> para procesar un nodo, y no hay una regla en la hoja de estilos que se relacione con ese nodo? En este caso, se invoca una regla de plantilla por omisión.

Existe una regla de plantilla por omisión para cada tipo de nodo. Las reglas por omisión funcionan de la siguente manera:

- Nodo documento: llama a `<xsl:apply-templates>` para procesar los hijos del nodo documento, en el mismo modo como el modo de llamada.
- Nodo elemento: llama a `<xsl:apply-templates>` para procesar los hijos de este elemento, en el mismo modo como el modo de llamada.
- Nodo atributo: copia el valor del atributo al árbol resultante, como texto y no como un nodo atributo.
- Nodo texto: copia el texto al árbol resultante.
- Nodo comentario: la regla de plantilla por omisión no hace nada.
- Nodo instrucción de procesamiento: la regla de plantilla por omisión no hace nada.
- Nodo espacio de nombre: la regla de plantilla por omisión no hace nada.

La reglas de plantilla por omisión sólo serán invocadas si no existe ninguna regla que relacione el nodo en ninguna parte de la hoja de estilos.

C Operadores de mutación

C.1. Operadores de mutación para ANSI C

Esta sección describe los operadores de mutación que se han definido en [2, 8] para el lenguaje ANSI C. Estos operadores se clasifican en cuatro categorías:

- Sentencias
- Operadores
- Variables
- Constantes

C.1.1. Operadores de sentencias

STRP Interrupción de una sentencia. Cada sentencia se reemplaza por `trap_on_statement()`. Cuando se ejecuta un `trap_on_statement` la ejecución del mutante termina. El mutante se considera muerto.

STRI Interrupción en una condición `if`. Este operador genera dos mutantes para cada sentencia `if`: `if(trap_on_true(v))` y `if(trap_on_false(v))`. Cuando `trap_on_true` (`trap_on_false`) se ejecuta, se mata al mutante si el valor del argumento función es `true` (`false`). Si el valor del argumento no es `true` (`false`), la función devuelve `false` (`true`) y la ejecución del mutante continúa.

SSDL Elimina una sentencia.

SRSR Intercambio de una sentencia por `return`.

SGLR Sustitución de etiquetas `GOTO`.

SCRB Cambia la sentencia `continue` por un `break`.

SBRC Cambia la sentencia `break` por un `continue`.

SBRn Cambia los `break` o `continue` por la función `break_out_to_level_n()` que termina con los *n* bucles.

SCRn Cambia los `break` o `continue` por la función `continue_out_to_level_n()` que continua en el bucle *n*.

SWDD Cambia la sentencia `while` por un `do-while`.

SDWD Cambia la sentencia `do-while` por un `while`.

SMTT Asegura que cada bucle se ejecuta al menos 2 veces.

SMTC Asegura que cada bucle se ejecuta al menos 2 veces, y asegura que tiene efecto en la salida.

SSOM Cambia el orden de expresiones separadas por (`,`).

SMVB Desplaza una (`}`) una línea de código arriba o abajo.

SSWM Errores a la hora de declarar un `switch`.

C.1.2. Operadores de operadores

Los operadores de operadores se clasifican en dos categorías:

- Mutación de operador binario:
  - Reemplazo de operador comparable
  - Reemplazo de operador incomparable
- Mutación de operaror unario:
  - Reemplazo de incremento/decremento
  - Operadores de negación
  - Mutación de operador de indirección
  - Reemplazo de operador `cast`

Operadores de reemplazo de operador comparable

OAAA Cambio en una asignación aritmética.

OAAN Cambio de un operador aritmético.

OBBA Cambio en una asignación de manejo de bits (`&=`, `^ =`, `|=`). Por ejemplo, al aplicar este operador a `a&=b` genera el mutante `a|=b`.

OBBN Cambio de un operador de manejo de bits (`&`, `^`, `|`). Por ejemplo, al aplicar este operador a `a&b` genera el mutante `a|b`.

OLLN Cambio de operador lógico.

ORRN Cambio de operador relacional.

OSSA Cambio en una asignación de desplazamiento.

OSSN Cambio de operador de desplazamiento.

Operadores de reemplazo de operador incomparable

OABA Cambio de asignación aritmética por asignación de manejo de bits.
OAEA Cambio de asignación aritmética por asignación.
OABN Cambio de operador aritmético por operador de manejo de bits.
OALN Cambio de operador aritmético por operador lógico.
OARN Cambio de operador aritmético por operador relacional.
OASA Cambio de asignación aritmética por asignación de desplazamiento.
OASN Cambio de operador aritmético por operador de desplazamiento.
OBAA Cambio de asignación de manejo de bits por asignación aritmética.
OBAN Cambio de operador de manejo de bit por operador aritmético.
OBEA Cambio de asignación de manejo de bits por asignación.
OBLN Cambio de operador de manejo de bit por operador lógico.
OBRN Cambio de operador de manejo de bit por operador relacional.
OBSA Cambio de asignación de manejo de bits por asignación de desplazamiento.
OBSN Cambio de operador de manejo de bit por operador de desplazamiento.
OEAA Cambio de asignación por asignación aritmética.
OEBA Cambio de asignación por asignación de manejo de bits.
OESA Cambio de asignación por asignación de desplazamiento.
OLAN Cambio de operador lógico por operador aritmético.
OLBN Cambio de operador lógico por operador de manejo de bits.
OLRN Cambio de operador lógico por operador relacional.
OLSN Cambio de operador lógico por operador de desplazamiento.
ORAN Cambio de operador relacional por operador aritmético.
ORBN Cambio de operador relacional por operador de manejo de bits.
ORLN Cambio de operador relacional por operador lógico.
ORSN Cambio de operador relacional por operador de desplazamiento.
OSAA Cambio de asignación de desplazamiento por asignación aritmética.
OSAN Cambio de operador de desplazamiento por operador aritmético.
OSBA Cambio de asignación de desplazamiento por asignación de manejo de bits.
OSBN Cambio de operador de desplazamiento por operador de manejo de bits.
OSEA Cambio de asignación de desplazamiento por asignación.
OSLN Cambio de operador de desplazamiento por operador lógico.
OSRN Cambio de operador de desplazamiento por operador relacional.

Operadores de reemplazo de incremento/decremento

OPPO Cambia operador de pre-incremento por post-incremento o viceversa.

OMMO Cambio operador de pre-decremento por post-decremento o viceversa.

Operadores de reemplazo de operadores de negación

OLNG Niega una de las partes en una condición lógica.

OBNG Niega una de las partes en una expresión de manejo de bits.

OCNG Niega la condición de una sentencia iterativa o selectiva.

Operadores de mutación de operador de indirección

OPIM Trata con el operador de indirección (`*`).

Operadores de reemplazo de operador `cast`

OCOM Cambia los tipos de variables.

C.1.3. Operadores de variables

Los operadores de variables se clasifican en cinco categorías:

- Reemplazo de Servicio de variable escalar
- Reemplazo de referencia de vector
- Reemplazo de referencia de estructura
- Reemplazo de referencia de puntero
- Otros

Operadores de Servicio de variable escalar

VGSR Cambio en variables escalares globales que afecten a una función `f`.

VLSR Cambio en variables escalares pasadas como parámetros a la función `f`.

Operadores de referencia de vector

VGAR Cambio de variables globales que sean vectores que afecten a una función `f`.

VLAR Cambio de variables que sean vectores que afecten a la función `f`.

Operadores de referencia de estructura

VGTR Cambio en variables globales de tipo `struct` que afecten a una función `f`.

VLTR Cambio de variables de tipo `struct` que afecten a la función `f`.

Operadores de referencia de puntero

VGPR Cambio en variables globales de tipo puntero que afecten a una función `f`.

VLPR Cambio de variables de tipo puntero que afecten a la función `f`.

Otros operadores

VSCR Cambio en la variable de un tipo `struct`.

VASM Errores en la referencia de un elemento de un vector multidimensional.

VDTR Cambia todo los valores de un tipo escalar por valores negativos, cero o positivos.

VTWD Incrementa o decrementa en 1 el valor de una variable o expresión.

C.1.4. Operadores de constantes

Los operadores de constantes se clasifican en tres categorías:

- Reemplazo de constante por constante
- Reemplazo de constante por escalares
- Otros

Operadores de reemplazo de constante por constante

CGCR Cambio de una constante global por otra en una función `f`.

CLCR Cambio de una constante pasada como parámetro a la función `f` por otra.

Operadores de reemplazo de constante por escalar

CGSR Cambio de una variable escalar global por una constante global en una función `f`.

CLSR Cambio de una variable escalar pasada como parámetro a la función `f` por otra constante.

Otros operadores

CRCR Cambia los valores de una constante por el conjunto {0, 1, -1, u}.

C.2. Operadores de mutación para C#

Esta sección describe los operadores de mutación que se han definido en [9, 10] para el lenguaje C#. Estos operadores se clasifican en diez categorías:

- Errores típicos de los programadores
- Ocultación de información
- Herencia
- Polimorfismo y construcción dinámica
- Sobrecarga de métodos
- Características de lenguajes orientado a objetos
- Excepciones
- Delegación de métodos
- Propiedad de reemplazo
- Otros

C.2.1. Operadores de errrores típicos de los programadores

EOA Reemplazamiento de una asignación referenciada por el contenido de la misma asignación.

EOC Reemplazamiento de una comparación referenciada por el contenido de la misma comparación.

EAM Cambio del método de acceso.

EMM Cambio del método modificador.

C.2.2. Operadores de ocultación de información

AMC Cambio del modificador de acceso.

C.2.3. Operadores de herencia

IHD Eliminación de la variable *hiding*.

IHI Inserción de la variable *hiding*.

IOD Eliminación del método *overriding*.

IOP Cambio de la posición de llamada del método *overriding*.

IOR Renombrado del método *overriding*.

ISK Eliminación de la palabra clave `super`.

IPC Llamada explícita a la eliminación del constructor padre.

C.2.4. Operadores de polimorfismo y construcción dinámica

PNC Llamada del método `new` con un tipo de la clase hijo.

PMD Declaración de variable miembro con el tipo de la clase padre.

PPD Declaración de un parámetro variable con el tipo de la clase hijo.

PRV Reemplazo de una referencia con su equivalente.

C.2.5. Operadores de sobrecarga de métodos

OMR Reemplazo de contenidos de método sobrecargado.

OMD Eliminación del método sobrecargado.

OAO Cambia el orden de los argumentos en un método.

OAN Cambia el número de argumentos en un método.

C.2.6. Operadores de características de lenguajes orientado a objetos

JTD Eliminación de la palabra clave `this`.

JSC Cambia la palabra clave `static` de las variables declaradas.

JID Eliminación de inicialización de variable miembro.

JDC Creación del constructor por defecto soportado por C#.

MNC Cambio del nombre de un método.

MBC Cambio de un miembro de una clase.

MCO Se llama a un miembro de una clase desde otro objeto.

MCI Se llama a un miembro de la clase desde una clase heredada.

C.2.7. Operadores de excepciones

EHR Elimina el manejador de excepciones.

EHC Cambia una sentencia del manejador de excepciones a una sentencia de propagación de excepción.

ENC Cambia un tipo de excepción utilizada en una sentencia `throw` o `catch`.

EXS Añade un `catch{}` vacío y general tras un bloque `try`.

C.2.8. Operadores de delegación de métodos

DMC Cambia un método delegado.

DMO Cambia el orden de un método delegado.

DEH Cambia un método delegado de un manejador de eventos.

C.2.9. Operadores de propiedad de reemplazo

IOK Sustituye la palabra `Override`.

OPD Elimina propiedades de `Overriding`.

OID Elimina la indexación de `Overriding`.

C.2.10. Otros operadores

RFI Inserta fallos de referencia de objeto.

PRM Reemplazo de propiedad con atributo miembro.

NDC Cambio en la declaración de un espacio de nombres (`namespace`).

C.3. Operadores de mutación para C++

Esta sección describe los operadores de mutación que se han definido en [55] para el lenguaje C++. Estos operadores se clasifican en tres categorías:

- Cambio de tipo de operadores
- Inserción de operadores
- Cambio de operadores aritméticos

C.3.1. Operadores de cambio de tipo de operadores

Los operadores de cambio de tipo de operadores se clasifican en:

- Variables
- Constantes
- Arrays
- Punteros

Operadores de variables

OVV Cambio de una variable por otra.

OVC Cambio de una variable por una constante.

OVA Cambio de una variable por una referencia a un vector.

OVP Cambio de una variable por una referencia a puntero.

Operadores de constantes

OCV Cambio de una constante por una variable.

OCC Cambio de una constante por otra constante.

OCA Cambio de una constante por una referencia a un vector.

OCP Cambio de una constante por una referencia a puntero.

Operadores de vectores

OAV Cambio de una referencia a un vector por una variable.

OAC Cambio de una referencia a un vector por una constante.

OAA Cambio de una referencia a un vector por otra.

OAP Cambio de una referencia a un vector por una referencia a puntero.

OAN Cambio del nombre de un vector por otro nombre.

Operadores de punteros

OPV Cambio de una referencia a puntero por una variable.

OPC Cambio de una referencia a puntero por una constante.

OPA Cambio de una referencia a puntero por una referencia a un vector.

OPP Cambio de una referencia a puntero por otra referencia a puntero.

OPN Cambio del nombre de un puntero por el nombre de otro puntero.

C.3.2. Operadores de inserción de operadores

Los operadores de insersión de operadores se clasifican en:

- Inserción de operadores binarios
- Inserción de operadores unarios

Operadores de inserción de operadores binarios

IBO1 Cambia cada operador binario (`*`, `/`, `%`) y de adición (`+`, `-`) por otro operador binario y de adición.

IBO2 Cambia cada operador relacional (`<`, `>`, `<=`, `>=`, `==`, `!=`) por otro operador del mismo tipo.

IBO3 Cambia cada operador lógico (`&&`, `||`) por otro del mismo tipo.

IBO4 Cambia cada operador de asignación (`=`, `+=`, `-=`, `*=`, `/=`) por otro del mismo tipo.

Operadores de inserción de operadores unarios

UOI Cambia cada operador unario (`!`, `&`, `~`, `*`, `+`, `++`, `-`, `-`) por otro operador del mismo tipo.

C.3.3. Operador de cambio de operadores aritméticos

AOR Cada uno de los operadores $+$, $-$, $\star$, y $/$ es reemplazado por otro operador.

C.4. Operadores de mutación para Java

Esta sección describe los operadores de mutación para el lenguaje Java definidos por diferentes autores.

Los operadores definidos en [32] se clasifican en:

- Ocultación de información (**AMC**).
- Herencia (**IHD**, **IHI**, **IOD**, **IOP**, **IOR**, **ISD**, **IPC**).
- Polimorfismo y construcción dinámica (**PNC**, **PMD**, **PPD**).
- Sobrecarga de métodos (**OMR**, **OMD**, **OAC**, **OAN**, **JSC**).

Los operadores de mutación para el comportamiento concurrente de Java (J2SE 5.0) definidos en [5] se clasifican en cinco categorías:

- Modifica parámetros de métodos concurrentes (**MXT**, **MSP**, **ESP**, **MSF**, **MXC**, **MBR**).
- Modifica las ocurrencias de los métodos concurrentes (**RTXC**, **RCXC**, **RNA**, **RJS**, **ELPA**).
- Modifica las palabras claves (**SAN**, **ASTK**, **RSTK**, **RSK**, **RSB**, **RVK**, **RFU**).
- Intercambio de objetos concurrentes (**RXO**, **EELO**).
- Modifica las regiones críticas (**SHCR**, **EXCR**, **SKCR**, **SPCR**).

Los operadores de mutación para Java definidos en [24] se clasifican en:

- Captura de excepciones (**CBR**, **CBI**, **CBD**, **PTL**, **CRE**).

Los operadores definidos en [50] se clasifican en:

- Método
  - Operaciones aritméticas (**AOIS**, **AORB**, **AOIU**).
  - Condicionales (**COD**, **COI**).
  - Lógicas (**LOI**).
- Clase
  - Específicos de Java (**JSI**, **JID**, **JSD**, **JDC**).
  - Polimorfismo (**PCI**, **PCD**, **PRV**).

Los operadores definidos en [34] se clasifican en:

- Tradicionales (**ABS**, **AOR**, **LCR**, **ROR**, **UOI**).
- Propios del lenguaje (**UOD**, **SOR**, **LOR**, **COR**, **ASR**).
- Orientados a objetos (**EOA**, **EOC**).
- Clases (**JTD**, **JTI**, **EAM**, **EMM**).

A continuación, establecemos una jerarquía común para los operadores de mutación para Java, a partir de las clasificaciones realizadas por los autores citados anteriormente:

- Método
- Clase:
  - Herencia
  - Polimorfismo
  - Características específicas de Java
- Concurrencia
  - Modifica parámetros de métodos concurrentes
  - Modifica las ocurrencias de los métodos concurrentes
  - Modifica las palabras claves
  - Intercambio de objetos concurrentes
  - Modifica las regiones críticas

C.4.1. Operadores a nivel de método

Los **operadores de mutación a nivel de método son**:

AOI Inserción de un operador aritmético.

AOD Eliminación de un operador aritmético.

AOIS Inserción de atajos de operadores aritméticos[1].

AORB Reemplazamiento de operadores aritméticos equivalentes[2].

AOIU Inserción de operadores aritméticos básicos[3].

ABS Inserción de un valor absoluto.

AOR Reemplazo de un operador aritmético.

ASR Reemplazo de un operador de asignación.

COD Eliminación de operador unario condicional.

[1]Este operador es una especialización del operador **AOI**
[2]Este operador es una especialización del operador **AOR**
[3]Este operador es una especialización del operador **AOI**

COI Inserción de operador unario condicional.

COR Reemplazo de un operador condicional.

CBR Reemplazo de un bloque `catch`.

CBI Inserción de un bloque `catch`.

CBD Eliminación de un bloque `catch`.

CRE Relanzamiento de la excepción tratada.

LOI Inserción de operador unario lógico.

LOR Reemplazo de un operador lógico.

LOD Eliminación de un operador lógico.

LCR Reemplazo de un conector lógico.

PTL Colocación de sentencias del bloque `try` después del mismo.

ROR Reemplazo de un operador relacional.

SOR Reemplazo de un operador de desplazamiento.

UOD Eliminación de un operador unario.

UOI Inserción de un operador unario.

C.4.2. Operadores de clase

Los **operadores relacionados con la herencia** son:

AMC Cambio del modificador de acceso.

IHD Eliminación de la variable *hiding*.

IHI Inserción de la variable *hiding*.

IOD Eliminación del método *overriding*.

IOP Cambio de la posición de llamada del método *overriding*.

IOR Renombrado del método *overriding*.

ISI Inserción de la palabra clave `super`.

ISD Eliminación de la palabra clave `super`.

IPC Llamada explícita a la eliminación del constructor padre.

Los **operadores relacionados con el polimorfismo** son:

PNC Llamada del método `new` con un tipo de la clase hijo.

PMD Declaración de variable miembro con el tipo de la clase padre.

PPD Declaración de un parámetro variable con el tipo de la clase hijo.

PCI Inserción del operador de tipo `cast`.

PCD Eliminación del operador de tipo `cast`.

PCC Cambio del tipo `cast`[4].

PRV Reemplazo de una referencia con su equivalente.

OMR Reemplazo de contenidos de método sobrecargado.

OMD Eliminación del método sobrecargado.

OAO Cambia el orden de los argumentos en un método.

OAN Cambia el número de argumentos en un método.

OAC Cambia los argumentos de llamada de método sobrecargado.

Los **operadores relacionados con las características específicas de Java** son:

EOA Reemplazamiento de una asignación referenciada por el contenido de la misma asignación.

EOC Reemplazamiento de una comparación referenciada por el contenido de la misma comparación.

EAM Cambio del método de acceso.

EMM Cambio del método modificador.

JTI Inserción de la palabra clave `this`.

JTD Eliminación de la palabra clave `this`.

JSI Inserción de modificador `static`.

JSD Eliminación de modificador `static`.

JSC Cambia la palabra clave `static` de las variables declaradas[5].

JID Eliminación de inicialización de variable miembro.

JDC Creación del constructor por defecto soportado por Java.

C.4.3. Operadores de concurrencia

Los **operadores que modifican parámetros de métodos concurrentes** son:

MXT Modifica el parámetro opcional de tiempo de las llamadas de métodos `wait()`, `sleep()`, `join()` y `await()`. La mutación consiste en incrementar o decrementar el tiempo en un factor de 2 (`t/2` o `t*2`).

MSP Modifica los parámetros de un bloque sincronizado.

ESP Intercambia los parámetros de un bloque sincronizado.

[4]Algunos autores en lugar de proponer un único operador para cambiar el tipo `cast`, proponen dos: **PCI** para insertarlo y **PCD** para eliminarlo.

[5]Algunos autores en lugar de proponer un único operador para cambiar la palabra clave `static`, proponen dos: **JSI** para insertarla y **JSD** para eliminarla.

MSF Modifica el parámetro opcional *fairness* de un semáforo. Si este parámetro es una constante este operador cambia el valor `true` a `false`, y viceversa. En el caso que sea booleano se niega.

MXC Modifica el contador de permisos en los semáforos y el contador de hilos en *latches* y *barriers*.

MBR Modifica el parámetro ejecutable *barrier*.

Los **operadores que modifican las ocurrencias de los métodos concurrentes** son:

RTXC Elimina los hilos de las llamada de métodos `wait()`, `join()`, `sleep()`, `yield()`, `notify()` y `notifyAll()`.

RCXC Elimina los mecanismos concurrentes de las llamadas de métodos (semáforos, *locks*, *barriers*...).

RNA Reemplaza `notifyAll()` por `notify()`.

RJS Reemplaza `join()` por `sleep()`.

ELPA Intercambia la adquisición de `lock/permit`.

Los **operadores que modifican las palabras clave** son:

SAN Cambio de llamada atómica a llamada no atómica.

ASTK Añade la palabra clave `static` a un método.

RSTK Elimina la palabra clave `static` de un método.

RSK Elimina la palabra clave `synchronized` de un método.

RSB Elimina un bloque sincronizado.

RVK Elimina la palabra clave `volatile`.

RFU Finalmente elimina tras desbloquear.

Los **operadores de intercambio de objetos concurrentes** son:

RXO Reemplaza una concurrencia de un mecanismo por otra (en *locks*, semáforos...).

EELO Intercambia objetos bloqueados explícitamente.

Los **operadores que modifican las regiones críticas** son:

SHCR Invierte las regiones críticas.

EXCR Expansión de las regiones críticas.

SKCR Contracción de las regiones críticas.

SPCR Separación de las regiones críticas.

C.5. Operadores de mutación para SQL

Esta sección describe los operadores de mutación para el lenguaje SQL definidos por diferentes autores.

Los operadores de mutación para SQL definidos en [11, 53] se clasifican en cuatro categorías:

- Cláusulas SQL
- Reemplazo de operador
- `NULL`
- Reemplazo de identificador

Los operadores definidos en [48, 49] se clasifican en dos categorías:

- Condiciones `WHERE`
- Llamadas a "funciones predefinidas" de la base de datos

Los operadores definidos en [6] se clasifican en una única categoría:

- Reemplazamiento

C.5.1. Operadores de cláusulas SQL

AGR Cada función de agregación (`MIN`, `MAX`, `AVG(DISTINCT)`, `SUM`, `SUM(DISTINCT)`, `COUNT`, `COUNT(DISTINCT)`) es reemplazada por otra.

GRU Eliminación de expresiones `GROUP BY`.

JOI Cada palabra clave (`INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`, `CROSS JOIN`) es reemplazada por otra.

ORD Intercambia las palabras clave `ASC` y `DESC`, o elimina expresiones `ORDER BY`.

SEL Intercambia las palabras clave `SELECT` y `SELECT DISTINCT`.

SUB Mutación de predicados de subconsultas.

UNI Intercambia las palabras clave `UNION` y `UNION ALL`, o elimina consultas de unión.

C.5.2. Operadores de reemplazo de operador

ABS Cada expresión aritmética o referencia a un número `e` es reemplazado por `ABS(e)` y `-ABS(e)`.

AOR Cada operador aritmético {`=`, `-`, `*`, `/`, `%`} es reemplazado por otro, o son aplicados `leftop` y `rightop`.

BTW Cada condición `a BETWEEN x AND y` es reemplazado por `a>x AND a<=y`, y por `a>=x AND a<y`.

LCR Cada operador lógico `AND`, `OR` es reemplazado por otro, por `falseop`, `trueop`, `leftop` o `rightop`.

LKE Mutaciones de condiciones de cadena `a LIKE s` (eliminando, reemplazando, añadiendo caracteres especiales { `%`, `_`},...)

ROR Cada operador relacional `=`, `<>`, `<`, `<=`, `>`, `>=` es reemplazado por otro, por `falseop` o `trueop`.

UOI Cada expresión aritmética o referencia a un número `e` es reemplazado por `e`, `e+1` y `e-1`.

C.5.3. Operadores de mutación NULL

NLF Intercambia los predicados `IS NULL` y `IS NOT NULL`.

NLI Establece el valor de la condición a `true` donde hay un valor `NULL` (cada atributo `a` en condición `C` es reemplazado por `C OR a IS NULL`).

NLO Cada atributo `a` en condición `C` es reemplazado por `NOT C OR a IS NULL`, `a IS NULL`, `a IS NOT NULL`.

NLS Cada referencia de columna `c` en la lista `SELECT` es reemplazado por una función que sustituye un valor `c` por `r` cuando un valor `NULL` se encuentra en `c`, la sintaxis depende del DBMS (DataBase Management System).

C.5.4. Operadores de reemplazo de identificador

IRC Cada referencia de columna es reemplazada por otra referencia de columna, constante o parámetro (de tipo compatible) en una cláusula `SELECT`.

IRP Cada parámetro es reemplazado por otra referencia de columna, constante o parámetro (de tipo compatible) en una cláusula `SELECT`.

IRT Cada constante es reemplazada por otra referencia de columna, constante o parámetro (de tipo compatible) en una cláusula `SELECT`.

IRH Cada referencia de atributo de columna es reemplazado por otro (de tipo compatible) definido en su tabla, pero no aplicado por operadores **IRC**, **IRT** o **IRP**.

C.5.5. Operadores de condiciones `WHERE`

Estos operadores introducen fallos en las condiciones `WHERE`.

RMWH Eliminación de condiciones `WHERE`.

NEGC Negación de cada una de las expresiones unitarias presentes en condiciones `WHERE`. Puede ser aplicado a consultas `SELECT`, `UPDATE` y `DELETE`.

FADP Añade `false and` después de `WHERE`. Puede ser aplicado a consultas `SELECT`, `UPDATE` y `DELETE`.

UNPR Añade un paréntesis izquierdo al comienzo de la condición `WHERE`, convirtiendo una consulta SQL a otra sintácticamente incorrecta.

C.5.6. Operadores de llamadas a "funciones predefinidas" de la base de datos

Estos operadores introducen fallos en las llamadas a "funciones predefinidas" de la base de datos.

MQFT Establece el flag de ejecución de consulta múltiple a valor `true`.

OVCR Modificación de los valores de flag de `commit` y `rollback`, o eliminación de sentencias que establecen los valores de flag de `commit` y `rollback`.

SMRZ Establece a cero (`0`) el valor máximo de número de registros que pueden ser devueltos por los conjuntos de resultados después de ejecutar una consulta `SELECT`.

SQDZ Establece a cero (`0`) el retardo de ejecución de consulta.

OVEP Modifica el valor, de `true` a `false` y viceversa, del flag responsable de la manipulación de caracteres de escape.

C.5.7. Operadores de reemplazo

PTCR Conmuta la participación de requisitos de tipos de entidades en la relación.

CDCR Reemplaza las cardinalidades de los tipos de entidad en la relación.

IWKR Reemplaza una expresión de tipos fuerte por una expresión de un tipo de entidad débil, o viceversa.

ATTR Reemplaza un atributo por otro atributo de tipo compatible.

GSCR Reemplaza una expresión de una superclase parcial por una expresión de una subclase y la forma negada de la superclase.

GSDR Reemplaza un tipo de entidad *sibling* por otra bajo la misma superclase.

UTCR Reemplaza un tipo de entidad por una subclase y/o superclase de la subclase, tal que estas superclases tienen la misma restricción de tipo unión.

C.6. Operadores de mutación para Ada

Esta sección describe los operadores de mutación que se han definido en [45] para el lenguaje Ada. Estos operadores se clasifican en cinco categorías:

- Reemplazo de operando
- Sentencia
- Expresión
- Cobertura
- Tarea

C.6.1. Operadores de reemplazo de operando

Los operadores de reemplazo de operando se clasifican en las siguientes categorías:

- Variables
- Constantes
- Referencias de vector
- Referencias de registro
- Referencias de puntero

Las reglas de tipado fuerte de Ada reducirán drásticamente el número de mutantes que son generados de este tipo:

- Muta inicializaciones (sólo **OCC**)
- Muta referencias de tipo enumerado
- No muta tipos
- No muta declaraciones
- No muta constantes `CASE`
- No muta parámetros de bucles en sentencias `FOR`.
- Las variables que son de un tipo que es declarado externamente y privadas son consideradas como escalares.

Operadores de variables

OVV Cambia una variable por otra.
OVC Cambia una variable por una constante.
OVA Cambia una variable por la referencia de un vector.
OVR Cambia una variable por la referencia de un registro.
OVP Cambia una variable por la referencia de un puntero.
OVI Eliminación de inicialización de variable.

Operadores de constantes

OVC Cambia una constante por una variable.
OCC Cambia una constante por otra constante.
OCA Cambia una constante por una referencia a un vector.
OCR Cambia una constante por la referencia de un registro.
OCP Cambia una constante por la referencia de un puntero.

Operadores de referencias de vector

OAV Cambia la referencia de un vector por una variable.
OAC Cambia la referencia de un vector por una constante.
OAA Cambia la referencia de un vector por otra referencia de un vector.
OAR Cambia la referencia de un vector por la referencia de un registro.
OAP Cambia la referencia de un vector por la referencia de un puntero.
OAN Cambia el nombre de un vector por otro nombre de vector.

Operadores de referencias de registro

ORV Cambia la referencia de un registro por una variable.
ORC Cambia la referencia de un registro por una constante.
ORA Cambia la referencia de un registro por la referencia de un vector.
ORR Cambia la referencia de un registro por la referencia de otro registro.
ORP Cambia la referencia de un registro por la referencia a un puntero.
ORF Cambia el `registro_eld` por otro `registro_eld`.
ORN Cambia el nombre de un registro por el nombre de otro registro.

Operadores de referencias de puntero

OPV Cambia la referencia de un puntero por una variable.

OPC Cambia la referencia de un puntero por una constante.

OPA Cambia la referencia de un puntero por la referencia de un vector.

OPR Cambia la referencia de un puntero por la referencia de un registro.

OPP Cambia la referencia de un puntero por la referencia de un puntero.

OPN Cambia el nombre de un puntero por el nombre de un puntero.

C.6.2. Operadores de sentencia

SEE Cambia el comienzo de la sentencia en cada bloque básico con `RAISE mut_trap;`. No realiza el cambio si la eliminación de la sentencia diera como resultado un error en tiempo de compilación, por ejemplo, si la sentencia es el único `RETURN` en una función. **SEE** se debería aplicar a sentencias y sentencias de bloque.

SRN Cambia cada sentencia con `NULL`.

SRR Cambia cada sentencia de una `FUNCTION` o `PROCEDURE` con `RETURN`.

SGL Cambia cada etiqueta `GOTO` con cualquier otra etiqueta legal o visible.

SRE Cambia sentencias en bucles con sentencias `EXIT`. Si hay una única sentencia en el bucle, este cambio sería equivalente a **SRN**, por lo que no se genera.

SWR Cambia la sentencia `WHILE` por la estructura `LOOP` y `EXIT`.

SRW Cambia la estuctura `LOOP` y `EXIT` por una sentencia `WHILE`.

SES Mueve cada sentencia `END` arriba o abajo una sentencia. No realiza el cambio si la eliminación de la sentencia diera como resultado un error en tiempo de compilación, por ejemplo, si la sentencia es el único `RETURN` en una función. **SEE** se debería aplicar a sentencias y sentencias de bloque.

SCA Cambia cada sentencia `CASE` con múltiples elecciones en alternativas donde cada alternativa contiene sólo una elección.

SER Cambia el nombre de la excepción por otro.

Existen algunos operadores de mutación de sentencias específicos de bucle:

SZI Salta el bucle.

SOI Provoca que el bucle solo itere una vez.

SIN Provoca que el bucle itere N veces.

SRI Provoca que el bucle itere en sentido inverso.

C.6.3. Operadores de expresión

EIA Inserta un operador unario `ABS` delante de cada expresión aritmética y subexpresión.

ENI Inserta un operador unario `-ABS` delante de cada expresión aritmética y subexpresión.

EEZ Inserta el subprograma `Except_on_Zero` delante de cada expresión aritmética y subexpresión.

EOR Cambia cada operador aritmético binario (`+`, `-`, `*`, `/`, `MOD`, `REM`, `**`) por otro operador aritmético binario sintácticamente legal. `MOD` y `REM` están sólo definidas para tipos enteros. `**` requiere el operando derecho para ser entero. `*` permite que el punto fijo y entero se mezclen. `/` permite punto fijo en la izquierda y entero en la derecha.

ERR Cambia cada operador relacional por otro operador relacional sintácticamente legal.

EMR Cambia cada `IN` por `NOT IN` y cada `NOT IN` por `IN`. Este operador es subsumado por el operador **CDE** y no debería ser utilizado si **CDE** está.

ELR Cambia cada operador lógico (`AND`, `OR`, `XOR`, `AND THEN`, `OR ELSE`) por otro operador lógico.

EUI Inserta el operador unario - delante de cada expresión aritmética y subexpresión. El operador unario + es la operación identidad.

EUR Cambia cada operador unario (`+`, `-`, `ABS`) por otro.

ESR Cambia cada nombre de función y subrutina con el nombre de otra que tiene la misma firma sintáctica y procede del mismo paquete.

EDT Cada pequeña expresión (operando: constante, variable, referencia de vector, referencia de registro, referencia de puntero) es modificada en una pequeña cantidad. No se deben modificar los subíndices de los vectores, puesto que la mayoría de los cambios causarían un fallo por salida de rango. No se debe mutar si el cambio produce un mutante que es equivalente a otro modificado levemente en la misma expresión. No se deben mutar parámetros de bucle si el cambio produce un rango `NULL` (esto sería equivalente a un mutante **OCC**).

EAR Cambia cada atributo por otro sintácticamente legal.

EEO Inserta delante de cada expresión aritmética el subprograma `Except_On_OverFlow`.

EEU Inserta delante de cada expresión aritmética el subprograma `Except_On_UnderFlow`.

C.6.4. Operadores de cobertura

CDE Cambia cada decisión `TRUE` por `FALSE` y viceversa.

CCO Cambia cada condición por `TRUE` y `FALSE`. Habría alguna redundancia. Esto sería reducido teniendo el sistema de mutación que suprimir algunos mutantes.

CDC Activa **CCO** y **CDE**.

CMC Se realizan todas las combinaciones de condiciones de forma separada. Si se utiliza **CMC**, **CDE**, **CCO**, y **CDC** serían redundantes y no deberían ser usados.

C.6.5. Operadores de tarea

TEM Cambia cada nombre de llamada `ENTRY` por otro nombre `ENTRY` que tenga la misma forma sintáctica y que proceda de la misma tarea.

TAR Cambia nombres de entrada por otras entradas visibles del mismo instante.

TSA Cada sentencia `SELECT alternative` con múltiples elecciones es separada en alternativas donde cada alternativa contiene sólo una opción.

C.7. Operadores de mutación para .NET

Esta sección describe los operadores de mutación que se han definido en [36] para evaluar la calidad de las pruebas realizadas a aplicaciones Web desarrolladas en el entorno .NET. Estos operadores se clasifican en tres categorías:

- Nivel de método y clase
- Nivel de presentación
- Nivel de evento

C.7.1. Operadores de nivel de método y clase

ORO Sustituye un operador por otro operador o constante.

EMO Modifica una expresión. Elimina o inserta operadores nuevos.

SMO Elimina parte o todo de una sentencia.

ICE Sustituye el nombre de la clase en la expresión de creación de instancia por nombres de clases compatibles.

POC Cambia el orden de los parámetros en declaraciones de métodos si el método tiene más de un parámetro.

VMR Elimina una declaración de método de métodos `overloading/overloaded`.

AOC Generan suficientes casos de prueba para probar la accesibilidad y visibilidad.

C.7.2. Operadores de nivel de presentación

ICC Comprueba si se está heredando en el fichero de presentación la clase `code-behind` correcta o no.

EIO Intercambia o rota elementos del mismo tipo.

OVO Cambia al valor opuesto. Controla que el texto visualizado corresponde a la salida resultante y que se emite el mensaje correcto para el evento correcto.

ERO Elimina un elemento. Comprueba si todos los elementos de una lista están incluidos, especialmente en grandes listas en las que es fácil omitir elementos.

C.7.3. Operadores de nivel de evento

VRO Elimina un evento. Comprueba si los hiperenlaces son activados correctamente.

VIO Elimina la implementación de un evento. Comprueba que el efecto del evento disparado ha sido implementado correctamente.

VSO Intercambia eventos. Comprueba que el efecto previsto del evento disparado ha sido implementado.

C.8. Operadores de mutación para Fortran

Esta sección describe los operadores de mutación que se han definido en [31] para el lenguaje Fortran. Estos operadores se utilizan en la herramienta de análisis de mutaciones construida por King y Offut, denominada *Mothra*.

ARR Reemplazamiento de una referencia vector por otra. Todos los tipos aritméticos están considerados compatibles y las cadenas de caracteres son no compatibles a no ser que tengan la misma longitud.

ABS Inserción de un valor absoluto. Cada expresión o subexpresión aritmética se le precede por un operador unario `ABS`, `NEGABS` y `ZPUSH`. `ABS` calcula el valor absoluto de la expresión, `NEGABS` calcula la negación del valor absoluto y `ZPUSH` se emplea siempre que la expresión sea cero.

ACR Reemplazo de una referencia vector por una constante. Cada constante se reemplaza por cada una de las referencias que sean compatibles. Todos los tipos aritméticos son considerados compatibles, las cadenas de caracteres no son compatibles, a no ser que tengan la misma longitud.

AOR Sustituye operadores +, −, ⋆, / y ⋆⋆ por otro. Además son sustituidos por `LEFTOP`, `RIGHTOP` y `MOD`.

ASR Cada variable escalar en un programa es reemplazada por cada referencia de vector de tipo compatible en dicho programa. Todos los tipos aritméticos se consideran compatibles exceptuando las cadenas de caracteres, incluso si tienen la misma longitud.

CAR Cada referencia de vector en un programa es reemplazada por cada constante visible, teniendo en cuenta el tipo base del vector y el tipo de la constante sin aritméticos.

CNR En cada referencia de vector, el nombre del vector es reemplazado por el nombre de otro vector de tipo compatible e idéntico número de dimensiones. Todos los tipos aritméticos se consideran compatibles exceptuando las cadenas de caracteres, incluso si tienen la misma longitud.

CRP Cada valor de la constante es modificado ligeramente para imitar la prueba en el dominio de la perturbación.

CSR Cada variable escalar en un programa es reemplazada por una constante visible en el mismo programa. Los tipos de ambas deben ser aritméticos.

DER La etiqueta en cada sentencia `DO` es reemplazada por cada etiqueta en el mismo programa. Además cada sentencia `DO` es reemplazada por una sentencia `ONETRIP`. Ésta es idéntica a `DO` excepto a que el cuerpo de un bucle se ejecuta sólo una vez.

DSA La mutación **DSA** es idéntica a la mutación **CRP**, aunque afecta a constantes en sentencias `DATA` más que a constantes en sentencias ejecutables.

GLR Las etiquetas de un `GOTO` incondicional, sentencias computerizadas `GOTO` e `IF` aritmético son reemplazadas por cada etiqueta del mismo programa.

LCR Cada ocurrencia de uno de los operadores lógicos (`AND`, `OR`, `EQV`, `NEQV`) es reemplazado por cada uno de los demás operadores. Además puede ser reemplazado por `FALSEOP`, `TRUEOP`, `LEFTOP` y `RIGHTOP`.

ROR Cada ocurrencia de uno de los operadores relacionales (`LT`, `LE`, `GT`, `GE`, `EQ`, `NE`) es reemplazado por otros operadores y por `FALSEOP` y `TRUEOP`.

RSR Cada sentencia en una subrutina o función externa (incluyendo la sentencia dentro de un `IF` lógico) es reemplazada por `RETURN`.

SAN Cada sentencia al comienzo de un bloque básico y una sentencia lógica `IF` es reemplazada por `TRAP`.

SAR Cada referencia de vector en un programa es reemplazada por cada variable escalar de tipo compatible que aparece en el programa. Todos los tipos aritméticos se consideran compatibles; las cadenas de caracteres son compatibles si son de la misma longitud.

SCR Cada constante en un programa es reemplazada por cada variable escalar de tipo compatible visible en el programa. Todos los tipos aritméticos se consideran compatibles; las cadenas de caracteres son compatibles si son de la misma longitud.

SDL Cada sentencia es reemplazada por `CONTINUE`.

SRC Cada constante aritmética en un programa es reemplazada por cada constante aritmética visible en ese programa.

SVR Cada variable escalar en un programa es reemplazada por cada variable escalar de tipo compatible visible en el programa. Todos los tipos aritméticos se consideran compatibles; las cadenas de caracteres son compatibles si son de la misma longitud.

UOI Cada expresion aritmética es negada, incrementada en 1 y decrementada en 1. Cada expresión lógica es complementada.

Bibliografía

[1] ActiveVOS. *ActiveBPEL WS-BPEL and BPEL4WS engine*. http://sourceforge.net/projects/activebpel, febrero 2008.

[2] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, y Eugene Spafford. *Design of Mutant Operators for the C Programming Language*. Informe Técnico SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[3] Roger T. Alexander, James M. Bieman, Sudipto Ghosh, y Bixia Ji. *Mutation of Java objects*. En *ISSRE 2002: 13th International Symposium on Software Reliability Engineering*, páginas 341–351. IEEE Computer Society, Annapolis, Maryland, USA, 2002.

[4] Juan Boubeta Puig, Inmaculada Medina Bulo, y Antonio García Domínguez. *Equivalencias entre los operadores de mutación definidos para WS-BPEL 2.0 y los definidos para otros lenguajes*. *PRIS 2010: V Taller sobre Pruebas en Ingeniería del Software*, pendiente de publicación, 2010.

[5] Jeremy S. Bradbury, James R. Cordy, y Juergen Dingel. *Mutation Operators for Concurrent Java (J2SE 5.0)*. En *Mutation 2006: Second Workshop on Mutation Analysis*, páginas 83–92. IEEE Computer Society, Raleigh, North Carolina, USA, 2006.

[6] W. K. Chan, S. C. Cheung, y T. H. Tse. *Fault-based testing of database application programs with conceptual data model*. En *QSIC 2005: Fifth International Conference on Quality Software*, páginas 187–196. 2005.

[7] Ben Collins-Sussman, Brian W. Fitzpatrick, y C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, primera edición, junio 2004. http://svnbook.red-bean.com/nightly/en/index.html.

[8] Márcio Eduardo Delamaro y José Carlos Maldonado. *Proteum–A Tool for the Assessment of Test Adequacy for C Programs*. En *PCS 1996: Proceedings of the Conference on Performability in Computing System*, páginas 79–95. julio 1996.

[9] Anna Derezińska. *Quality Assessment of Mutation Operators Dedicated for C# Programs*. En *QSIC 2006: Sixth International Conference on Quality Software*, páginas 227–234. IEEE Computer Society, Beijing, China, 2006.

[10] Anna Derezińska. *Advanced mutation operators applicable in C# programs*. En *Software Engineering Techniques: Design for Quality*, páginas 283–288. 2007.

[11] Anna Derezińska. *An experimental case study to applying mutation analysis for SQL queries*. En *IMCSIT 2009: International Multiconference on Computer Science and Information Technology*, páginas 559–566. 2009.

[12] Edsger W. Dijkstra. *Chapter I: Notes on structured programming*. páginas 1–82, 1972.

[13] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, y Inmaculada Medina-Bulo. *GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions*. En *ECOWS 2009: Seventh IEEE European Conference on Web Services*, páginas 97–106. IEEE Computer Society, Eindhoven, The Netherlands, 2009.

[14] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, y Inmaculada Medina-Bulo. *A Framework for Mutant Genetic Generation for WS-BPEL*. En *SOFSEM 2009: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science, Lecture Notes in Computer Science*, tomo 5404, páginas 229–240. Springer-Verlag, 2009. ISBN 978-3-540-95890-1.

[15] Juan José Domínguez Jiménez, Antonia Estero Botaro, y Inmaculada Medina Bulo. *Generación de mutantes con algoritmos genéticos*. En *Actas del III Taller sobre Pruebas en Ingeniería del Software*, tomo 2, páginas 27–32. 2008.

[16] Antonia Estero-Botaro, Francisco Palomo-Lozano, y 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.

[17] Antonia Estero-Botaro, Francisco Palomo-Lozano, y Inmaculada Medina-Bulo. *Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions*. *IEEE International Conference on Software Testing Verification and Validation Workshop*, 0:142–150, 2010.

[18] Eric T. Freeman, Elisabeth Robson, Bert Bates, y Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.

[19] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Patrones de Diseño*. Addison-Wesley, 2003.

[20] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Yves Lafon, Jean-Jacques Moreau, Anish Karmarkar, y Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C recommendation, W3C, abril 2007. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[21] Giuseppe Di Guglielmo y the contributing authors. *Mutation Testing Online*, febrero 2010. URL http://www.mutationtest.net.

[22] Tim Hallwyl. *Evaluating the BPEL Standard Specification*. Proyecto Fin de Carrera, Department of Computer Science, University of Copenhagen, 2008.

[23] IDC. *Research Reports*, 2008. URL http://www.idc.com.

[24] Changbing Ji, Zhenyu Chen, Baowen Xu, y Ziyuan Wang. *A New Mutation Analysis Method for Testing Java Exception Handling*. En *COMPSAC 2009: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, páginas 556–561. Seattle, Washington, USA, julio 2009.

[25] Yue Jia y Mark Harman. *An Analysis and Survey of the Development of Mutation Testing*. *IEEE Transactions of Software Engineering*, Por aparecer, 2010.

[26] JUnit.org. *Resources for Test Driven Development*, marzo 2010. URL http://www.junit.org/.

[27] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C recommendation, W3C, enero 2007. URL http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[28] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*. Wrox Press Ltd., Birmingham, UK, UK, 2008. ISBN 0470192747, 9780470192740.

[29] Michael Kay, Don Chamberlin, Jonathan Robie, Mary F. Fernández, Jérôme Siméon, Scott Boag, y Anders Berglund. *XML Path Language (XPath) 2.0*. W3C recommendation, W3C, enero 2007. URL http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[30] Sunwoo Kim, John A. Clark, y John A. McDermid. *The Rigorous Generation of Java Mutation Operators Using HAZOP*. Informe técnico, The University of York, 1999.

[31] K. N. King y A. Jefferson Offutt. *A FORTRAN Language System for Mutation-based Software Testing*. *Software – Practice and Experience*, 21(7):685–718, 1991.

[32] Yu-Seung Ma, Yong-Rae Kwon, y Jeff Offutt. *Inter-Class Mutation Operators for Java*. En *ISSRE 2002: Proceedings of the 13th International Symposium on Software Reliability Engineering*, página 352. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-8186-1763-3.

[33] Yu-Seung Ma, Jeff Offutt, y Yong-Rae Kwon. *MuJava: An Automated Class Mutation System*. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[34] Lech Madeyski y Norbert Radyk. *Judy - A mutation testing tool for Java*. *IET Software*, 4(1):32–42, 2010. URL http://dx.doi.org/10.1049/iet-sen.2008.0038.

[35] Ashok Malhotra y Paul V. Biron. *XML Schema Part 2: Datatypes Second Edition*. W3C recommendation, W3C, octubre 2004. URL http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[36] Nashat Mansour y Manal Houri. *Testing web applications*. *Information and Software Technology*, 48(1):31–42, 2006. URL http://dx.doi.org/10.1016/j.infsof.2005.02.007.

[37] Mike Mason. *Pragmatic Version Control Using Subversion*. The Pragmatic Programmers, segunda edición, junio 2008.

[38] Philip Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*. Proyecto Fin de Carrera, Leibniz Universität Hannover, Fakultät für Electrotechnik und Informatik, septiembre 2006.

[39] Philip Mayer y Daniel Lübke. *Towards a BPEL unit testing framework*. En *TAV-WEB 2006: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, páginas 33–42. ACM, 2006. ISBN 1-59593-458-8. URL http://doi.acm.org/10.1145/1145718.1145723.

[40] Sun Microsystems. *NetBeans*, diciembre 2009. http://www.netbeans.org/.

[41] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. ISBN 0471043281.

[42] OASIS. *ebXML*, 2006. URL http://www.ebxml.org/.

[43] OASIS. *Web Services Business Process Execution Language 2.0*. Organization for the Advancement of Structured Information Standards, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[44] OASIS. *Universal Description, Discovery, and Integration OASIS Standard*, 2010. URL http://uddi.xml.org/.

[45] A. Jefferson Offutt, Jeff Voas, y Jeff Payne. *Mutation Operators for Ada*. Informe Técnico ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.

[46] Manuel Palomo-Duarte, Antonio García-Domínguez, y Inmaculada Medina-Bulo. *Takuan: A Dynamic Invariant Generation System for WS-BPEL Compositions*. En *ECOWS 2008: Proceedings of the 2008 Sixth European Conference on Web Services*, páginas 63–72. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3399-5. URL http://dx.doi.org/10.1109/ECOWS.2008.17.

[47] Roger S. Pressman. *Ingeniería del Software: Un Enfoque Práctico*. McGraw-Hill, sexta edición, 2005.

[48] Hossain Shahriar. *Mutation-based testing of buffer overflows, SQL injections, and format string bugs*. Proyecto Fin de Carrera, University of Queen, agosto 2008. URL http://hdl.handle.net/1974/1359.

[49] Hossain Shahriar y Mohammad Zulkernine. *MUSIC: Mutation-based SQL Injection Vulnerability Checking*. En *QSIC 2008: The Eighth International Conference on Quality Software*, páginas 77–86. 2008.

[50] Ben H. Smith y Laurie Williams. *Should software testers use mutation analysis to augment a test set? Systems and Software*, 82(11):1819–1832, 2009. ISSN 0164-1212. URL http://dx.doi.org/10.1016/j.jss.2009.06.031.

[51] Fachgebiet Software Engineering. *BPELUnit - The Open Source Unit Testing Framework for BPEL*, marzo 2010. URL http://www.se.uni-hannover.de/forschung/soa/bpelunit/.

[52] Ian Sommerville. *Ingeniería del Software: Un Enfoque Práctico*. Pearson, séptima edición, 2005.

[53] Javier Tuya, María J. Suárez-Cabal, y Claudio de la Riva. *Mutating database queries*. *Information and Software Technology*, 49(4):398–417, 2007.

[54] Asir S. Vedamuthu. *Web Services Description Language (WSDL) Version 2.0 SOAP 1.1 Binding*. W3C note, W3C, junio 2007. URL http://www.w3.org/TR/2007/NOTE-wsdl20-soap11-binding-20070626.

[55] Hong Zhang. *Operadores de mutación para C++*, febrero 2010. URL http://people.cis.ksu.edu/~hzh8888/mse_project.htm.