UNIVERSIDAD COMPLUTENSE DE MADRID

UCM

FACULTAD DE INFORMÁTICA,

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA.

UNA APROXIMACIÓN MDA PARA LA CONVERSIÓN ENTRE SERVICIOS WEB SOAP Y RESTFUL.

Trabajo Fin Máster en Sistemas Inteligentes

ANAYANSI DA SILVA DE LA CRUZ

Director: Antonio Navarro Martín

Madrid, España
2012 - 2013

Convocatoria: Septiembre 2013
Calificación: Sobresaliente (10)

Autorización de Difusión

ANAYANSI DA SILVA DE LA CRUZ

Septiembre/2013

La abajo firmante, matriculada en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Una Aproximación MDA Para La Conversión Entre Servicios Web SOAP y REST", realizado durante el curso académico 2012-2013 bajo la dirección de Dr. Antonio Navarro Martín en el Departamento de Sistemas Inteligentes, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Hoy en día hay diversas aproximaciones que permiten la conversión entre los servicios web SOAP y RESTful. Sin embargo, ninguno de estas define un modelado de alto nivel del proceso de transformación, lo que dificulta la plena comprensión del mecanismo de conversión. Con el fin de aclarar este proceso, proporcionamos un enfoque basado en MDA que formaliza la conversión entre los servicios web SOAP y RESTful. Este enfoque puede utilizarse para guiar el proceso de traducción en los próximos marcos de conversión y para la publicación de los servicios web en entornos de desarrollo.

Para caracterizar el mecanismo de conversión se definen tres metamodelos MOF: SOAP, RESTful y un metamodelo intermedios SOA. Este metamodelo intermedio se utiliza como un puente o mediador en la conversión entre los otros dos metamodelos. También se definen reglas de transformación en QVT Relations entre los metamodelos SOAP, RESTful y el metamodelo SOA para lograr una caracterización formal del proceso de transformación.

Palabras Claves: MOF, e-core, QVT, SOA, WSDL, WADL, REST, MDA, MDD.

Abstract

Today there are several approaches that allow the conversion between SOAP and RESTful web services. However, none of these approaches defines a high-level modeling of the transformation process, hindering the full understanding of the conversion mechanism. To clarify this process, we provide a MDA-based approach to formalize the conversion between SOAP and RESTful web services. This approach can be used to guide the translation process in the forthcoming frameworks, as well as the conversion and publishing services in IDEs.

To characterize the conversion mechanism three MOF metamodels are defined: SOAP, RESTful and SOA intermediate metamodel. This intermediate metamodel is used as a bridge or mediator in the conversion between the other two metamodels. We also define transformation rules in QVT Relations between the SOAP, RESTful and SOA metamodels to provide a formal characterization of the transformation process.

Keywords: MOF, e-core, QVT, SOA, WSDL, WADL, REST, MDA, MDD.

Índice General

1. Introducción 10
1.1. Planteamiento y justificación del problema 10
1.2. Hipótesis y objetivo 12
1.3. Organización de la memoria 13
2. Estado del arte 15
2.1. Servicios web SOAP y RESTful. Comparación 15
2.1.1. Servicios SOAP 15
2.1.2. Transferencia de estado representacional (REST) 19
2.1.3. Comparación entre REST y SOAP 21
2.2. Transformaciones entre servicios RESTful y SOAP. Herramientas 22
2.2.1. Puertas de enlace de servicios (Service Gateways) 22
2.2.1.1. Policy Studio y Vordel XML Gateway 23
2.2.1.2. SOA Gateway 25
2.2.2. Enterprise Servicie Bus (ESB) 25
2.2.2.1. Oracle Service Bus (OSB) 27
2.2.2.2. ESB de IBM WebSphere 30
2.2.2.3. ESB de Apache 35
2.3. Arquitectura dirigida por modelos (MDA) 39
2.4. Arquitectura de metamodelado de cuatro capas 43
2.5. Estándar QVT 45
2.5.1. QVT Relations 46
3. Metamodelo REST 48
3.1. Metamodelo REST propuesto 54
3.1.1. RESTful. RESTService 55
3.1.2. RESTful. Type 57
3.1.3. RESTful. Representation 58
3.1.4. RESTful. Parameter 60
3.1.5. RESTful. Method 61
3.2. Ejemplo de Instanciación del Metamodelo RESTful 64
3.2.1. Instancia del metamodelo RESTful 65
4. WSDL: Web Services Description Language 66
4.1. Metamodelo SOAP propuesto 68
4.1.1. WSDL. WSDL Definition 69

4.1.2. WSDL. Type ... 70
4.1.3. WSDL. Message ... 73
4.1.4. WSDL. Port Type ... 74
4.1.5. WSDL. Binding ... 76
4.1.5.1. Binding. Style/User. ... 78
4.1.6. WSDL. Service ... 83
4.2. Instanciación del metamodelo SOAP ... 84
4.2.1. Instancia del metamodelo SOAP. ... 86
5. Metamodelo Intermedio SOA. Reglas de Conversión ... 87
5.1. Metamodelo intermedio SOA propuesto ... 88
5.1.1. SOA. SOAPackage ... 89
5.1.2. SOA. NamedElement ... 90
5.1.3. SOA. Type ... 91
5.1.4. SOA. Interface ... 92
5.2. Instanciación del metamodelo SOA ... 94
5.3. Reglas de Conversión ... 96
5.3.1. Conversión RESTful to SOA ... 97
5.3.2. Conversión SOAP to SOA. ... 98
6. Conclusiones y trabajo futuro. ... 101
Bibliografía. ... 103
Apéndice A. Reglas de transformaciones QVT. ... 112
A.1. RESTful to SOA. ... 112
A.2. SOA to RESTful. ... 116
A.3. SOAP to SOA. ... 121
A.4. SOA to SOAP. ... 126

Índice de Figuras

Figura 1.1: Mecanismo de conversión previsto en nuestro enfoque 13

Figura 2.1: Intercambio de mensajes SOAP (Hansen, 2007)...................................... 16

Figura 2.2: Ejemplo de mensaje SOAP para una reserva de viaje (W3C, 2003).......... 17

Figura 2.3: Ejemplo de mensaje SOAP enviado como respuesta del ejemplo de la figura 2.2 (W3C, 2003)... 18

Figura 2.4: Respuesta del mensaje SOAP al ejemplo de la figura 3, dando continuidad al intercambio de mensajes (W3C, 2003)... 19

Figura 2.5: Arquitectura de la conversión Vordel (O'Neill, 2008) 24

Figura 2.6: Creación de la política de conversión Vordel REST a SOAP (O'Neill, 2008) 24

Figura 2.7: Tipología de los ESB (Abbas, 2009) 26

Figura 2.8: Capas de unión y transporte de los OSB (ORACLE, 2011)....................... 28

Figura 2.9: Flujo de mensaje de los OSB (ORACLE, 2011) 30

Figura 2.10: Conversión de REST a SOAP a través de IBM WebSphere Enterprise Service Bus (Abbas, 2009) 31

Figura 2.11: Conversión de SOAP a REST a través de IBM WebSphere Enterprise Service Bus (Abbas, 2009) 32

Figura 2.12: Conversión de REST a SOAP a través de IBM WebSphere Message Broker (Abbas, 2009) 32

Figura 2.13: Conversión de SOAP a REST a través de IBM WebSphere Message Broker (Abbas, 2009) 33

Figura 2.14: Topología de un cliente RESTful que transmite JSON a WebSphere DataPower (Muschett, 2011)... 34

Figura 2.15: Conversión de REST a SOAP a través de IBM WebSphere *DataPower (Abbas, 2009)*... 34

Figura 2.16: Conversión de SOAP a REST a través de *IBM WebSphere DataPower* (Abbas, 2009) 35

Figura 2.17: Arquitectura del JBI (Caponi, 2008).. 36

Figura 2.18: Interacción entre componentes JBI (Caponi, 2008)................................ 36

Figura 2.19: Interacción entre componentes SE (servicios externos) y BC(controladores multiprotocolo) a través de NMR (Caponi, 2008).. 38

Figura 2.20: Ciclo de vida de desarrollo del software tradicional (Kleppe, 2003)
Figura 2.21: Ciclo de vida de desarrollo del software en MDA (Kleppe, 2003) .. 40
Figura 2.22: Interoperabilidad de MDA mediante puentes (Kleppe, 2003).................. 42
Figura 2.23: Jerarquía de metamodelos en la arquitectura de cuatro capas (OMG, 2003) 44
Figura 2.24: Dependencia de paquetes en la especificación QVT (OMG, 2011)......... 46
Figura 3.1: Descriptor WALD 51
Figura 3.2: Esqueleto de un documento WDSL 2.0 (Mandel, 2008)............................ 52
Figura 3.4: Diagrama del elemento *RESTService* 56
Figura 3.5: Diagrama del elemento *Type* 58
Figura 3.7: Diagrama del elemento *Parameter*.. 61
Figura 3.8: Diagrama del elemento *Method*... 63
Figura 3.9: Instancia del metamodelo RESTful... 65
Figura 4.2: Diagrama del elemento raíz de la definición WSDL (corregir XMLSchema) 70
Figura 4.3: Diagrama del elemento "*Type"* de la definición WSDL (XMLSchema)... 72
Figura 4.4: Diagrama del elemento "*Message"* de la definición WSDL...................... 74
Figura 4.5: Diagrama del elemento "*PortType"* de la definición WSDL..................... 76
Figura 4.6: Diagrama del elemento *"Binding"* de la definición WSDL (SOAPBinding, SOAPOperation)... 81
Figura 4.8: Instancia del metamodelo SOAP.. 86
Figura 5.1: Metamodelo SOA 88
Figura 5.2: Diagrama del elemento *SOAPackage* 89
Figura 5.3: Diagrama del elemento *NamedElement* 90
Figura 5.4: Diagrama del elemento *Type*. 91
Figura 5.5: Diagrama del elemento *Interface*.. 93
Figura 5.6: Representación en UML de la interfaz para la gestión de productos 94
Figura 5.7: Instancia del metamodelo SOA 96

Índice de Tablas

Tabla 2.1: Características de REST y SOAP (Navarro Marset, 2006-07) 21

Tabla 2.2: Componentes JBI incluidos en *ServixMix* (Caponi, 2008) 38

Tabla 3.1: Diferencias notables entre WSDL 1.1 y WSDL 2.0 (Mandel, 2008)........... 52

Tabla 4.1: Patrón de intercambio de mensajes (MEP) 75

Tabla 5.1: Transformaciones *RESTful to SOA* 97

Tabla 5.2: Transformaciones *SOA to RESTful* 98

Tabla 5.3: Transformaciones *SOAP to SOA* 99

Tabla 5.4: Transformaciones *SOA to SOAP* 100

1. Introducción

El presente trabajo de fin de máster aborda la problemática asociada a alcanzar un mecanismo de conversión entre servicios web *RESTful* y *SOAP*. En este capítulo se presenta, a modo de introducción el trabajo que se ha llevado a cabo: el planteamiento y la justificación del trabajo; hipótesis y objetivos a tener en cuenta; y por último, la organización de esta memoria.

Así, en la sección 1.1, se plantea el problema abordado y se justifica la necesidad de un marco de conversión para el desarrollo orientado a servicios de las diferentes implementaciones web, describiendo además las ventajas que puede brindar la defiinición de este marco a través de una arquitectura dirigida por modelos. La sección 1.2 contiene la hipótesis planteada al inicio de este trabajo, así como el objetivo principal y derivados del mismo; y para finalizar, la sección 1.3 describe la organización de los restantes capítulos de este trabajo.

1.1. Planteamiento y justificación del problema

La Arquitectura Orientada a Servicios (SOA) es una arquitectura de éxito hoy en día. Esta arquitectura centra su atención en los servicios proporcionados por las empresas que suministran sistemas (Erl, 2005). Dentro del mundo de la computación orientada a servicio, la arquitectura SOA puede tener diferentes implementaciones, teniendo en cuenta dos factores: cómo se efectúa el servicio y la forma en que estos son proveídos a los clientes. CORBA (Bolton, 2001) puede ser una implementación de SOA, sin embargo, hoy en día, los servicios web son la aplicación más común (Erl, 2005).

En la actualidad hay dos enfoques principales en la implementación de servicios web: los servicios web SOAP (Erl, 2008) y los servicios web RESTful (T. Erl, 2012)

En ambos casos, una función de software (un servicio) se invoca a través de la *World Wide Web*, pero existe una diferencia importante: en el caso de los servicios web RESTful el cliente utiliza un URL para invocar un proceso remoto. Por lo tanto, el cliente es responsable de la codificación de la información transmitida y recibida (por ejemplo, el uso de XML (W3C, 2008) o JSON (W3Schools, 2013), así como para abrir y cerrar la conexión. En el caso de los servicios web SOAP una capa de software *middleware* aísla clientes de la codificación/decodificación de datos y de la gestión de la

conexión, haciendo una invocación a un objeto remoto que puede ser definida en el mismo lenguaje de programación del cliente.

En las dos implementaciones el cliente no tiene conocimiento del lenguaje de implementación del servicio, que puede ser el mismo o diferente del cliente. No obstante, en el caso de los servicios RESTful, el cliente realiza un esfuerzo importante para invocar el servicio, mientras que la invocación de servicios en SOAP es mucho más simple. Sin embargo, la invocación de servicios RESTful consume menos recursos y tiempo para el cliente y proveedor de servicios, que los servicios web SOAP; donde la capa de *middleware* tiene que ser implementada en ambos lados (cliente y proveedores de servicios).

Debido a que SOA está destinado a la integración de sistemas heterogéneos no es inusual que ambos tipos servicios web (RESTful y SOAP) tenga que coexistir en la misma organización. Por tanto, no es raro encontrar enfoques que permitar unificar ambas implementaciones de servicios. Así, en la actualidad, existen marcos que hacen posibles la conversión entre servicios web SOAP y RESTful.

El principal problema en el uso de estos marcos, es que no proporcionan un modelo de alto nivel que describa los mecanismos utilizados en la conversión, lo que dificulta la comprensión del proceso y por lo tanto el mantenimiento del servicio web convertido.

Por otro lado MDA (*Model Driven Architecture*) (OMG, 2003) constituye una importante herramienta para la alineación entre los procesos de negocio de alto nivel y las tecnologías de la información (Harmon, 2004). Esto se debe a que MDA proporciona una estructura conceptual que se extiende desde los modelos utilizados por los analistas de negocio, hasta diversos modelos utilizados por los desarrolladores de software. Además MDA ofrece, a través de la transformación de modelos, la posibilidad de que los elementos especificados en un diagrama puedan transformarse, en elementos de otros diagramas más detallados, siendo estos elementos derivados o diseñados a partir de la definición del elemento del diagrama origen.

Por tanto, si tenemos en cuenta los principios de SOA, estudiamos a profundidad las diferencias implementaciones de los servicios web, y utilizamos MDA como herramienta, podemos plantearnos los objetivos necesarios para dar solución a la problemática expuesta.

1.2. Hipótesis y objetivo

Desde el comienzo de este trabajo y teniendo en cuenta lo descrito en la sección anterior, se planteó la siguiente hipótesis:

- Si logramos especificar las implementaciones de los servicios web RESTful y SOAP utilizando MDA; entonces podemos definir un mecanismo de conversión de alto nivel que permita la transformación entre los modelos resultantes.

Por tanto, el objetivo principal de este trabajo derivado directamente de la hipótesis sería:

- Definir un mecanismo de conversión basado en MDA que permita la transformación entre servicios web RESTful y SOAP.

Este mecanismo de alto nivel se podría utilizar para la construcción de marcos de conversión de servicios web o para la publicación de servicios web en entornos de desarrollo integrados.

Para lograr el objetivo propuesto se han planteado los siguientes objetivos parciales:

1. Construir un metamodelo RESTful.
2. Construir un metamodelo SOAP utilizar WSDL como lenguaje descriptor de los servicios SOAP.
3. Definir un mecanismo de conversión entre los metamodelos RESTful y SOAP.
   - 3.1. Establecer semejanzas y diferencias entre los metamodelos involucrados en la conversión.
   - 3.2. Construir un metamodelo intermedio SOA que permita la conversión
   - 3.3. Definir reglas de conversión entre los metamodelos RESTful, SOAP y el metamodelo intermedio SOA.
4. Validar el mecanismo de conversión.
   - 4.1. Implementar reglas en QVT Relation que permite validar el mecanismo de conversión, así como la integridad de los metamodelos propuestos.

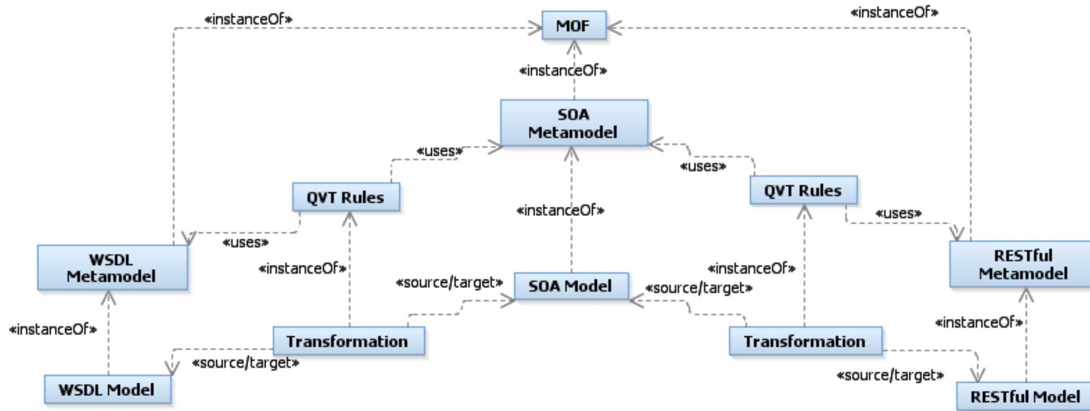

Figura 1.1: Mecanismo de conversión previsto en nuestro enfoque

1.3. Organización de la memoria

La organización de los restantes capítulos de este trabajo es la siguiente:

- Capítulo segundo: en este capítulo se analiza las diferentes implementaciones de los servicios web efectuando una comparación entre ellos, también se analizan herramientas que, debido a sus funcionalidades, permiten el filtrado de información que es utilizado para realizar la conversión entre ambas implementaciones de servicios web. Además en este capítulo se resume la propuesta de OMG (MDA) y la arquitectura de metamodelo de cuatro capas.
- Capítulo tercero: propone un metamodelo RESTful basado en el análisis de los requisitos contemplados en la definición de este estilo, así como restricciones actuales resultante de su aplicación en la ingeniería web. También presentamos una instanciación con un caso real del metamodelo propuesto.
- Capítulo cuarto: presenta un metamodelo SOAP fundamentado en el análisis de WSDL como lenguaje descriptor de los servicios SOAP, así como la instanciación de este metamodelo con un caso real.
- Capítulo quinto: en este capítulo se relata la conversión entre los metamodelos propuesto en los capítulos segundo y tercero, definiendo un metamodelo intermedio de conversión y su instanciación con un caso real. Además se especifica un mecanismo de conversión.
- Capítulo seis: presenta las conclusiones del trabajo y futuros trabajos a ser realizados.

- Bibliografía: este apartado incluye toda la bibliografía consultada para la realización de este trabajo.
- Apéndices: finalmente, se incluyen los apéndices, relatando las reglas de conversión definidas en QVT Relations del mecanismo de conversión propuesto en el quinto capítulo.

2. Estado del arte

2.1. Servicios web SOAP y RESTful. Comparación

A lo largo de los años, se ha popularizado un nuevo paradigma en el diseño de aplicaciones informáticas para la web: los llamados servicios web (*web services*):

Un servicio web se suele definir como una unidad de aplicación capaz de ofrecer datos o servicios de procesamiento a otras aplicaciones informáticas (Eiton Brun, 2002).

Los servicios web pueden implementarse en distintos estilos o arquitecturar, tales como, SOAP (*Simple Object Access Protocol*) y RESTful (servicios web del estilo REST). Estos señalan cómo se deben cursar las peticiones de servicio a los servidores, la forma en la cual deben enviar los resultados, y cómo se deben publicar o dar a conocer los servicios que están accesibles a través de un servidor web.

En los servicios web SOAP, la unidad básica de comunicación es el mensaje más que la operación, debido a esto, esta implementación es típicamente referenciada como servicios orientados a mensajes.

REST (*Representation State Transfer*). Los Servicios Web basados en REST intentan emular al protocolo HTTP o protocolos similares mediante el establecimiento de restricciones arquitectónicas a un conjunto de operaciones estándar (por ejemplo GET, PUT,…). Por tanto, este estilo se centra más en interactuar con recursos y estados, que con mensajes y operaciones.
Es importante agregar que, en general, el lenguaje XML constituye la base de estos estilos arquitectónico.

2.1.1. Servicios SOAP

Los servicios web SOAP se rigen por un conjunto de estándares de comunicación, basados en XML, SOAP (*Simple Object Access Protocol*) para el intercambio de datos, y el lenguaje WSDL (*Web Services Description Language*) para describir las funcionalidades de un servicio Web.

La especificación **SOAP** define dos modelos de mensajes (San Cristobal Ruiz, 2010):

- Un mensaje que se enviará desde la aplicación cliente a la aplicación servidor, solicitando la ejecución de un método al que se pasan una serie de parámetros.

- Un mensaje que se enviará desde la aplicación servidor al cliente, y que contendrá datos XML con los resultados de la ejecución del método solicitado.

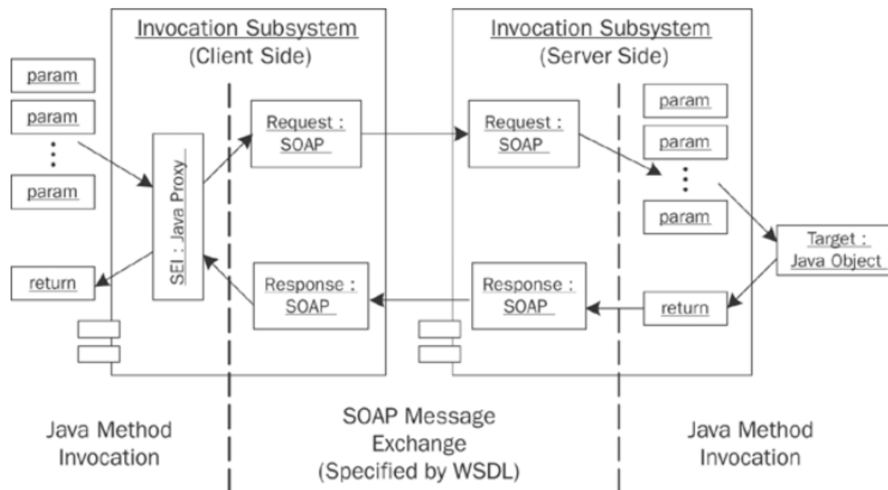

Figura 2.1: Intercambio de mensajes SOAP (Hansen, 2007)

El subsistema de invocación cliente se traduce en una llamada al método del proxy SEI *(Service Endpoint Interface)* dentro de la solicitud/respuesta SOAP y viceversa. La invocación del subsistema por lado del servidor de la solicitud/respuesta SOAP se traduce a una llamada al método Java (Java Object) (Hansen, 2007).

A modo de ejemplo, en la figura 2.2, el encabezado contiene dos bloques, cada uno de ellos se define en su propio espacio de nombres XML y representa algún aspecto perteneciente al proceso global del cuerpo del mensaje SOAP. Los bloques de encabezado *"reservation"* y *"passenger"* deben ser procesados por el siguiente intermediario SOAP que se encuentre en el camino del mensaje o, si no existe tal intermediario, por el destinatario último del mensaje. El elemento *env:Body* y sus elementos hijos asociados, *"itinerary"* y alojamiento, están destinados al intercambio de información entre el remitente SOAP inicial y el nodo SOAP que asume el papel del destinatario SOAP final en el camino del mensaje, que es la aplicación del servicio de viajes. Por tanto, el *env:Body* y sus contenidos son dirigidos implícitamente y esperan

ser entendidos por el destinatario último. Los medios por los que un nodo SOAP asume tal papel no están definidos por la especificación SOAP, y es determinado como parte de la semántica y el flujo de mensajes de la aplicación global.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
    xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid-morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
  <q:lodging
   xmlns:q="http://travelcompany.example.org/reservation/hotels">
   <q:preference>none</q:preference>
  </q:lodging>
 </env:Body>
</env:Envelope>
```

Figura 2.2: Ejemplo de mensaje SOAP para una reserva de viaje (W3C, 2003)

La figura 1.3 muestra un mensaje SOAP devuelto por la aplicación del servicio de viajes en respuesta al mensaje de petición de reserva (Figura 2.2). Esta respuesta busca la redefinición por parte de la información de la petición, concretamente la elección de aeropuertos en la ciudad de origen. El cuerpo del mensaje *(env:Body)* contiene el contenido principal del mensaje, que en este ejemplo incluye una lista de varias alternativas para la elección del aeropuerto, conforme a una definición de esquema XML en el espacio de nombres *http://travelcompany.example.org/reservation/travel*. En este ejemplo, los bloques de encabezado de la petición son devueltos (con algunos valores de subelementos alterados) en la respuesta.

Podemos ver como los intercambios de mensajes pueden acabar convirtiéndose en un patrón de intercambio de mensajes de ida y vuelta múltiple "conversacional". En la

figura 2.4 se muestra un mensaje SOAP enviado por la aplicación de reservas de viajes en respuesta al mensaje de la figura 2.2 eligiendo uno de la lista de aeropuertos disponibles. El bloque de encabezado "reserva" se acompaña del mismo valor del subelemento "referencia" en esta conversación, ofreciendo por tanto, un camino, por si fuese necesario, para la correlacionar los mensajes intercambiados entre ellos a nivel de aplicación.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:35:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itineraryClarification
    xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>
       <p:airportChoices>
          JFK LGA EWR
       </p:airportChoices>
     </p:departing>
   </p:departure>
   <p:return>
     <p:arriving>
       <p:airportChoices>
         JFK LGA EWR
       </p:airportChoices>
     </p:arriving>
   </p:return>
  </p:itineraryClarification>
 </env:Body>
</env:Envelope>
```

Figura 2.3: Ejemplo de mensaje SOAP enviado como respuesta del ejemplo de la figura 2.2 (W3C, 2003)

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation
     xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
    <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
    <m:dateAndTime>2001-11-29T13:36:50.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
   xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>LGA</p:departing>
   </p:departure>
   <p:return>
     <p:arriving>EWR</p:arriving>
   </p:return>
  </p:itinerary>
 </env:Body>
</env:Envelope>
```

Figura 2.4: Respuesta del mensaje SOAP al ejemplo de la figura 3, dando continuidad al intercambio de mensajes (W3C, 2003).

2.1.2. Transferencia de estado representacional (REST)

Tal y como define Roy Fielding: *"La transferencia de estado representacional (REST), es un estilo de arquitectura para sistemas hipermedia distribuidos. REST proporciona un conjunto de restricciones arquitectónicas que, cuando se aplica como un todo, hace hincapié en la escalabilidad de las interacciones de los componentes, la generalidad de interfaces, implementación independiente de los componentes y componentes de mediación, con el fin de reducir la latencia de la interacción, reforzar la seguridad, y encapsular los sistemas heredados"* (Roy Thomas Fielding, 2000).

REST es una estilo de arquitectura que accede a los recursos de un manara sencilla y sin tener en cuenta los estado, esto quiere decir, que no se almacena (ni en el cliente, ni el servidor) el estado de la comunicación.

Este estilo arquitectónico intenta simular HTTP, utiliza los cuatro métodos fundamentales de este protocolo, GET, PUT, POST y DELETE. Estos métodos permiten realizar las operaciones de recuperar, crear, actualizar y borrar recursos en el servidor. Estos recursos son ofrecidos en Internet y se identifican mediante un identificador único de recurso (URI) o una URL, también existe un estándar publicado

por W3C para describir los servicios. Esta especificación recibe el nombre de lenguaje de descripción de aplicaciones Web (WADL) (W3C, 2009).

De esta forma es posible implementar servicios web a través de la filosofía REST (i.e. servicios *RESTful*). Basta con invocar funciones a través de los puntos de entrada del interfaz HTTP (e.g. GET), así, cualquier motor web (e.g. PHP) asociado a un servidor web (e.g. Apache WS) puede invocar cualquier proceso computacional al ejecutar la respuesta a una invocación GET a través de HTTP. Es en esta invocación donde se implementa la funcionalidad expuesta por la aplicación. El acceso a través de HTTP/GET permite un acceso independiente del motor asociado al servidor web. El intercambio de información se realiza en un formato (e.g. XML o JSON), aislando al invocador (cliente) del servicio del mismo (servidor).

Como ejemplo, supongamos un sistema de mantenimiento de una lista de contactos de empleados. En tal sistema cada usuario debería tener su propia URI con una apropiada representación, además, la colección de recursos es otro recurso. Una vez identificado dos tipos de recursos, habrá dos tipos de URIs:

- *Employee* (Una URI por empleado).
- *AllEmployee* (Listado de los empleados).

A la hora de asignar el formato, tenemos que tener en cuenta que estamos hablando de la representación. Como ya hemos comentado con anterioridad, no se puede acceder directamente al recurso, hay que obtener una representación de este, pudiendo ser un documento HTML, XML, una imagen u otro fichero. Para cada uno de los recursos, se tiene que decidir cuál va a ser su representación.

Continuando con nuestro ejemplo, el formato de representación va a ser XML, ya que es el principal formato para intercambio de información. El formato del empleado podría ser el siguiente (Navarro Marset, 2006-07):

```
<employee xmlns='HTTP://example.org/my-example-ns/'>
  <name>Full name goes here.</name>
  <title>Persons title goes here.</title>
  <phone-number>Phone number goes here.</phone-number>
</employee>
```

Para el listado de empleados podríamos tomar este otro:

```
<employee-list xmlns='HTTP://example.org/my-example-ns/'>
```

```
  <employee-ref href="URI of the first employee"/>
  Full name of the first employee goes here.</employee>
  <employee-ref href="URI of employee #2"/>Full
name</employee>
  .
  .
  <employee-ref href="URI of employee #N"/>Full
name</employee>
</employee-list>
```

2.1.3. Comparación entre REST y SOAP

La tabla 2.1 resume algunas de las características de ambas aproximaciones.

Tabla 2.1: Características de REST y SOAP (Navarro Marset, 2006-07)

	REST	**SOAP**
Características	Las operaciones se definen en los mensajes. Una dirección única para cada instancia del proceso. Cada objeto soporta las operaciones estándares definidas.	Las operaciones son definidas como puertos WSDL. Dirección única para todas las operaciones. Múltiple instancias del proceso comparten la misma operación.
Ventajas declaradas	Bajo consumo de recursos. Las instancias del proceso son creadas explícitamente. El cliente no necesita información de enrutamiento a partir de la URI inicial. Los clientes pueden tener una interfaz "*listener*" (escuchadora) genérica para las notificaciones. Generalmente fácil de construir y adoptar.	Fácil (generalmente) de utilizar. La depuración es posible. Las operaciones complejas pueden ser escondidas detrás de una fachada. Envolver APIs existentes es sencillo Incrementa la privacidad. Herramientas de desarrollo.
Posibles desventajas	Gran número de objetos. Manejar el espacio de nombres (URIs) puede ser engorroso. La descripción sintáctica/semántica muy informal (orientada al usuario). Pocas herramientas de desarrollo.	Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones. Las instancias del proceso son creadas implícitamente.

2.2. Transformaciones entre servicios RESTful y SOAP. Herramientas

Unas de las ventajas de REST respecto a SOAP está dada por la generalidad del interfaz HTTP, permitiendo que cualquier cliente HTTP pueda comunicarse con cualquier servidor HTTP sin necesidad de una configuración previa. A pesar de que los servicios SOAP suelen utiliza HTTP como medio de transporte, este se rige por el protocolo SOAP de intercambio de datos. Los partidarios de SOAP argumentan que gracias a esta tecnología, los diseñadores encapsulan la complejidad de la aplicación, dando lugar a interfaces generadas automáticamente que facilitan el diseño del sistema. Esta polémica ha generado la necesidad de preguntarnos si es posible realizar las transformaciones de RESTful a SOAP y viceversa.

En la actualidad existen herramientas que permiten realizar esta transformación mediante artefactos de mediación ubicados entre cliente y proveedores de información, desarrollándose, principalmente dos enfoques: los buses de servicios, llamado ESB por sus siglas en inglés (*Enterprise Service Bus*) y puertas de enlace o *gateways* SOA.

La mayor parte de estas herramientas son fabricadas por proveedores industriales. Apache ((APACHE, 2011) y (APACHE, 2012)), Fiorano (Fiorano, 2013), IBM (IBM, 2013c), Microsoft (Microsoft, 2013), Mulesoft (MuleSoft, 2013), Oracle (ORACLE, 2013b) y WSO2 (WSO2, 2013) son fabricantes de ESB, así como Cisco (CISCO, 2013), Intel (INTEL, 2013), Layer 7 (LAYER7, 2013), Oracle (ORACLE, 2013a) y Vordel (VORDEL, 2013) son algunos de los desarroladores más importantes de gateways.

Veamos algunas de estas herramientas, así como el mecanismo utilizado en la conversión.

2.2.1. Puertas de enlace de servicios (Service Gateways)

Las *Service Gateway* proporcionan la capacidad de tener una única mediación capaz de controlar las solicitudes de múltiples tipos de consumidores y proveedores de servicios. Estas herramientas están adaptadas para su uso dentro de SOA, y como tal su principal propósitos es encapsular el acceso de una aplicación cliente de los sistemas externos, logrando aislar los detalles de comunicación entre clientes y proveedores de servicios.

Las SOA Gateways fueron concebidas como componentes de software, normalmente implementados en hardware, con el objetivo de ser un interceptor de mensajes (C. Steel, 2005), con el fin de velar por la seguridad de los documentos XML incluidos en el tráfico de SOA. Sin embargo, hoy en día, las SOA Gateways han evolucionado, asumiendo el papel tradicionalmente desempeñado por los bus de servicios empresariales (ESB) (Ryan, 2011).

2.2.1.1. Policy Studio y Vordel XML Gateway

Vordel Policy Studio (Vordel, 2011) , es un potente e intuitivo entorno gráfico, que permite a los desarrolladores trabajar en el entorno familiar e integrado de Eclipse, resultando fácil la configuración, personalización y extensión de la aplicación *Vordel XML Gateway*.

Este entorno gráfico permite a los desarrolladores crear a lo que se le denominan políticas, que no son más que filtros enlazados dentro de un escenario. Los filtros son elementos fundamentales que llevan a cabo la integración específica, las transformaciones, o tareas de seguridad. *Vordel Policy Studio* relaciona más de ciento veinte filtro y políticas que cubren los escenarios de usos comunes, las tecnologías de terceros, y los estándares más utilizados.

Vordel Gateway a menudo referida como puerta de enlace XML o SOA (*Service Oriented Architecture*), está diseñada para proporcionar una integración en tiempo de ejecución, asegurando, controlando y acelerando todo tipo de tráficos entre las aplicaciones y la infraestructura SOA. Su política está impulsada en el procesamiento eficaz de SOAP, REST, XML, JSON y otros formatos.

O'Neill plantea una forma para realizar la conversión de REST a SOAP utilizando *Vordel Gateway XML* en el entorno de *Vordel Policy Studio,* esta herramienta hace de mediador entre las dos arquitecturas permitiendo la conversión (ver figura 2.5).

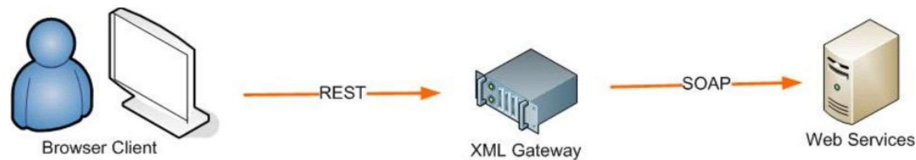

Figura 2.5: Arquitectura de la conversión Vordel (O'Neill, 2008)

Primeramente, se crea una política donde se capturan los parámetros de la URL REST, que se insertaran en un mensaje SOAP que posteriormente será transmitido (ver figura 2.6).

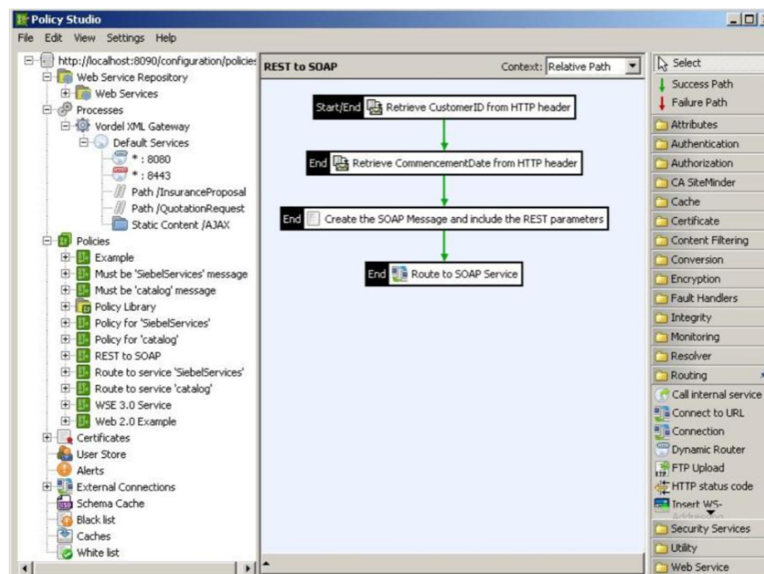

Figura 2.6: Creación de la política de conversión Vordel REST a SOAP (O'Neill, 2008)

La captura de parámetros por parte de la herramienta es posible por unos de los filtros que posee esta propiedad. Una vez obtenidos los parámetros REST, estos son tratados como los posibles atributos del mensaje SOAP. A continuación se crear un mensaje SOAP utilizando el filtro *Set Message* (conjunto de mensajes), y se escriben los parámetros REST como atributos del mensaje, a través de un atributo dinámico que guarda la información REST, este, es procesado por la herramienta obteniéndose un mensaje SOAP.

Como resultado, tenemos un cliente REST, con un objeto *XMLHttpRequest*, puede acceder a una URL, y luego esto se traduce mediante la herramienta *XML Gateway*, enviando una solicitud SOAP.

2.2.1.2. SOA Gateway

SOA Gateway (System, 2008) es un componente de infraestructura básica basado en una implementación SOA con la capacidad de integrar servicios de forma segura, normalmente implementado como un dispositivo de hardware. Un *SOA Gateway* controla el acceso a los servicios, protege la información mediante el cifrado a nivel de datos, garantiza la integridad de un mensaje, y controla el flujo de la información corporativa.

La virtualización de servicios es la práctica más importante de esta herramienta, que se lleva a cabo utilizándola como intermediario entre el cliente y el proveedor de información.

Lascelles (Lascelles, 2010), convierte un servicio *RESTful* codificado en JSON a SOA, utilizando *SOA Gateway*, asumiendo que ya se tiene una puerta de enlace *SecureSpan* desplegada entre los solicitantes potenciales REST y el servicio Web SOAP existente. Primeramente, se crear un punto de acceso de servicios con el fin de poder asignar todo lo que viene en el patrón de URI/posición/*, permitiendo acceder a los verbos *GET* y *POST*. A continuación, con ayuda de la herramienta aislamos la identificación de los recursos de la URI y se guarda la información en una variable de contexto. Para lograr esto, es posible utilizar una expresión regular simple. También, se bifurcará el verbo HTTP entrante mediante una instrucción *OR*, en caso que el servicio este centrado en *GET* y *POST*, pero es posible agregar lógica adicional para otros verbos HTTP.

Para peticiones *GET*, la transformación es muy simple, la herramienta nos permite declara una variable mensaje con un esqueleto SOAP (*SOAP request template),* en la que nos referimos a la variable contexto anteriormente mencionada. Este mensaje SOAP se enruta al servicio web existente y los elementos esenciales se encuentran aislados con las afirmaciones de *XPath.*

Finalmente se transmite el mensaje SOAP, y la información REST contenida en el mensaje se construye nuevamente mediante una plantilla (*Template JSON response*) brindada por la herramienta. Una lógica similar se realiza para el mensaje *POST.*

2.2.2. Enterprise Servicie Bus (ESB)

Un Bus de Servicios Empresariales (ESB por sus siglas en inglés) brinda capacidades de mediación a fin de proporcionar más opciones a los proveedores y consumidores de los

servicios web. Una de las capacidades clave consiste en cambiar el modo en que se invoca un servicio sin tener que realizar cambios al servicio en sí mismo. Por ejemplo, se logra que los servicios web SOAP sean accesibles a los clientes REST. Además, también pueden ser utilizados para exponer servicios RESTful como servicios SOAP.

Desde el punto de vista del negocio, en caso que se cuente con servicios que exponen todas las operaciones requeridas por los consumidores, no deberá realizar cambios a dichos servicios solamente para que se les pueda llamar utilizando otro modo, como por ejemplo, para que puedan ser invocados por REST cuando originalmente fueron implementados para clientes basados en SOAP y viceversa. Todo lo que se necesita es producir el artefacto de mediación adecuado y ponerlo a disposición de los consumidores en el ESB.

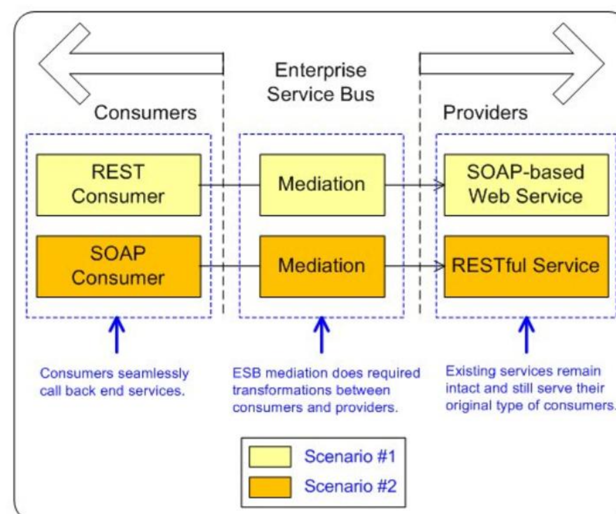

Figura 2.7: Tipología de los ESB (Abbas, 2009)

En la actualidad hay muchas definiciones del término ESB usadas por proveedores, analistas de mercado, y usuarios finales. No existe una definición formal del término, pero se puede apreciar que en general existe una coincidencia en las funcionalidades básicas que un ESB debe proveer. Estas son enumeradas a continuación (Caponi, 2008),

- Transparencia de localización. Los ESBs desacoplan a los servicios proveedores de los consumidores. Estos proveen una plataforma central para comunicar aplicaciones haciendo transparente al consumidor la localización del receptor.
- Conversión de protocolos de transporte. Un ESB es capaz de integrar aplicaciones por más que estas se comuniquen bajo diferentes protocolos de transporte (HTTP, SMTP, FTP, etc).

- Transformación de mensajes. Los ESBs proveen funcionalidades para transformar formatos de mensajes. Por lo general soportan transformaciones basadas en estándares como *XSLT* y *XPath*.
- Enrutamiento de mensajes. Determinar el destinatario de un mensaje, es una de las funcionalidades más importantes de los ESBs.
- Seguridad. Los ESBs proveen funcionalidades de autenticación, autorización y encriptación de mensajes.
- Monitoreo y administración. Los ESBs proveen ambientes de monitoreo y administración necesarios para el control de los flujos de mensajes.
- Soporte a ejecución de procesos de negocio. Permiten la especificación y ejecución de procesos de negocio que orquestan servicios accesibles a través del bus.
- Manejo de transacciones. Los ESBs proveen soporte transaccional en las interacciones con el bus de servicios, lo que permite que se pueda asegurar el envío o recepción de varios mensajes a través de este de forma atómica.

En general cada ESB provee interfaces propietarias con soporte para múltiples plataformas. Esto hace que luego de desarrollada una solución sobre un ESB particular, no sea simple el cambio de proveedor. Un esfuerzo para estandarizar estas interfaces y su comportamiento en el contexto de la plataforma Java, es la especiación JBI *(Java Bussines Integration)*, descrita en la sección 2.2.2.3

2.2.2.1. Oracle Service Bus (OSB)

Oracle Service Bus (ORACLE, 2011) es un ESB que procesa los mensajes entrantes de solicitud de servicio, determina la lógica de enrutamiento, y transforma estos mensajes para la compatibilidad con los consumidores de servicios. Recibe mensajes a través de un protocolo de transporte como HTTP (S), JMS, de archivos o FTP, enviando los mensajes a través del mismo o de un protocolo de transporte diferente. Los mensajes de servicio de respuesta siguen el camino inverso. El procesamiento del mensaje por parte de *Oracle Service Bus* es impulsado por los metadatos, especificado en la definición del flujo de mensajes de un servicio proxy.

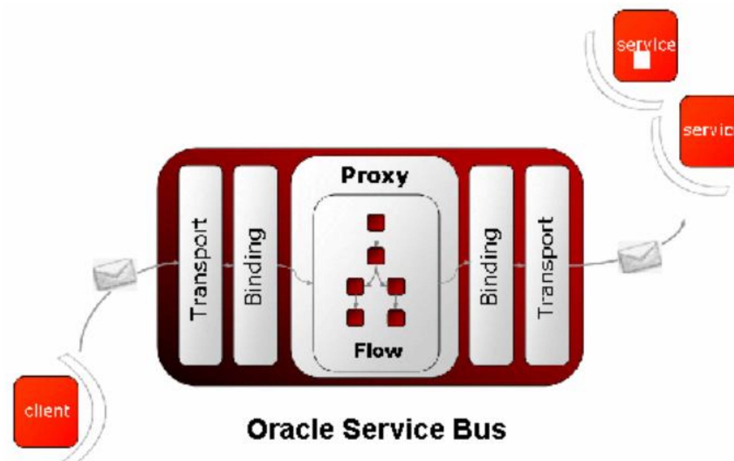

Figura 2.8: Capas de unión y transporte de los OSB (ORACLE, 2011)

Procesamiento de mensajes

Los mensajes pueden contener datos o información sobre el estado de los procesos de aplicación, instrucciones para el destinatario, o ambas cosas. OSB permite enrutar los mensajes en función de su contenido y así, en caso que sea necesario, llevar a cabo transformaciones para lograr la compatibilidad cliente-servidor.

El procesamiento de los mensajes se produce en la siguiente secuencia de eventos:

1. Procesamiento transporte contenido de entrada
2. Procesamiento por parte del flujo de mensaje.
3. Procesamiento transporte contenido de salida

Posteriormente se envía el mensaje a un punto final (ya sea un servicio comercial u otro servicio proxy), y se procesa el mensaje de respuesta en un modelo similar al descrito en la secuencia anterior.

Capa de Transporte (Ida)

La capa de transporte de entrada es la capa de comunicación entre los servicios de cliente (o consumidores de servicios) y el OSB. Es responsable de manejar la comunicación con el punto final de servicio y actúa como punto de entrada para los

mensajes en *Oracle Service Bus*, por consecuente también devuelve los mensajes de respuesta a los consumidores de servicios

Capa de Transporte (Vuelta)

La capa de transporte de salida es la responsable de la comunicación entre los servicios de negocio (o productores de servicios) y el OSB. Su objetivo es mover los mensajes de *Oracle Service Bus* al servicio de negocio o un proxy de servicio, también se encarga de la recepción de la respuesta de los servicios.
Ambas capas no están involucradas en el procesamiento de los datos, pero se encarga de los meta-datos del mensaje, incluyendo el *endpoint* de la URI, los encabezados de transporte, entre otros.

Servicios de proxy

Los servicios de proxy es un concepto fundamental en la arquitectura de *Oracle Service Bus*. Se trata de la interfaz que los consumidores de servicios utilizada para conectar o gestionadas servicios *back-end*. Los servicios de proxy son las definiciones de los servicios Web intermediarios que el bus de servicios implementa a nivel local. La consola de administración de los OSB permite la configuración de un servicio proxy mediante la definición de su interfaz en términos del WSDL y el tipo de transporte que utiliza. La lógica de procesamiento del mensaje se especifica en las definiciones de flujo del mensaje ubicado dentro de las definiciones de un servicio proxy.

Flujo de mensajes.

La implementación de un servicio de proxy se especifica mediante la definición de un flujo de mensajes. Este último define el flujo de los mensajes de la solicitud y respuesta a través del servicio de proxy. Los siguientes elementos constituyen el flujo de mensajes:

- Un nodo de inicio.
- Dos flujos de intercambio, una para la solicitud y otro para la respuesta. Estos flujos de comunicación consisten en una secuencia de etapas que especifican las acciones a realizar durante la solicitud o el procesamiento de la respuesta.

- Un nodo que almacena los valores de las partes designadas del mensaje, contexto del mensaje o la operación invocada.
- Un nodo ruta utilizado para definir el destino del mensaje. Este por defecto es un nodo de eco que refleja la petición como la respuesta.

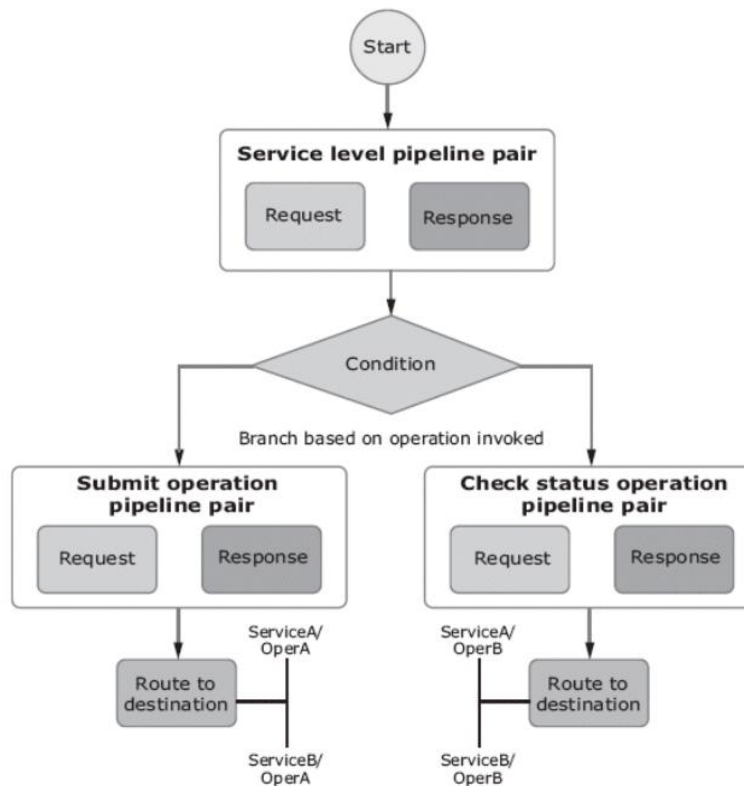

Figura 2.9: Flujo de mensaje de los OSB (ORACLE, 2011)

2.2.2.2. ESB de IBM WebSphere

IBM WebSphere (Abbas, 2009) brinda la mediación requerida para exponer servicios SOAP, para que sean invocados por el estilo REST y viceversa. Esto se realiza mediante *IBM WebSphere Enterprise Service Bus*, *IBM WebSphere Message Broker* e *IBM WebSphere DataPower*.

IBM WebSphere Enterprise Service Bus

Debido a que las interacciones RESTful utilizan las capacidades del protocolo HTTP, *WebSphere Enterprise Service Bus* soporta interacciones RESTful haciendo uso de los enlaces HTTP.

Para poder convertir de REST a SOAP, este componente arquitectónico utiliza una exportación con enlace HTTP para aceptar las solicitudes HTTP de los clientes de REST. La exportación HTTP se conecta a un flujo de mediación que realiza la transformación requerida de la carga útil (cuerpo de la solicitud HTTP) enviada desde el cliente de REST. En consecuencia, el flujo de mediación transforma el contenido de la carga útil en un formato de cadena para formar parte del XML que constituirá la solicitud SOAP que pasará al servicio de *back-end*.

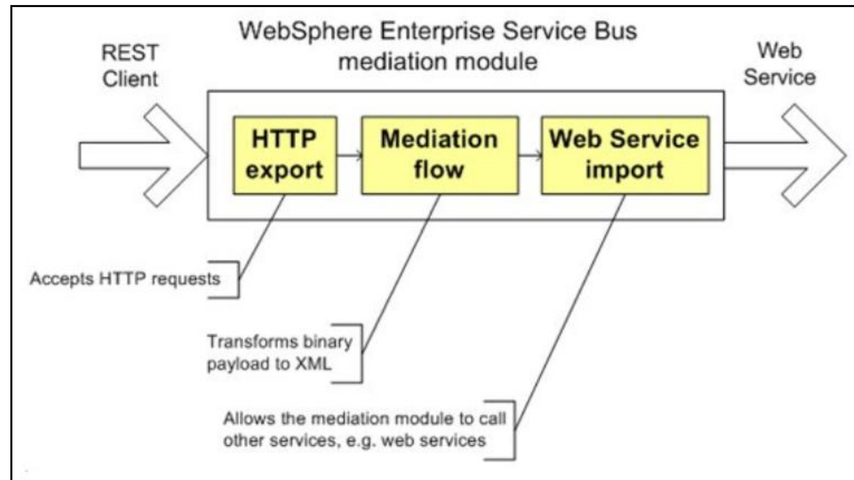

Figura 2.10: Conversión de REST a SOAP a través de IBM WebSphere Enterprise Service Bus (Abbas, 2009)

Para convertir de SOAP a REST la dinámica es la misma, las únicas diferencias son que la exportación usa un enlace de servicio web, la importación usa un enlace HTTP y el flujo de mediación extrae la carga útil de la solicitud SOAP y la envía como carga útil en este caso al servicio RESTful back-end.

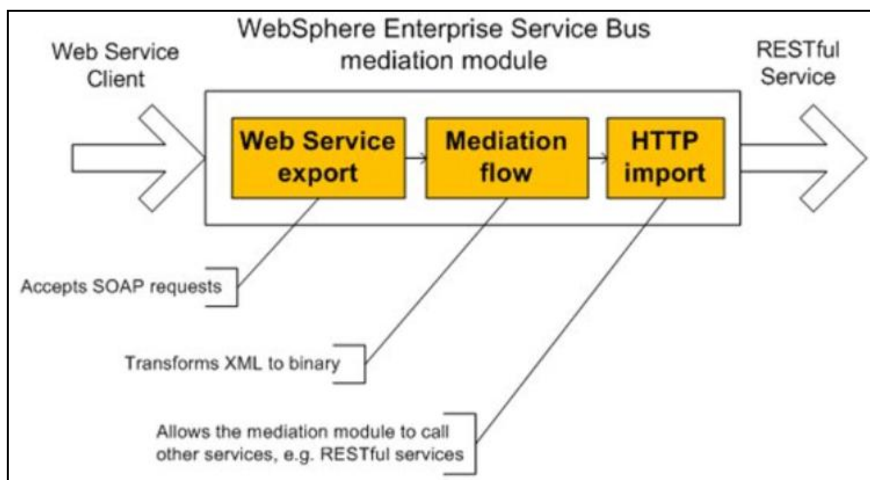

Figura 2.11: Conversión de SOAP a REST a través de IBM WebSphere Enterprise Service Bus (Abbas, 2009)

IBM WebSphere Message Broker

IBM WebSphere Message Broker soporta interacciones RESTful usando tres primitivas dentro de un flujo de mensajes: HTTPInput, HTTPRelpy y HTTPRequest.
Para convertir de REST a SOAP se utiliza *HTTPInput* para aceptar las solicitudes HTTP de los clientes REST, este reenvía las solicitudes a un nodo *Compute* (Calcular) que realiza la transformación necesaria de la carga útil de la solicitud a cadena para poder realizar cualquier otro procesamiento que resulte necesario. El nodo *Compute* invoca el servicio web de *back-end* a través de *SOAPRequest* el cual maneja la comunicación basada en SOAP. *HTTPReply* es el encargado de elaborar la respuesta HTTP para reenviarla al cliente REST que la solicita.

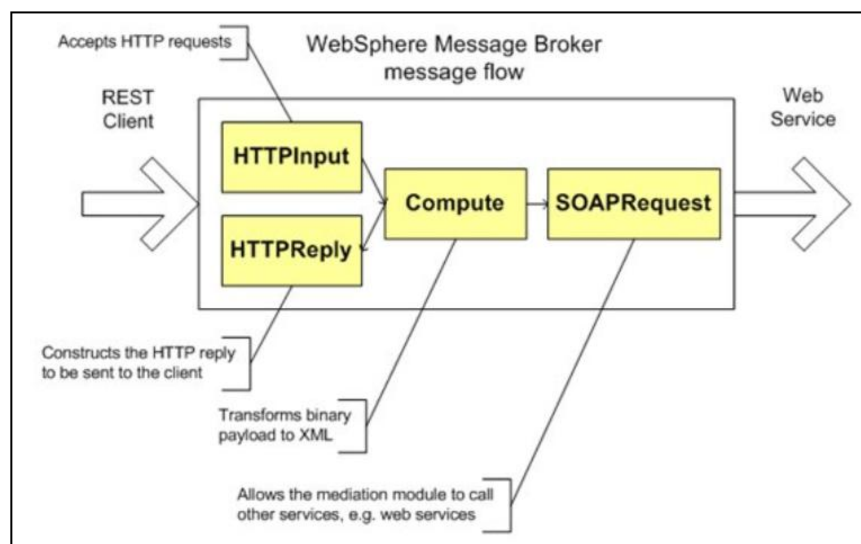

Figura 2.12: Conversión de REST a SOAP a través de IBM WebSphere Message Broker (Abbas, 2009)

En el caso, SOAP a REST, *WebSphere Message Broker* brinda *SOAPInput* para aceptar solicitudes basadas en SOAP. La solicitud se reenvía al nodo *Compute* que realizará la transformación necesarias, luego *HTTPRequest* la encaminada al servidor REST, y posteriormente, se usa *Compute* para transformar el resultado nuevamente a SOAP y así ser enviado al cliente del servicio web que lo solicitara a través de *SOAPReply*.

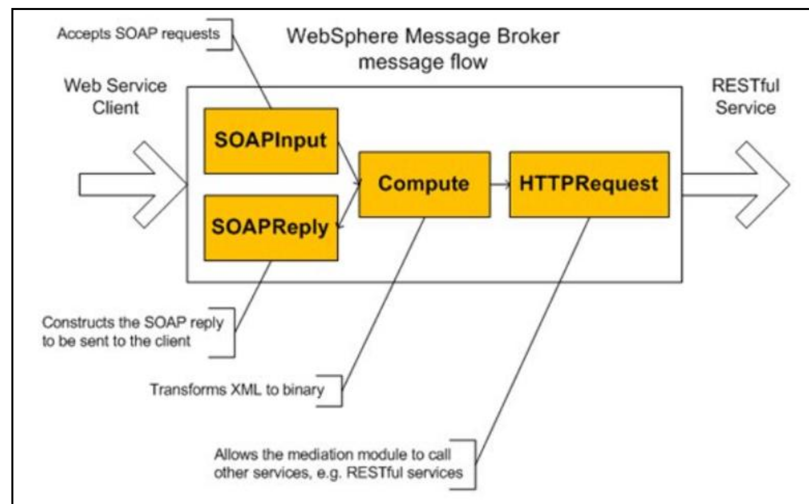

Figura 2.13: Conversión de SOAP a REST a través de IBM WebSphere Message Broker (Abbas, 2009)

IBM WebSphere DataPower

La Figura 2.14 muestra un panorama general de WebSphere DataPower exponiendo una fachada REST contra un servicio web SOAP como back-end. El formato representacional que usa la fachada REST es JSON. La puerta de enlace multiprotocolo acepta las solicitudes del método RESTful (GET, PUT, POST, DELETE) del consumidor de servicios y luego transforma estas solicitudes en las solicitudes SOAP correspondientes, las cuales se envían al WS-Proxy (proxy de servicios web). El WS-Proxy es opcional, ya que las solicitudes SOAP podrían enviarse directamente al back-end, pero esta práctica se desaconseja totalmente debido a que la configuración podría requerir de monitoreo o seguridad en el futuro. Por lo tanto, la mejor práctica consiste en incluir un WS-Proxy como parte de la configuración de transformación REST.

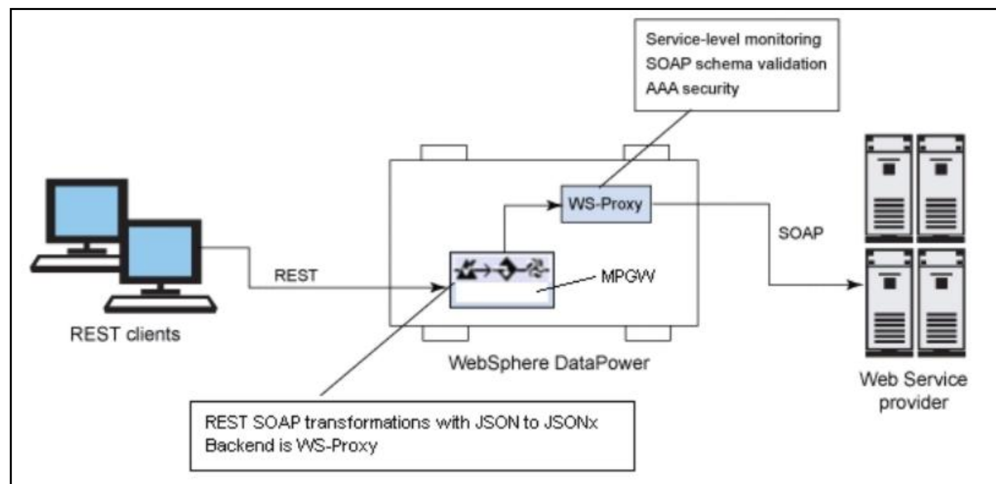

Figura 2.14: Topología de un cliente RESTful que transmite JSON a WebSphere DataPower (Muschett, 2011)

A continuación veremos el modo de implementación recomendado, que usa un Firewall XML para aceptar las solicitudes HTTP de un cliente REST. Como las solicitudes entrantes no están en formato XML, el Firewall XML utilizará un mapa de *WebSphere Transformation Extender* para transformar la carga útil RESTFUL en SOAP y así ser aceptada por los servicios web. También se realiza una transformación en sentido inverso al transformar la respuesta SOAP del servicio web en algún formato RESTFUL, para que pueda ser enviada al cliente REST.

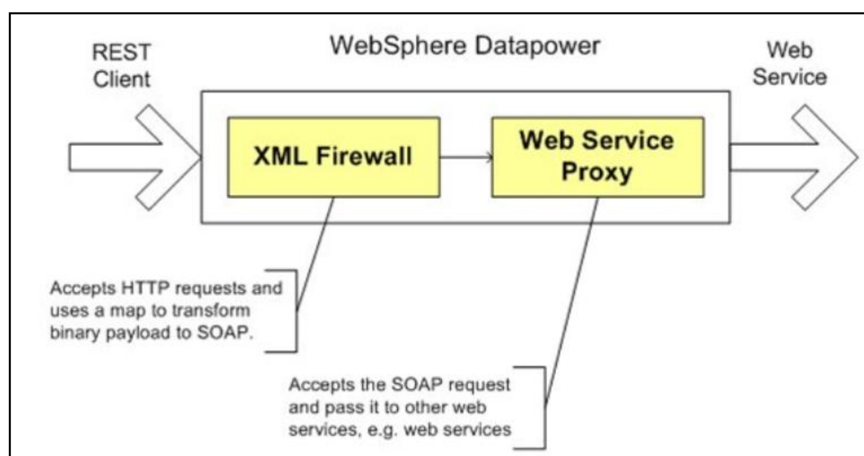

Figura 2.15: Conversión de REST a SOAP a través de IBM WebSphere *DataPower (Abbas, 2009)*

Para lograr convertir de SOAP a REST, básicamente intercambiamos el *Web Service Proxy* con el *Firewall XML* y viceversa. El *Web Service Proxy* acepta las solicitudes SOAP de clientes de servicios web basadas en el WSDL del servicio web de back-end

de muestra, luego pasa la solicitud al Firewall XML para realizar la transformación necesaria entre SOAP y el código RESTful. El Firewall XML también invoca el servicio RESTful de back-end.

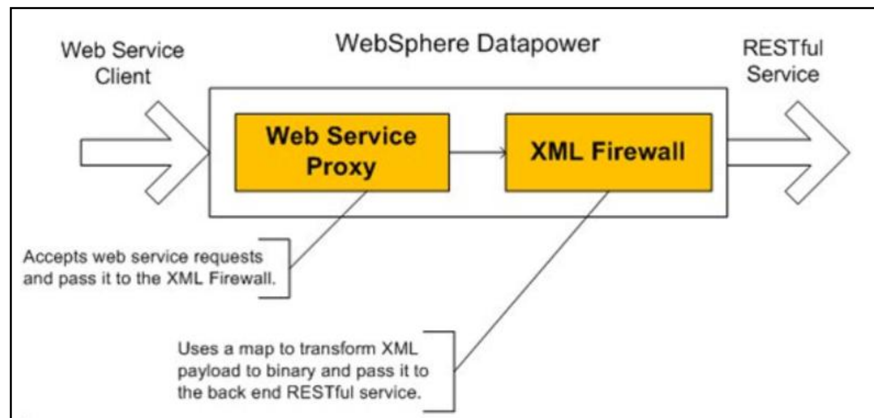

Figura 2.16: Conversión de SOAP a REST a través de *IBM WebSphere DataPower* (Abbas, 2009)

2.2.2.3. ESB de Apache

Los ESB comerciales suelen disponer de ciertas herramientas tipo GUI que facilitan la administración de los distintos componentes, además de contar con soporte técnico y documentación de referencia. Apache ha optado por implementar ESB no comerciales, aunque algunos de los ESB pertenecientes a este mundo suelen carecer de interfaces ricas de administración y en general la documentación deja mucho que desear, por el contrario ofrecen un rango mayor de posibilidades de integración y un alto nivel de extensibilidad gracias a estar basados la mayoría de las veces en estándares.

Apache ServiceMix

El objetivo del desarrollo de Apache ServiceMix es brindar un ESB que implemente completamente la especificación JBI, con foco en la extensibilidad, confiabilidad y soporte para interconexión con múltiples plataformas (Caponi, 2008), esto se logró con la publicación de la versión 3.3, donde el proyecto Apache ServiceMix dio por concluido el desarrollo de un ESB compatible por completo con el estándar JBI *(Java Bussines Integration)*.

Actualmente, contamos con Apache ServiceMix 4, este supone un cambio en la arquitectura del ESB alineándose en este caso con el estándar OSGi (*Open Services*

Gateway Initiative). No obstante, los componentes JBI desarrollados para la versión 3 seguirán funcionando perfectamente en la versión 4 (Vázquez, 2009).

Java Bussines Integration (JBI)

Java Bussines Integration (JBI) es una especificación creada por la *Java Community Process* (JCP). En esta se define una arquitectura que permite construir aplicaciones en base a integración de componentes (*plug-in*), los cuales pueden inter-operar intercambiando mensajes en un formato XML estándar. La figura 2.17 ilustra tal arquitectura.

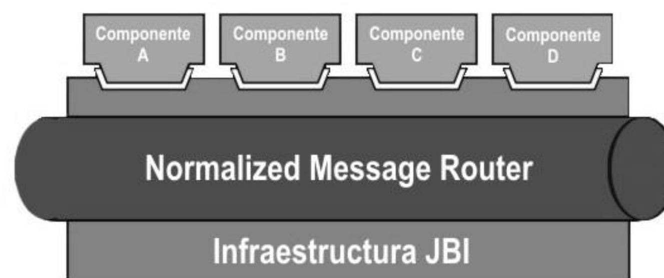

Figura 2.17: Arquitectura del JBI (Caponi, 2008)

Los *plug-in* no interactúan directamente sino que lo harán a través de la infraestructura JBI, que será la encargada de resolver el ruteo de mensajes entre estos. La figura 2.18 ilustra tal interacción donde se puede apreciar que los proveedores y consumidores de servicios están desacoplados entre sí.

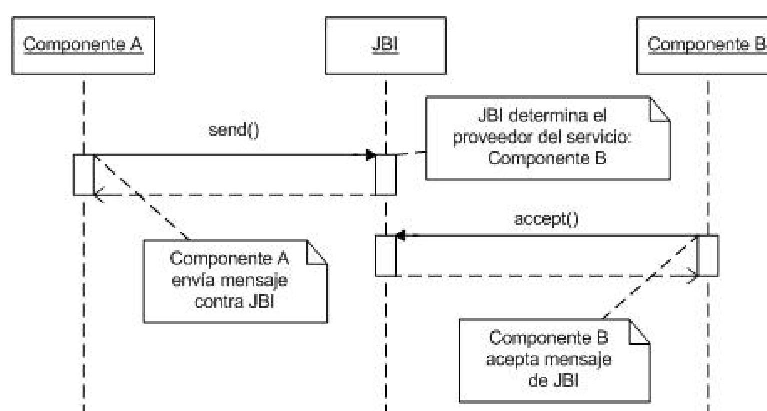

Figura 2.18: Interacción entre componentes JBI (Caponi, 2008)

Según la especificación, hay dos tipos de componentes *plug-in* (SE y BC) (Caponi, 2008).

Service Engine (SE)

Los SEs son contenedores estándar donde residen proveedores y consumidores de servicios internos al ambiente JBI. Ejemplos de SEs pueden ser, transformadores de datos, motores de reglas de negocios, etc.

Binding Component (BC)

Los BCs conectan al entorno JBI con el exterior, esto es, con consumidores de servicios y proveedores externos. Por citar un ejemplo de BC, podemos considerar a un contenedor que expone un *Web Service* al exterior, y una vez que recibe los requerimientos externos, los redirige a un componente de tipo SE en el cual reside la lógica de procesamiento de esos requerimientos. Los BCs pueden integrar aplicaciones que eventualmente no estén escritas en Java, brindando soporte a la integración multiplataforma.

La columna vertebral del entorno JBI es el denominado *Normalized Message Router* (NMR). La finalidad de este es recibir mensajes de todos los componentes JBI (SEs y BCs) y rutear estos a su correspondiente componente JBI destinatario. Dado que los componentes JBI no interactúan directamente, sino que lo hacen a través del NMR, ambos componentes pueden manejar formatos de mensajes diferentes, por lo que el NMR transporta y rutea mensajes normalizados (*normalized messages*). Estos son mensajes en un formato XML estándar que consisten de dos partes: la primera contiene metadatos utilizados por la infraestructura JBI; la segunda contienen los datos en sí mismos, más conocido como *payload*. La forma en que los componentes JBI se comunican con el NMR es a través de los *Delivery Channels* (DC). A través de estos se entregan los mensajes en formato normalizado al NMR, luego de una transformación del formato de mensaje particular que maneja el componente, al formato estándar. En la figura 2.19 podemos ver la interacción antes mencionada.

Para especificar la forma en que interactúan los componentes JBI a partir de mensajes, la especificación se apoya en los patrones de intercambio de mensajes o *Message Exchange Patterns* (MEPs) definidos en la especificación WSDL 2.0 (Caponi, 2008).

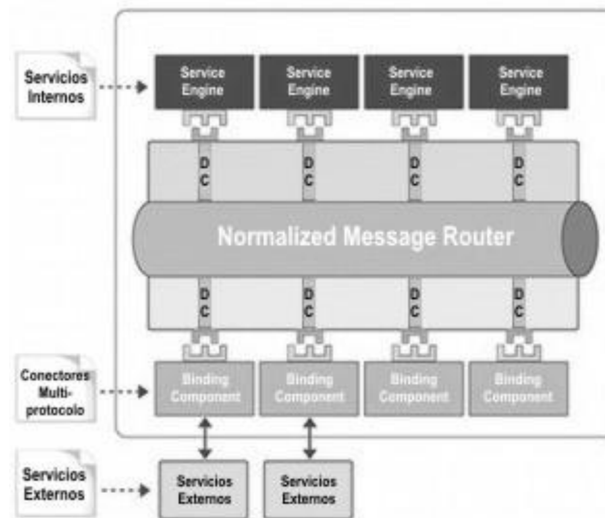

Figura 2.19: Interacción entre componentes SE (servicios externos) y BC(controladores multiprotocolo) a través de NMR (Caponi, 2008)

El objetivo del desarrollo de Apache ServiceMix es brindar un ESB que implemente completamente la especificación JBI, con foco en la flexibilidad, confiabilidad y soporte para interconexión con múltiples plataformas. Se apoya sobre el proveedor de mensajería Apache ActiveMQ, mediante el cual se lograr confiabilidad y posibilidades de trabajo en ambientes distribuidos y de *cluster*. Además se integra con varios productos de integración, como por ejemplo *Apache CXF, Apache ODE, Apache Camel, Apache Geronimo, JBoss* y cualquier contenedor web. Incluye múltiples componentes JBI que proveen las funcionalidades de base de un ESB. En la tabla 2.2 se describen algunos de estos componentes (Caponi, 2008).

Tabla 2.2: Componentes JBI incluidos en *ServixMix* (Caponi, 2008)

Componente	Tipo	Descripción
servicemix-bean	SE	Permite el desarrollo de componentes de lógica basados en POJOs.
servicemix-eip	SE	Provee implementaciones de varios de los patrones de mensajería
servicemix-camel	SE	Permite el uso de Apache Camel para la implementación de los patrones de mensajería

servicemix-saxon	SE	Permite realizar transformaciones sobre mensajes especificadas mediante XSLT y XQuery.
servicemix-cxf	SE/BC	Permite publicar servicios en el bus NMR y/o al exterior. Se basa en el stack Apache CXF, con lo que se agregan todas sus funcionalidades (implementación de varias WS-*).
servicemix-_le	BC	Provee integración con sistemas de archivos. Puede ser utilizado para escribir o leer archivos.
servicemix-http	BC	Provee integración con el protocolo HTTP, que permite la invocación de servicios publicados en el NMR a través de mensajes SOAP sobre HTTP.
servicemix-ftp	BC	Provee integración con servidores FTP. Puede ser utilizado para escribir o leer archivos sobre FTP.
servicemix-jms	BC	Provee integración con sistemas de mensajería que soportan JMS.
servicemix-mail	BC	Provee soporte para el envío y recepción de correos electrónicos.

2.3. Arquitectura dirigida por modelos (MDA)

La arquitectura dirigida por modelos (MDA) es una aproximación para el desarrollo de software definido por el OMG (*Object Management Group*). La clave de MDA es el papel que juegan los modelos en los procesos de desarrollo.

Figura 2.20: Ciclo de vida de desarrollo del software tradicional (Kleppe, 2003)

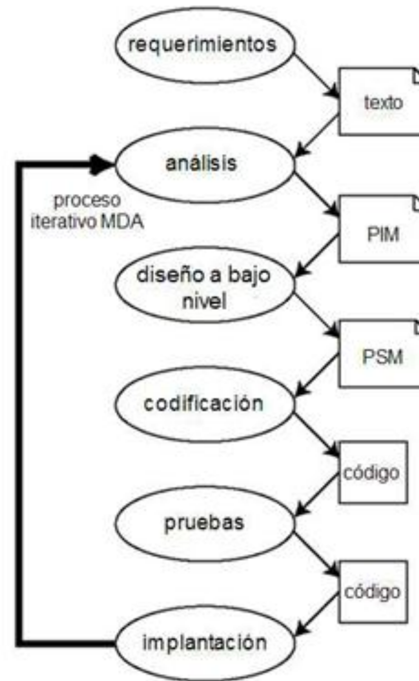

Figura 2.21: Ciclo de vida de desarrollo del software en MDA (Kleppe, 2003)

Se puede observar en la Figura 2.21, que el ciclo de vida de desarrollo MDA, no es muy diferente del tradicional, descrito en la Figura 2.20.

Independientemente del enfoque dado al ciclo de vida la diferencia fundamental entre ambos enfoques está dada en la comunicación entre las fases del proceso de desarrollo, realizándose en MDA mediante modelos. Básicamente existen tres tipos de modelos que constituyen la base de MDA (OMG, 2003):

- Modelos independiente del cómputo (CIM). Los requisitos del sistema se modela a través de los CIM estos modelos son asimilables a un modelo de dominio y/o de negocio. Mostrando el entorno de operativo del sistema.
- Modelos independientes de la plataforma (PIM): son modelos con un alto nivel de abstracción que son independientes de la tecnología en la que se van a implantar. Describen el sistema desde el punto de vista de los procesos de negocio que va a soportar. Así, un PIM de una biblioteca hablaría de servicios de la aplicación de la biblioteca, de los objetos del negocio Usuario, Ejemplar, Préstamo, entre otros.

- Modelos específicos de plataforma (PSM): especifican el sistema en términos de las construcciones que se va a implementar a la hora del desarrollo. Un modelo PIM puede generar distintos modelos PSM, en función de las tecnologías utilizadas. Así, un PSM de una biblioteca hablaría de JSP (SUN JSP), conexiones JDBC (SUN JDBC), etc.
- Código: la fase final del desarrollo es transformar cada PSM en código. Como cada PSM es relativo a una tecnología determinada esta transformación es, teóricamente, relativamente sencilla.

MDA define los modelos PIM, PSM y el código, así como la manera de relacionarse unos con otros. Los modelos PIM se deben crear, después se deben transformar en uno o varios PSM (el paso más complejo en los desarrollos MDA) y finalmente transformarlo en código.

La novedad de MDA frente al desarrollo tradicional, es que las transformaciones se pueden hacer mediante herramientas que las ejecutan de forma automática. En concreto, la mayor aportación de MDA y su mayor beneficio, es la transformación de modelos PIM a modelos PSM.

Sin embargo, estas transformaciones, no siempre se pueden realizar de forma automática. El propio OMG, indica en los diferentes métodos de transformación que éstas pueden realizarse de forma manual, mediante el uso de perfiles, a través de marcas y patrones o de forma automática. En general se utiliza una mezcla entre todas ellas y el programador va a tener que especificar algunas de forma manual, pero la idea de las transformaciones automáticas es permitir a los desarrolladores tener retroalimentación de forma rápida de un modelo PIM, ya que va a poder generar prototipos de forma inmediata.

Con este planteamiento, el desarrollo se focaliza en la definición de los modelos PIM, ya que tanto los PSM, como el código se van a generar mediante transformaciones. Es cierto que la transformación precisa del esfuerzo que requiere una tarea tan especializada, pero la ventaja es que haciéndola una vez, se puede aplicar en muchos sistemas. Los desarrolladores definen mejor sus modelos porque se aíslan de los detalles técnicos, lo que permite centrarse exclusivamente en los detalles del negocio concreto, obteniendo eficiencia (buenos resultados en menos tiempo). Por otro lado las

trasformaciones, generarán tanto el PSM, como el código de una forma rápida, sin perder ningún detalle técnico, ya que todos ellos quedan definidos en las transformaciones.

Al contrario que el planteamiento tradicional, MDA obtiene teóricamente una alta productividad debido a que, la documentación generada en las primeras fases, es algo más que documentación. Cualquier persona, con la formación adecuada, será capaz de leer y comprender los modelos que a su vez sirven como punto de partida para la generación automática del código. Así mismo, los modelos PIM no se abandonan al comenzar a codificar. Todo lo contrario. Cada vez que se modifican los requisitos del sistema, no se va a modificar el código, como ocurría en el enfoque tradicional, si no que se va a modificar directamente el PIM y se podrían regenerar desde éste el PSM y el código.

La portabilidad en MDA se consigue gracias a su propio planteamiento. Siempre se va a partir del mismo PIM y en el caso de tener que migrar el sistema a otra tecnología, sólo será necesario generar el PSM apropiado para la nueva plataforma. Sólo es necesario tener una herramienta que realice la transformación, que puede encontrarse en el mercado para tecnologías con una alta tasa de uso o que haya que construirla, en caso de ser una tecnología poco utilizada.

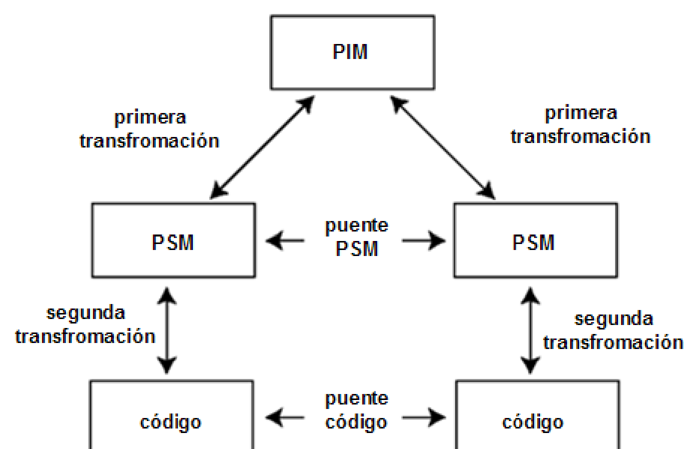

Figura 2.22: Interoperabilidad de MDA mediante puentes (Kleppe, 2003)

Como hemos visto, un PIM puede generar uno o varios PSM, en función de las plataformas en las que se vaya a implementar el sistema. El conjunto resultante de PSM, sin embargo, no estarán directamente comunicados entre ellos. Para conseguir la

interoperabilidad, hay que transformar conceptos de una plataforma en los de otra, construyendo lo que en terminología MDA se llaman puentes (*bridges*). Esta idea está mostrada en la Figura 2.22. Como conocemos los detalles técnicos de ambas plataformas, ya que sin ellos no se podrían definir las transformaciones, podemos obtener toda la información necesaria para construir los puentes entre los dos PSM. Lo mismo ocurre a nivel de código.

2.4. Arquitectura de metamodelado de cuatro capas

La arquitectura de metamodelado de cuatro capas define qué papeles juegan los estándares MDA definidos por OMG. Cada uno de ellos (UML, MOF, QVT, etc.) tiene su lugar y su sentido en esta definición de capas.

Las capas definidas por OMG son las siguientes:

- Capa M0: Las instancias. Es la capa en donde se ejecuta el sistema, donde están las instancias reales que se han creado durante la ejecución. A este nivel existirán las instancias de un gato llamado "Tom" o un ratón llamado "Jerry". Como es lógico podrán existir varias instancias de cualquier objeto. "Mickey", por ejemplo, será otro ratón de nuestro sistema.
- Capa M1: El modelo del sistema. A este nivel se define el modelo del sistema o la aplicación propiamente dicha. Es donde se reflejarán los conceptos existentes en nuestro sistema, tales como Ratón y Gato, junto con sus propiedades. Los conceptos a este nivel son categorizaciones o clasificaciones de las instancias del nivel M0. Esto es, cada elemento del nivel M0 es una instancia del nivel M1 (por ejemplo "Jerry" es una instancia de Ratón).
- Capa M2: Metamodelo. Este nivel contiene los elementos del lenguaje de modelado o metamodelo. El lenguaje tendrá elementos como Clases y Atributos que permitirán definir el modelo del sistema de la capa M1. Al igual que en el nivel anterior, todos los conceptos definidos en el nivel M1 son instancias de los elementos definidos en este nivel. Así tenemos que los conceptos Ratón y Gato, son instancias del elemento Clase del metamodelo. Igualmente, cada propiedad es una instancia de Atributo.
- Capa M3: Meta-metamodelo. Siguiendo la misma línea que los anteriores, podemos definir los elementos existentes en la capa M2, mediante instancias de

elementos existentes en esta capa. Realmente esta capa define un lenguaje de modelado para el metamodelo de la capa inmediatamente anterior. En definitiva, en este nivel está definido el meta-metalenguaje o meta-metamodelado.

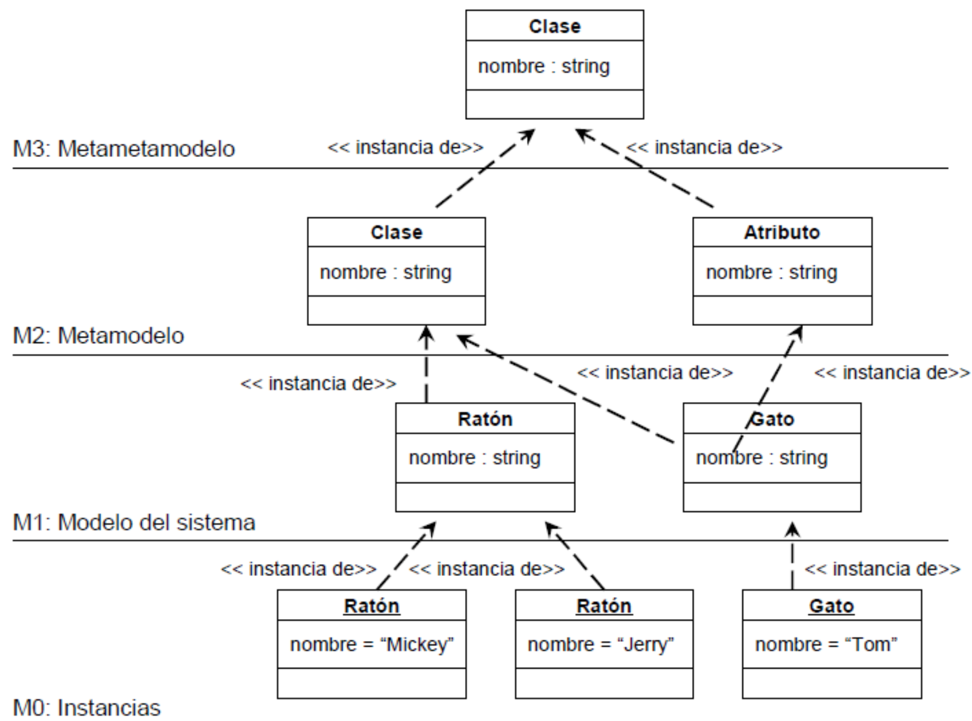

Figura 2.23: Jerarquía de metamodelos en la arquitectura de cuatro capas (OMG, 2003)

La idea fundamental que hay detrás de esta definición de niveles jerárquicos es que existen elementos que cuyas instancias están asociadas a instancias de otros elementos. Si nos centramos en los estándares de OMG, estos elementos serán clases MOF que se podrán instanciar en clases UML, que a su vez se instanciarán en las clases del modelo del sistema. En definitiva con esta jerarquía de niveles se observa que M3 describe el paradigma del modelado. Es razonable contemplar la posibilidad de definir quinto nivel M4 que fuera el meta-metametamodelo. Igualmente, se podría definir una jerarquía de modelos de forma infinita, al menos desde el punto de vista teórico. El nivel M3 es un lenguaje de modelado, por lo que se podría definir un metamodelo de M3. No obstante, para definir este hipotético nivel M4, podemos usar el propio lenguaje definido en M3, ya que M3 es auto-descriptivo. En base a esto, OMG limita esta recursividad a un número discreto, definiendo cuatro niveles ya expuestos.

2.5. Estándar QVT

Para soportar transformaciones se ha seguido el estándar *Query View Transformation* (QVT) (OMG, 2011) propuesto por la OMG. La especificación de QVT depende de otros dos estándares de la OMG como son MOF 2.0 (OMG, 2006a) y OCL 2.0 (OMG, 2006b). De esta manera, la utilización del estándar QVT para especificar transformaciones, aporta reutilización de tecnología que sigue estándares y reducción de la curva de aprendizaje de la herramienta.

La especificación QVT se define a través de dos dimensiones ortogonales: la dimensión del lenguaje y la dimensión de la interoperabilidad. La dimensión del lenguaje define los diferentes lenguajes de transformación presentes en la especificación QVT. Concretamente son tres: *Relation*, *Core* y *Operational* y la principal diferencia entre ellos es su naturaleza declarativa o imperativa (OMG, 2011).

Paquetes QVT

QVT define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational (ver figura 2.24) (OMG, 2011).

El paquete “QVTBase” define una estructura común para las transformaciones. El paquete “QVTOperational” hereda del “QVTRelation” y usa el mismo *framework* para trazabilidad definido en este paquete. Además, usa expresiones imperativas definidas en el paquete “ImperativeOCL”. Todo QVT depende del paquete “EssentialOCL” que proporciona OCL 2.0 (OMG, 2006b) y todos los paquetes de lenguajes dependen del paquete EMOF (OMG, 2011).

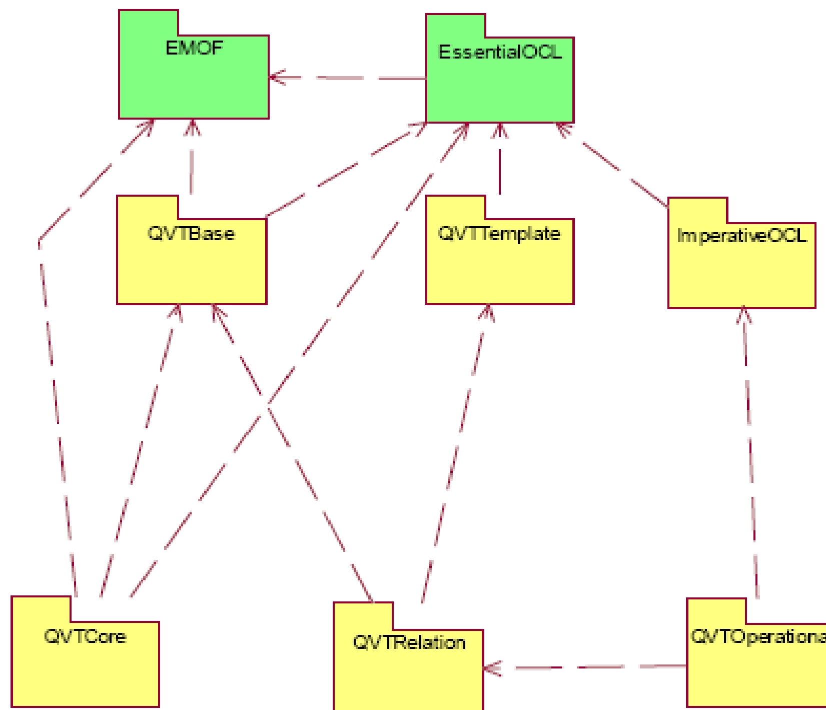

Figura 2.24: Dependencia de paquetes en la especificación QVT (OMG, 2011)

En este trabajo se ha elegido QVT Relations como lenguaje de transformación por proporcionar una especificación declarativa de relaciones entre modelos MOF; soporta *patter maching* de objetos complejos y de manera implícita crea clase de trazabilidad y sus instancias correspondientes, de forma que permite guardar todo lo que ocurre durante la ejecución de una transformación.

2.5.1. QVT Relations

El lenguaje Relations proporciona una especificación declarativa de relaciones entre modelos MOF. Soporta *pattern matching* de objetos complejos y de manera implícita crea clases de trazabilidad y sus instancias correspondientes, de forma que permite guardar todo lo que ocurre durante la ejecución de una transformación (OMG, 2011).

Transformaciones entre modelos

En el lenguaje QVT Relations, una transformación entre modelos candidatos se especifica como un conjunto de relaciones que han de tener lugar para llevar a cabo la transformación. Un modelo candidato es cualquier modelo que se ajuste a un tipo de modelo o metamodelo, que es una especificación de los diferentes tipos que pueden

tener los elementos del modelo que lo conforma. Los modelos candidatos tienen un nombre y los tipos de los elementos que pueden contener, los cuales quedan restringidos al paquete al que se hace referencia en la declaración del modelo candidato (OMG, 2011).

Una *"transformation"* invocada para realizar una transformación se ejecuta en una dirección determinada seleccionando uno de los modelos candidatos como modelo destino u objetivo. El modelo destino debe estar vacío o contener elementos para ser relacionados por la transformación. El procedimiento de transformación se basa en comprobar que todas las relaciones definidas en la transformación se cumplen y para aquellas relaciones que no se cumplen, se intenta cumplirlas creando, eliminando o modificando el modelo destino (OMG, 2011).

Esta política ha sido aplicada entre los metamodelos propuestos en este trabajo validando el marco de conversión definido.

3. Metamodelo REST

REST (*Representational State Transfer*) término con el que Roy T. Fielding denomina al estilo arquitectónico que presenta en su Tesis Doctoral titulada *"Architectural Styles and the Design of Networkbased Software Architectures"* no es un protocolo, una arquitectura software, un producto software, un estándar o, un nombre extravagante para denotar a los Servicios Web (Roy T. Fielding, 2008).

Según Fielding, una *arquitectura de software* define una abstracción de los elementos en tiempo de ejecución de un sistema de software durante alguna fase de su operación (en términos de componentes e interrelaciones entre estos). Un sistema puede componerse de distintos niveles de abstracción (y de diferentes granularidades de componentes) y de diferentes fases de operación, cada una con su propia arquitectura (Roy T. Fielding, 2008).

Un *estilo arquitectónico de software* es un conjunto (nombrado) de restricciones sobre la interacción de diferentes componentes que, cuando se respeta, dota a la arquitectura resultante de determinadas propiedades (deseables) (Roy T. Fielding, 2008).

Por tanto, según Fielding, la arquitectura es una abstracción de la implementación del sistema, y los estilos son los patrones nombrados mediante los cuales podemos comprender mejor la arquitectura y el diseño arquitectónico.

Otros autores no hacen está sutil distinción entre arquitectura y estilo arquitectónico, y proporcionan catálogos de patrones que caracterizan arquitecturas de software (Alur, 2003; Fowler, 2002; Steel, 2005).

REST en sí, es un estilo a alto nivel que podrá ser implementado utilizando muchas tecnologías diferentes, e instanciado utilizando diferentes valores para sus propiedades abstractas. Por ejemplo, REST incluye concepto de recursos y una interfaz uniforme, o sea, la idea de que todo recurso debería responder a los mismos métodos; pero REST no especifica qué métodos deben ser, ni cuantos deberían ser.

En la actualidad se utiliza el término *RESTful* para denominar los servicios web basados en el estilo REST. Stefan Tilkov (Tilkov, 2007), menciona en su artículo el término

“HTTP Restful”, alegando que una encarnación del estilo REST es HTTP o de forma un poco más abstracta: la arquitectura de la Web en sí.

Este estilo arquitectónico se rige por los siguientes principios básicos (Saugar, 2010-2011):
Un recurso es cualquier cosa que tenga identidad y debe estar identificado mediante algún mecanismo. El identificador de un recurso no expone detalles acerca de su implementación.
Todos los recursos deben compartir la misma interfaz uniforme. Los métodos definidos en esta interfaz deben ser utilizados acorde a la semántica que se les impuso al crearlos.
El formato de las representaciones debe estar documentado y ser estándar. Las representaciones deben incluir enlaces a otros recursos relacionados *(“hypermedia as the engine of application state”* HATEOAS).
La comunicación debe ser auto-contenida (stateless). *“Cada solicitud del cliente al servidor debe contener toda la información necesaria para conocer la petición, y no se puede tomar ventaja de cualquier contexto almacenado en el servidor.”* (Roy Thomas Fielding, 2000)
Si bien los principios de REST fueron dictados por Fielding en el año 2000, debemos tener en cuenta que pertenecen a una época donde los servicios web no eran un mecanismo difundido en informática. Hoy el universo de la web es mayor, y mezclar hiperenlaces con servicios web es restringir este universo.

Un hiperenlace tiene que ser iniciado desde la ventana de un navegador, sin embargo, un servicio web puede ser iniciado desde un programa Java. Por tanto, tenemos que tener cuidado con los principios de REST para no ver los servicios RESTful como algo que se invoca desde un *href* HTML, cuando eso, es una visión limitada de los servicios web.

Otro aspecto a tener en cuenta es que muchas de las aplicaciones que usan HTTP no siguen los principios de REST, por ejemplo, usar HTTP GET para invocar operaciones, como eliminar un objeto, viola una regla de seguridad de REST (violación de la semántica del método).

Esto puede plantear un problema en el proceso de conversión: ¿cómo se convertiría un interfaz WSDL con nueve operaciones en un interfaz *RESTful* con cuatro métodos

(GET, POST, PUT y DELETE)? En este sentido, hemos preferido mantenernos fiel a la filosofía RESTful, como los marcos de publicación *RESTful* actuales (p.e. JAX-RS (ORACLE, 2010)), suponer la existencia de sólo cuatro métodos por servicio *RESTful*. Así, un interfaz WSDL con más de cuatro métodos tendría que dividirse en varios servicios web *RESTful*.

Web Application Description Language (WALD)

WADL, es un lenguaje de descripción similar a WSDL, estrictamente dirigido a los requisitos de los servicios *RESTful*. REST comenzó bastante simple, pero a medida que ha aumentado el interés por los servicios Web, proporcionalmente aumenta el estudio de los enfoques alternativos, y los descriptores se convierten a su vez en una adición natural.
Los servicios RESTful se basan en los métodos HTTP- GET, POST, PUT, DELETE. Los parámetros de entrada por otro lado también pueden llegar a ser numerosos para servicios complejos, por lo que el uso de estos valores es necesario. Cuando se trata de valores de respuesta, estas también han crecido en complejidad, que van desde espacios de nombres XML personalizados a JSON, sin mencionar el manejo de escenarios de fallos en caso de que la solicitud sea cancelada.
En la siguiente figura se muestra un descriptor WADL (Rubio, 2007).

```
<?xml version="1.0"?>
<application xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:yn="urn:yahoo:yn"
  xmlns:ya="urn:yahoo:api"
  xmlns="http://research.sun.com/wadl">

  <grammars>
    <include href="NewsSearchResponse.xsd"/>
    <include href="NewsSearchError.xsd"/>
  </grammars>

  <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource uri="newsSearch">
      <method href="#search"/>
    </resource>
  </resources>

  <method name="GET" id="search">
    <request>
      <query_variable name="appid" type="xsd:string" required="true"/>
      <query_variable name="query" type="xsd:string" required="true"/>
      <query_variable name="type" type="xsd:string"/>
      <query_variable name="results" type="xsd:int"/>
      <query_variable name="start" type="xsd:int"/>
      <query_variable name="sort" type="xsd:string"/>
      <query_variable name="language" type="xsd:string"/>
```

```
    </request>
    <response>
      <representation mediaType="application/xml"
      element="yn:ResultSet"/>
      <fault id="SearchError" status="400" mediaType="application/xml"
        element="ya:Error"/>
    </response>
  </method>

</application>
```

Figura 3.1: Descriptor WALD

Un contrato WADL es bastante auto descriptivo por sí mismo, pero por supuesto hay más cosas que puedes hacer con él. Una de las acciones más notables de tomar, es crear clases *stub* de estos contratos WADL para facilitar la creación de clientes de servicios - un proceso idéntico al de los contratos basados en SOAP WSDL (Rubio, 2007).

Si bien la mecánica del uso de este enfoque (generación de código automático) ha provocado críticas entre los primeros analistas WADL, afirmando que no es sólo innecesario, sino que comenzaran los servicios RESTful por el mismo camino que SOAP, que dependen de lenguajes intermedios o descriptores (Rubio, 2007).

Desde un punto de vista práctico, no podemos negar que el uso de este tipo de contrato para obtener clases *stub* es una forma aún más rápida para comenzar el manejo de servicios Web clientes basados en REST.

WSDL 2.0 para describir servicios web REST

Hasta aquí hemos visto que *REST* como estilo arquitectónico generalmente se refiere a una arquitectura de servicios Web basada en los recursos que utiliza HTTP y puede utilizar como representación el esquema XML.

El estándar WSDL no apoyaba por igual los servicios *SOAP* y *RESTful*. En WSDL 1.1 *HTTP Binding* era inadecuado para describir la comunicación entre HTTP y XML, así que no había manera de describir formalmente los servicios RESTful con WSDL.

La publicación de WSDL 2.0, diseñado pensando en los servicios web basados en REST, fue declarado en junio de 2007 como una recomendación de W3C, significando que existe un lenguaje estándar para describir los servicios Web *RESTful (Mandel, 2008).*

Esta segunda versión de WSDL se creó para hacer frente a los problemas de WSDL 1.1, muchos de los cuales habían sido identificados por la organización de Interoperabilidad de los Servicios Web (WS-I) *(Mandel, 2008).*

La utilización de WSDL se extiende más allá de su uso como un contrato API. Siendo una definición formal, WSDL puede ser consumido por las herramientas de servicios Web para realizar acciones, como por ejemplo *(Mandel, 2008):*

- Generar cliente y servicio *stubs* en varios idiomas.
- Publicar un servicio Web.
- Probar dinámicamente un servicio Web.

WSDL 2.0 es un lenguaje XML con núcleo en el espacio de nombres *http://www.w3.org/ns/wsdl*. El elemento raíz de un documento WSDL 2.0 es denomina *Description*. La descripción contiene cuatro elementos secundarios (ver figura 2.2) que juntos encapsulan todos los detalles acerca de un servicio Web *(Mandel, 2008)*

```
<wsdl:description xmlns:wsdl="http://www.w3.org/ns/wsdl">
  <wsdl:types/>
  <wsdl:interface/>
  <wsdl:binding/>
  <wsdl:service/>
</wsdl:description>
```

Figura 3.2: Esqueleto de un documento WDSL 2.0 (Mandel, 2008)

La estructura de un documento WSDL 2.0 difiere en algunos aspectos de WSDL 1.1. La tabla 3.1 muestra algunas de las diferencias más notables entre las versiones.

Tabla 3.1: Diferencias notables entre WSDL 1.1 y WSDL 2.0 (Mandel, 2008)

WSDL 1.1	WSDL 2.0	Comentarios
Definition	Description	El elemento raíz ha cambiado de *Definition* a *Description*
Type	Type	El esquemas XML, que en WSDL 1.1 se podía importar en un número de maneras, para WSDL 2.0 se importan mediante un elemento *xsd:import* como hijo directo del elemento *Type*.
PortType		El elemento *portType* ha sido sustituido por el elemento *Interface* para reflejar mejor su uso.

Message	Interface	El elemento *Message* ya no existe como un elemento global. Descripciones de los mensajes están encapsuladas en el elemento *Interface*.
Binding	Binding	Un *Binding* es ahora reutilizable. No necesita estar asociado a una Interface específica. La asociación se puede hacer en la declaración del elemento *Service*.
Service	Service	

En este trabajo no fue utilizado WSDL 2.0 para describir los servicios RESTful, optamos por ser un tanto conservadores y seguir los principios planteados por Fielding teniendo en cuenta el crecimiento de la web, ya que, algunos conceptos planteados por Roy Fielding en el año 2000 restringen el mundo web actual (hipermedia como el motor de estado de la aplicación).

Frameworks RESTful

Hoy por hoy, han aparecido diferentes marcos que ayudan a la publicación de funciones como servicios web RESTful, como por ejemplo *Java API for RESTful Web Services, JAX-RS* (ORACLE, 2010), entre otros. Este API permite incluir anotaciones @Path, @GET, @POST, @PUT y @DELETE en métodos de clases Java para ser publicados como servicios web .En principio, este tipo de marcos no afectan para nada a la filosofía RESTful y simplemente facilitan el proceso de población de funciones como servicios web.

A lo largo de este capítulo pretendemos explicar un metamodelo RESTful obtenido a partir del estudio, análisis, diseño de componentes, así como, las relaciones entre estos, planteadas por el estilo REST para los servicios RESTful. Este modelo está construido en MOF (OMG, 2006a), tomando algunas consideraciones para cumplir el objetivo principal de este trabajo, la transformación mediante metamodelos de servicios SOAP a servicios *RESTful* y viceversa.

Este capítulo muestra el metamodelo RESTful en elementos Ecore, representación escogida por ser más detallada y compacta que un diagrama MOF.

3.1. Metamodelo REST propuesto

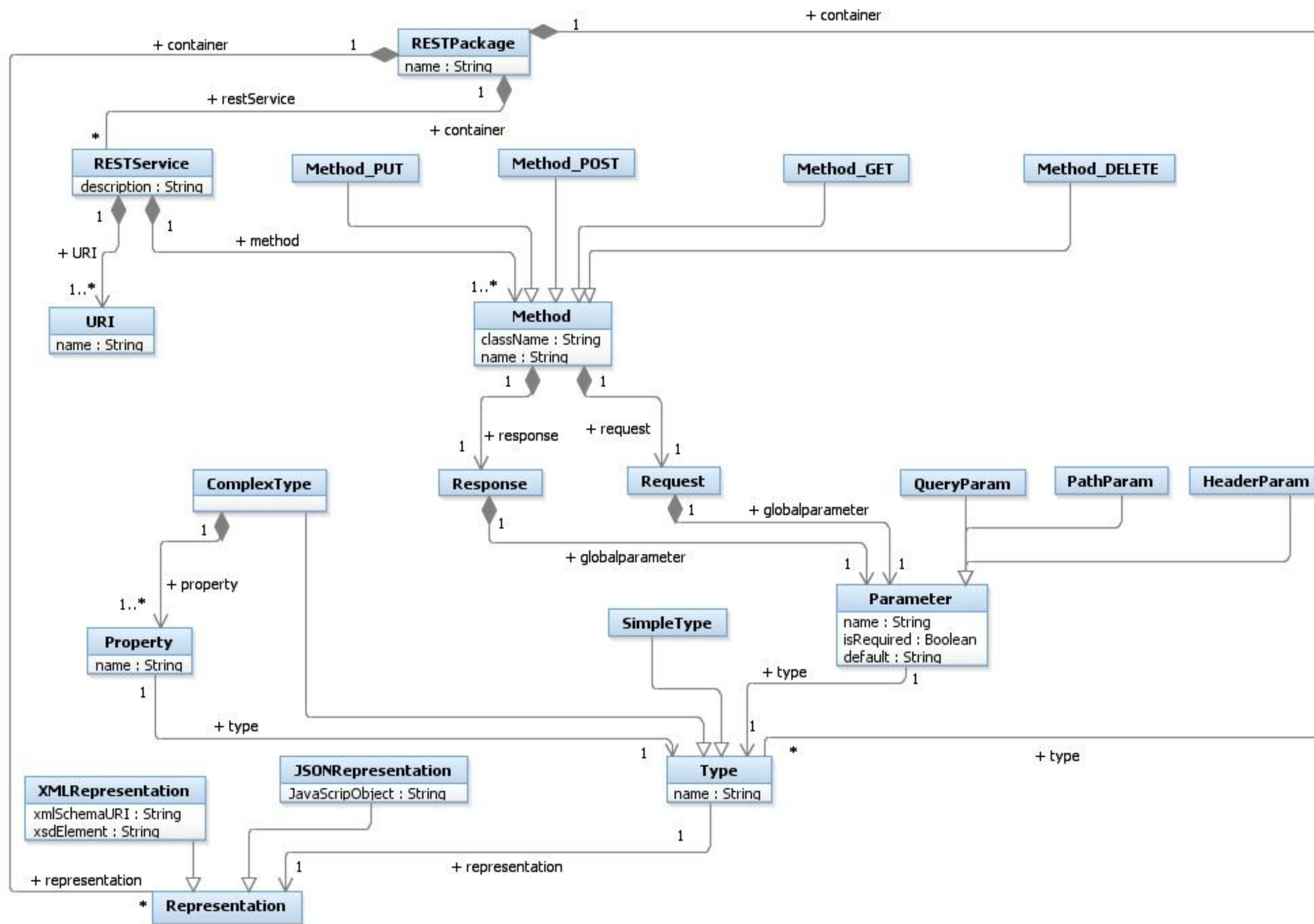

Figura 3.3: Metamodelo *RESTful* propuesto

Esta sección describe nuestra propuesta de metamodelo para caracterizar los servicios web RESTful en el contexto del modelo de conversión propuesto en este trabajo. La Figura 3.3 presenta el metamodelo RESTful propuesto. Sus distintos elementos son analizados en profundidad a continuación.

3.1.1. RESTful. RESTService

Descripción del elemento RESTService

En esta sección se describe el diagrama del elemento raíz "*RESTService*", que contiene el atributos "*description*" que permite ver especificaciones e información que se ha omitido durante la definición del elemento, y se relaciona mediante una composición con el elemento "*URI*", identificador que nos da la localización y nombre del servicio dentro de la web. Esta última clase fue creada pensando en los mecanismos presentes en algunos lenguajes de programación web (p.e. en los servlets) que permiten asociar distintas URLs con el mismo proceso (servicio) que se ejecuta en el servidor.

Un *Uniform Resource Identifier* o *URI* (en español «identificador uniforme de recursos») es una cadena de caracteres corta que identifica *inequívocamente* un recurso. Normalmente estos recursos son accesibles en una red o sistema. Los URI pueden ser localizadores uniformes de recursos (URL), *Uniform Resource Name* (URN), o ambos (Wikipedia, 2013c) .

Un URI consta de las siguientes partes (Wikipedia, 2013c):

- *Esquema*: nombre que se refiere a una especificación para asignar los identificadores, e.g. urn:, tag:, cid:. En algunos casos también identifica el protocolo de acceso al recurso, por ejemplo http:,mailto:, ftp:.
- *Autoridad*: elemento jerárquico que identifica la autoridad de nombres (por ejemplo: //es.wikipedia.org).
- *Ruta*: Información usualmente organizada en forma jerárquica, que identifica al recurso en el ámbito del esquema URI y la autoridad de nombres (p.e. /wiki/Uniform_Resource_Identifier).
- *Consulta*: Información con estructura no jerárquica (usualmente pares "clave=valor") que identifica al recurso en el ámbito del esquema URI y la autoridad de nombres. El comienzo de este componente se indica mediante el carácter '?'.

- *Fragmento*: Permite identificar una parte del recurso principal, o vista de una representación del mismo. El comienzo de este componente se indica mediante el carácter '#'.

  Aunque se acostumbra llamar URL a todas las direcciones web, URI es un identificador más completo y por eso es recomendado su uso en lugar de la expresión URL.

  Un URI se diferencia de un URL en que permite incluir en la dirección una subdirección, determinada por el "fragmento".

  Podemos entender que un URI = URL + URN (Wikipedia, 2013b).

Diagrama del elemento *RESTService*

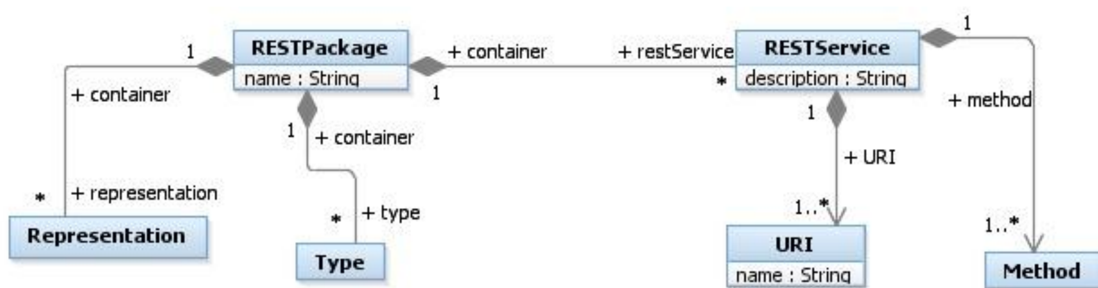

Figura 3.4: Diagrama del elemento *RESTService*

Clases y atributos.

- *Clase RestService:* Representa el elemento raíz del metamodelo RESTful, contiene los métodos HTTP y define la página del servidor (por ejemplo, un *Java Servlet*).
  - *description:* Este atributo permite ver especificaciones e información que se ha omitido durante la definición del servicio.
- *Clase URI:* Representa el identificador base del servicio web. Contiene los datos necesarios para acceder a la información del servicio.
  - *name:* Representa la cadena de caracteres que conforma la dirección de la URI.
- *Clase RESTPackage:* Representa a alto nivel un contenedor de los elementos para las transformaciones QVT.
- *Clase Representation:* Ver epígrafe 3.1.3.
- *Clase Method:* Ver epígrafe 3.1.5.
- *Clase Type:* Ver epígrafe 3.1.2.

- + *method:* Composición que relaciona el elemento RESTService con el conjunto de métodos que permiten exponer el servicio web.
- + *restService, + type,+ representation:* Definen las composiciones que relaciona el contenedor *RESTPackage* con los elementos *RESTService, Type,* y *Representation* consecutivamente.
- + *URI:* Composición que relaciona la clase *RESTService* con la clase *URI.*

3.1.2. RESTful. Type

Descripción del elemento Type

El elemento Type contiene la información referente al tipo (int, string, float, etc) de los parámetros envueltos en la solicitud o respuesta del servicio web.

En este trabajo con el fin de facilitar la legibilidad de los datos a la hora de la conversión *RESTful to SOAP* tomamos como convenio que cada solicitud o respuesta va contener un único parámetro global (*globalParameter*) que será de tipo complejo, su función seria de contenedor (*wrapped*) cuya función será la de agrupar los parámetros de E/S del servicio web. Esta decisión coincide con la filosofía *wrapped* SOAP (descrita en la sección 4.1.5).

Así, por ejemplo, si tenemos un servicio RESTful que efectúa la suma de dos números, en la solicitud tendríamos un único parámetro de tipo complejo denominado sumandos que contendría dos propiedades (los números envueltos en la suma). Las propiedades podrán ser tipadas con un tipo simple o complejo (para nuestro caso, serian dos enteros de tipo simple). En la respuesta tendríamos un único parámetro que tendría como propiedad el resultado de la suma (un número entero).

Diagrama del elemento Type

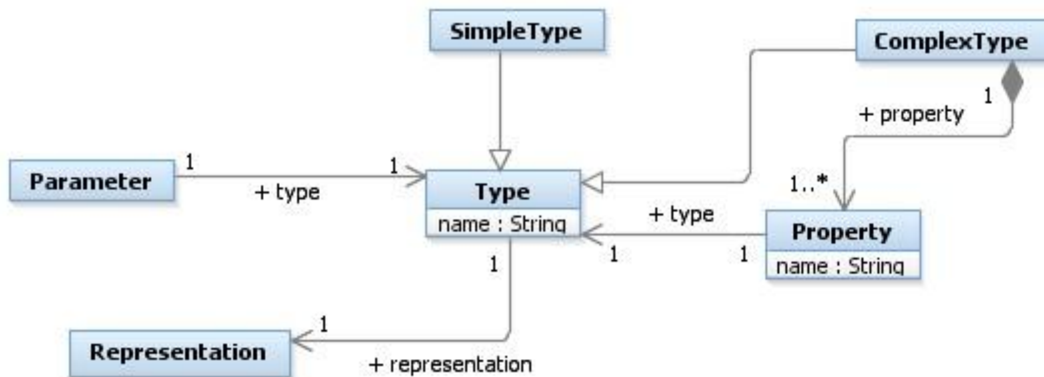

Figura 3.5: Diagrama del elemento *Type*

Clases y atributos.

- *Clase Type:* Define la información referente a los tipos de datos.
  - *name:* Especifica el nombre del tipo de dato a definir.
- *Clase SimpleType:* Representa los tipos simples. Hereda los atributos de su clase progenitora "*Type*".
- *Clase ComplexType:* Representa los tipos complejos. Hereda los atributos de su clase progenitora "*Type*".
  - *+ property:* Esta composición relaciona los tipos complejos con una propiedad o atributo, haciendo posible disponer de diferentes parámetros de tipos simples en una estructura compleja.
- *Clase Property:* Define las propiedades o atributos de un tipo complejo.
  - *name:* Define el nombre de la propiedad.
- *+ type:* Asociación que vincula los parámetros con la información de tipo, permitiendo el tipado de estos.
- *Clase Representation:* Ver epígrafe 3.1.3.
- *Clase Parameter:* Ver epígrafe 3.1.4.

3.1.3. RESTful. Representation

Descripción del elemento Representation

Una representación consta de datos, metadatos que describen los datos, y, en ocasiones, los metadatos para describir los metadatos (por lo general con el propósito de verificar la integridad del mensaje) (Roy Thomas Fielding, 2000).

Según Fielding, el formato de datos de una representación se conoce como *"media_type"*. Las representaciones son procesadas de acuerdo al control de datos, que define el objetivo o propósito entre los elementos de un mensaje, pudiéndose utilizar para parametrizar las solicitudes y anular el comportamiento predeterminado de algunos elementos de conexión (Roy Thomas Fielding, 2000).

Una representación puede ser incluida en un mensaje y procesada por el destinatario de acuerdo con el control de dato del mensaje y de la naturaleza del *media_type*. Algunos *media_type* están destinados para el procesamiento automatizado, otros son utilizados para su visualización por un usuario, y unos pocos son capaces de ambos. *Media_type* compuestos se pueden utilizar para contener múltiples representaciones en un solo mensaje (Roy Thomas Fielding, 2000).

En este trabajo para lograr una legibilidad de los datos a la hora de la conversión hemos tomado como política que los parámetros solo pueden contener una única representación, por tanto, instanciaciones de servicios web *RESTful* con *media_type* compuestos no son permitidos durante la conversión a proponer, por tanto, este atributo no es requerido en el metamodelo propuesto.

Además, hemos restringido las representaciones a XML y JSON para favorecer la conversión automática a objetos, en un contexto de conversión *RESTful to SOAP*, pero, en general, la representación puede ser muy variadas, y los principios de REST no restringe el uso de estas, pudiéndose utilizar cualquier cantidad de formatos estándar para este propósito.

Diagrama del elemento Representation

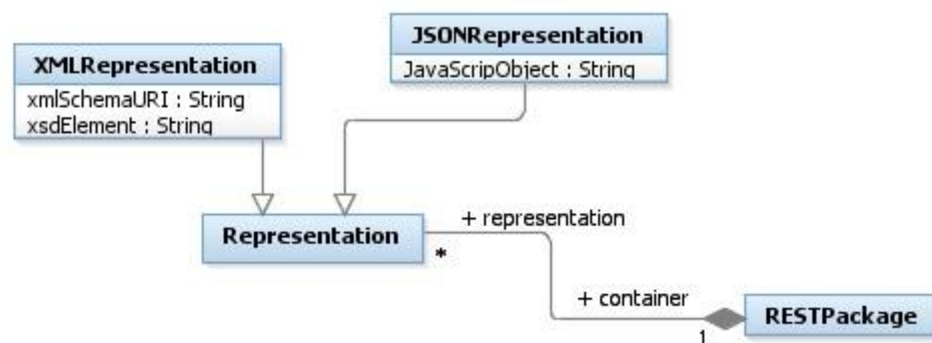

Figura 3.6: Diagrama del elemento *Representation*

Clases y atributos

- *Clase Representation:* Esta clase está definida como una entidad abstracta que es generalizada en los distintos tipos de representaciones a tener en cuenta (para nuestro caso XML y JSON).
- *Clases XMLRepresentation:* Representa el formato XML.
  - *xmlSchemaURI:* Referencia el esquema XML, permitiendo ver la sintaxis a ser utilizada.
  - *xsdElement:* Representa un elemento XML, el cual posibilita a partir del esquema asignar representaciones específicas.
- *Clase JASONRepresentation:* Define el formato JSON.
  - *JavaScripObject:* Representa un objeto en Java.
- *Clase RESTPackage:* Ver epígrafe 3.1.1.

3.1.4. RESTful. Parameter

Descripción del elemento Parameter

Los parámetros en los servicios RESTful pueden ser enviados de varias formas: en el camino como parte del URI (PathParam), en forma de consulta en el URI, después del carácter '?' (QueryParam), o como una cabecera HTTP (HeaderParam).

En este trabajo para facilitar la conversión *RESTful to SOAP* y viceversa hemos asumido que los parámetros serán codificados utilizando el estilo documento (ver epígrafe 4.1.5). Este enfoque obliga la existencia de un elemento contenedor que envuelve el/los parámetros, por tanto, solo un parámetro será definido para cada *Request* y/o *Response*.

Diagrama del elemento Parameter

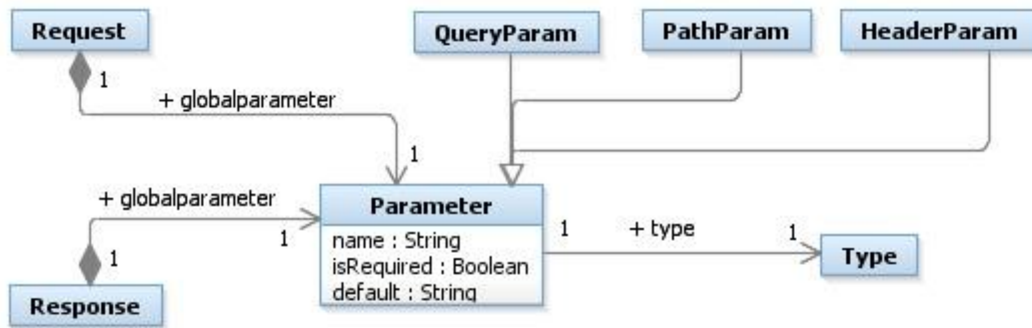

Figura 3.7: Diagrama del elemento *Parameter*

Clases y atributos.

- *Clase Parameter:* Define los parámetros envueltos en el servicio web.
  - *name:* Define el nombre de un parámetro
  - *isRequired:* Define un atributo *booleano* para establecer si es necesario o no un parámetro.
  - *default:* Define cualquier valor por defecto a ser enviado o recibido.
- *Clases (QueryParam, PhatParam, HeaderParam):* Representan los distintos tipos de parámetros según la forma de envío utilizada.
- *Clases Request, Response:* Ver epígrafe 3.1.5.
- *Clase Type:* Ver epígrafe 3.1.2.
- *+ type:* Ver epígrafe 3.1.2.
- *+ globalparameter:* Composición que relaciona el "*Request*" o el "*Response*" con sus parámetros pertinentes.

3.1.5. RESTful. Method

Descripción del elemento Method

En este trabajo hemos decidido llamarle método (*Method*) a las operaciones que permiten manipular un recurso.

Un recurso se define como: cualquier información que pueda ser nombrada, por ejemplo, una imagen, un servicio temporal o una colección de otro recurso (Roy Thomas Fielding, 2000).

Estas operaciones (métodos HTTP) estarían identificadas a través del nombre de la clase que la implementa y de la operación que referenci a o expone; siendo así, nuestro

modelo RESTful, contiene la clase *RESTService* con métodos identificados que de acuerdo a su semántica permiten la manipulación del recurso. REST como estilo arquitectónico utiliza los métodos HTTP para definir sus operaciones y restringe su interfaz a la semántica de estos, conozcamos un poco de esta semántica.

Algunos métodos HTTP son considerados seguros, en caso que la solicitud no modifica el recurso, pueden presentan la propiedad de indepotencia, cuando varias llamadas a la operación no alteran el resultado, o no contener ninguna de las dos propiedades (IBM, 2013a). A continuación se analizan diversos métodos HTTP desde la perspectiva RESTful:

Método HTTP GET: El método GET recupera cualquier información (en forma de una entidad) (R. Fielding). Este método es seguro e indempotente, solicitudes repetidas GET no debe alterar ningún recurso (IBM, 2013a).

Método HTTP PUT: Este método es normalmente usado para actualizar recursos o para crear una nueva entidad en un URL conocida. Este método no es seguro y es indempotente, por tanto, varios pedidos PUT idénticos (la misma identidad en la misma URL) va a tener el mismo resultado, como si fuese solicitado un pedido PUT, asumiendo que ningún pedido relevante fue realizado (IBM, 2013a).

Método HTTP DELETE: Elimina un recurso en un determinado URL. Este método no es seguro y es indempotente (IBM, 2013a).

Método HTTP POST: El método POST se utiliza para que el servidor de origen acepte la entidad incluida en la solicitud como un nuevo subordinado del recurso identificado. POST está diseñado para permitir las siguientes funciones (R. Fielding, 1999):

- Anotación de los recursos existentes.
- Publicar un mensaje en un tablón de anuncios, grupos de noticias, listas de correo, o un grupo similar de artículos.
- Proporcionar un bloque de datos, tales como el resultado de la presentación de un formulario, a un proceso de manejo de datos.
- Ampliación de una base de datos a través de una operación de anexión.

La función real a realizar por el método POST es determina por el servidor y por lo general depende del Request- URI. La entidad publicada está subordinada a la URI de la

misma manera que un archivo está subordinado al directorio que lo contiene, un artículo de noticias subordinado al grupo de noticias donde está publicado, o un registro subordinado a una base de datos (R. Fielding, 1999).

Este método no es seguro y no presenta la propiedad de indempotencia, por tanto si, un administrador emite una solicitud POST a un recurso `/usuarios` que crea un usuario con un identificador único, por ejemplo, *1234567890*. El identificador único se utiliza como parte de la ruta URL para describir el nuevo recurso de usuario, tal como `/users/1234567790`. En este caso, múltiples solicitudes POST a la colección `/usuarios` pueden seguir creando un nuevo identificador único y la adición de este nuevo ID a la colección usuarios, incluso si el usuario dispone de la misma información (IBM, 2013a).

Diagrama del elemento Method

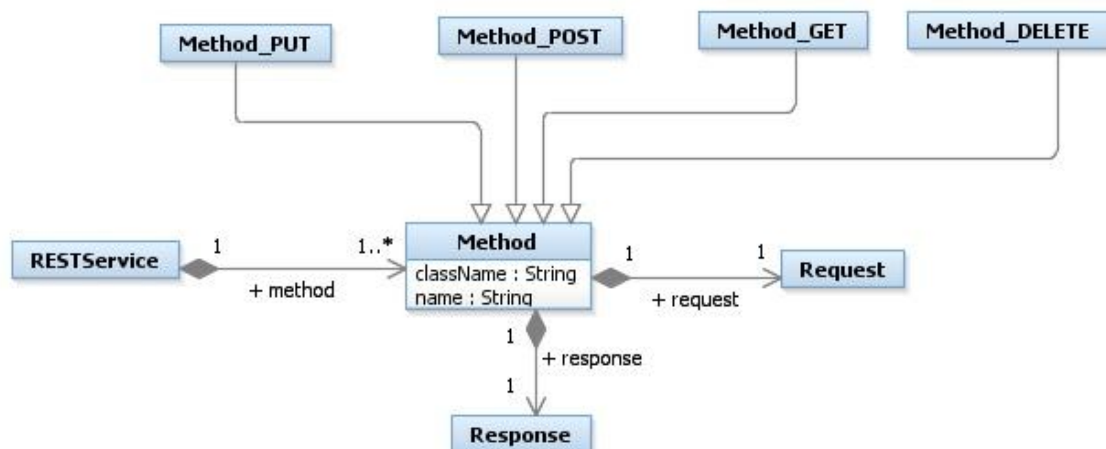

Figura 3.8: Diagrama del elemento *Method*

Clases y atributos

- *Clase Method:* Define la operación utilizada para manipular un recurso o parámetro.
  - *className:* Define el nombre de la clase donde está implementada la operación. Así, esta clase representa el servicio de aplicación (Alur, 2003) que implementa el servicio web. Este atributo no es necesario para la descripción de un servicio RESTful, pero desempeña un papel fundamental en el proceso de conversión (vea capítulo 5).
  - *name:* Define el nombre de la operación que se desea acceder.

- *Clases Method (GET, POST, PUT, DELETE):* Representan los distintos métodos HTTP.
- *Clase Request* y *Response:* Definen los encabezados HTTP.
- *Clase RESTService:* Ver epígrafe 3.1.1.

3.2. Ejemplo de Instanciación del Metamodelo RESTful

Para la instanciación del metamodelo RESTful hemos ejemplificado una interfaz para la gestión de productos. Esta interfaz está definida mediante las operaciones CRUD (*Create, Read, Update* y *Delete*) que permiten aceptar y devolver identificadores, así como elementos de transferencia que caracterizan a los productos.

Cada una de estas operaciones (GET, POST, PUT y DELETE) contienen parámetros que pueden ser de entrada o de salida (*return*) según al escenario que pertenecen (*Request* o *Response* respectivamente).

Se declara un tipo complejo (*TransferProduct*) que define la transferencia de un producto, este contiene tres propiedades: un identificador (id), el nombre del producto (*name*) y el precio (*price*). A su vez se definen los tipos simples que caracterizan estas propiedades y la representación del formato de datos. Para facilitar la comprensión del ejemplo hemos escogido la representación XML.

La figura 3.8 muestra una instancia para este ejemplo en términos de elemento *Ecore*, eligiéndose esta representación por ser más detallada y compacta que un diagrama MOF.

3.2.1. Instancia del metamodelo RESTful

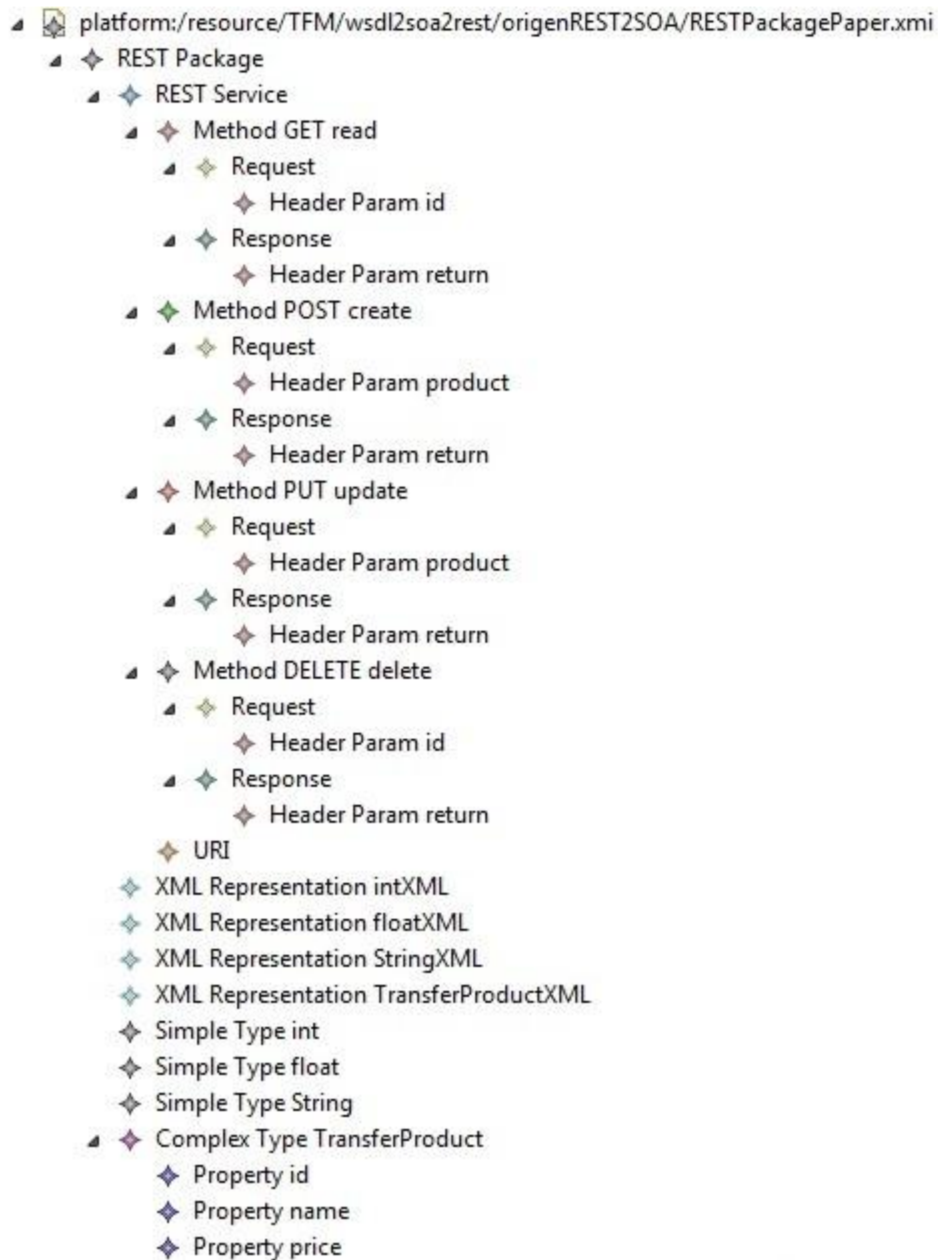

Figura 3.9: Instancia del metamodelo RESTful

4. WSDL: Web Services Description Language

Los servicios web SOAP, utilizan SOAP como protocolo de comunicación y WSDL como un tipo de contrato o acuerdo que especifica la sintaxis y los mecanismos de intercambio de mensajes

SOAP *(Protocolo Simple de Acceso a Objetos)* es un formato de mensaje XML utilizado en las interacciones de servicios Web. Los mensajes SOAP se envían normalmente a través de HTTP o JMS, pudiéndose utilizar otros protocolos de transporte. El uso de SOAP en un servicio Web específico es descrito por una definición WSDL (Onstine, 2011).

Una de las funcionalidades con las que cuenta SOAP y que lo hace extremadamente atractivo es el envío de mensajes vía protocolo HTTP, todo ello realizado de forma directa por medio de las librerías de SOAP.

Este hecho provoca que existen ciertas diferencias entre un mensaje SOAP normal y uno enviado haciendo uso del protocolo HTTP. De esta forma debe cumplirse que, el HTTP *header* del mensaje de petición al servicio Web debe contener un campo *SOAPAction*. El campo *SOAPAction* alerta a los servidores Web y firewalls por los que pase de que el mensaje contiene documento SOAP, esto facilita a dichos firewalls el filtrado de las peticiones SOAP sin necesidad de tener que explorar el contenido *body* del mensaje.

WSDL (*Web Services Description Language*) el lenguaje de descripción de servicios web, proporciona una gramática para describir servicios como un conjunto de extremos que intercambian mensajes. Un documento WSDL sirve como una descripción independiente del lenguaje y de la plataforma (XML) de uno o más servicios. Describe los servicios, la manera de acceder a ellos y qué tipo de respuesta (si hubiera) se debe esperar. Un documento WSDL puede ser intercambiado de manera privada o asentado en un registro UDDI (público o privado) para permitir un acceso más amplio. En realidad, se podría exponer una interfaz 100% basada en Java, documentación basada en texto o cualquier otra interfaz para el servicio. Este lenguaje ofrece un formato de interfaz estandarizado y neutra (IBM, 2012).

Se podría resumir que WSDL proporciona el contrato de un servicio entre la entidad que lo solicita y la que lo proporciona, de forma que usando el WSDL, un cliente puede localizar un servicio web e invocar cualquiera de sus funciones de forma correcta.

No habiendo un modelo para los servicios SOAP presentado por el W3C (consorcio internacional que produce recomendaciones para la *World Wide Web*), hemos utilizado WSDL como fuente esencial para construir nuestro modelo SOAP, siendo este el objetivo principal del capítulo. La relación entre WSDL, SOAP y HTTP forma parte del modelo propuesto y la instanciación de este nos permite ver mediante un ejemplo la relación entre los conceptos.

A pesar de existir en la actualidad dos especificaciones SOAP, la versión 1.1 y la versión 2.0, hemos optado por proporcionar un metamodelo para la versión 1.1 por tener mayor difusión (Hansen, 2007). No obstante, WSDL 1.1 sigue siendo un estándar demasiado complicado (Hansen, 2007), por eso, hemos tenido en cuenta simplificaciones, como las consideras en *WS-I Basic Profile* (WS-I, 2006), para simplificar el metamodelo.

4.1. Metamodelo SOAP propuesto

Figura 4.1: Metamodelo SOAP propuesto

Esta sección describe nuestra propuesta de metamodelo para caracterizar los servicios web SOAP en el contexto del modelo de conversión propuesto en este trabajo. La Figura 4.1 presenta el metamodelo SOAP propuesto. Sus distintos elementos son analizados en profundidad a continuación.

4.1.1. WSDL. WSDL Definition

Descripción General de los elementos.

Para la construcción del modelo WSDL hemos tenido en cuenta sus componentes y la relación entre ellos.

La figura 4.2 muestra elemento raíz "*WSDL Definition*", y contiene dos atributos "*DefinitionName*" y "*TargetNameSpace*", que caracterizan los espacios de nombres. Dichos espacios se utilizan para evitar conflictos de nombres cuando varios servicios o aplicaciones están integrados (Juan M. Vara, 2005).

La clase "*WSDL Definition*" está asociada con los siguientes componentes: "*Types*", *"Messages", "Port Type", "Binding"*, "*Service*" *y WSDLPackage*. Todos los componentes WSDL pueden estar asociados a un atributo *documentation* (documentación), que permite ver especificaciones e información que se ha omitido durante la definición.

El componente "*Type*" tiene como función definir los tipos de datos presente en los mensajes, utilizando el esquema XML para lograr este propósito.

Para describir las operaciones del servicio nos apoyamos del componente *"PortType"*. Cada operación está asociada al componente "*Message*".

El componente *"Binding"* describe el estilo del mensaje y el protocolo de transmisión (SOAP, HTTP o MIME) que permite el enlace entre el componente "*PortType*" y las operaciones asociadas a un formato de mensaje.

"Service", tiene como función describir el conjunto de puertos que proporciona un servicio y por último tenemos el componente *WSDLPackage* cuya función es agrupar los elementos necesarios para las transformaciones QVT.

Durante este capítulo estudiaremos con más profundidad cada uno de estos elementos, así como los elementos que lo componen.

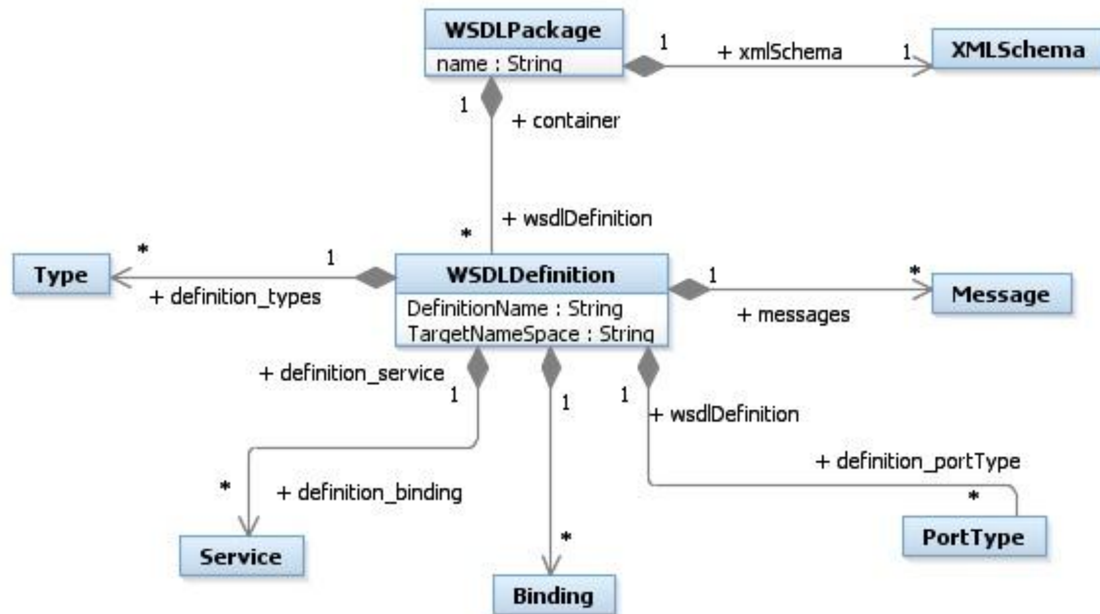

Figura 4.2: Diagrama del elemento raíz de la definición WSDL (corregir XMLSchema)

4.1.2. WSDL. Type

Descripción del elemento Type

El elemento *Type* encapsula definiciones de tipos de datos que son relevantes en el intercambio de mensaje. Para obtener la máxima interoperabilidad y neutralidad de la plataforma, WSDL prefiere el uso de XSD como el sistema de tipo canónico. El lenguaje de definición de esquemas XML (XSD) proporciona un sistema de tipos primitivo para entornos de procesamiento XML, dando lugar al esquema XML que es utilizado en la definición de tipos de datos en WSDL. En resumen, el esquema XML permite describir los tipos a ser utilizados (Skonnard, 2003b).

En la definición de WSDL, el elemento *Type* funciona como contenedor para las definiciones de Tipo del esquema XML. Estas definiciones hacen referencia a las definiciones de mensajes a alto nivel, con el fin de definir los detalles estructurales del mensaje. Oficialmente, WSDL 1.1 permite el uso de cualquier lenguaje de definición de tipo, aunque se recomienda el uso de esquemas XML como sistema de tipo específico. WS-I hace cumplir este requisito al requerir el uso de esquemas XML en el Perfil Básico 1.0.

La estructura básica del elemento *Type* (con espacios de nombres omitidos) es como sigue (* significa cero o más) (Skonnard, 2003a):

```
<definitions .... >
    <types>
        <xsd:schema .... />*
    </types>
</definitions>
```

Se puede utilizar cualquier constructor del esquema XML dentro del elemento *schema*, por ejemplo, definiciones de tipo simple, complejo, y elementos. Sin embargo, dado que es razonable esperar que una gramática de tipo pueda ser utilizada para describir todos los tipos abstractos presentes y futuros, WSDL permite la extensión de sus sistemas de tipo a través de elementos de extensibilidad.

Tipos por extensión.

En caso que se desee manipular otro sistema de tipo aparecerá un elemento de extensibilidad bajo el elemento *Type*. El nombre de este elemento debería identificar el sistema de tipos utilizados, y su función puede ser comparada con el propósito de la clase *Schema* del esquema XML (W3C, 2001)

```
<definitions .... >
    <types>
        <-- type-system extensibility element --> *
    </types>
</definitions>
```

Los elementos de extensibilidad se utilizan para representar determinadas tecnologías. Por ejemplo, son útiles para especificar el idioma que se utiliza en el esquema de los elementos *Types*. El esquema para un determinado conjunto de elementos de extensibilidad se debe definir dentro de distintos espacios de nombres del documento WSDL. La definición de los propios elementos puede contener un atributo *wsdl:requiered* que indique un valor *boolean,* si el atributo *requiered* se establece a *true* en una definición de elementos, una asociación que haga referencia a ese conjunto concreto de electos de extensibilidad tiene que incluir dicho elemento (González., 2004).

Diagrama del elemento Type

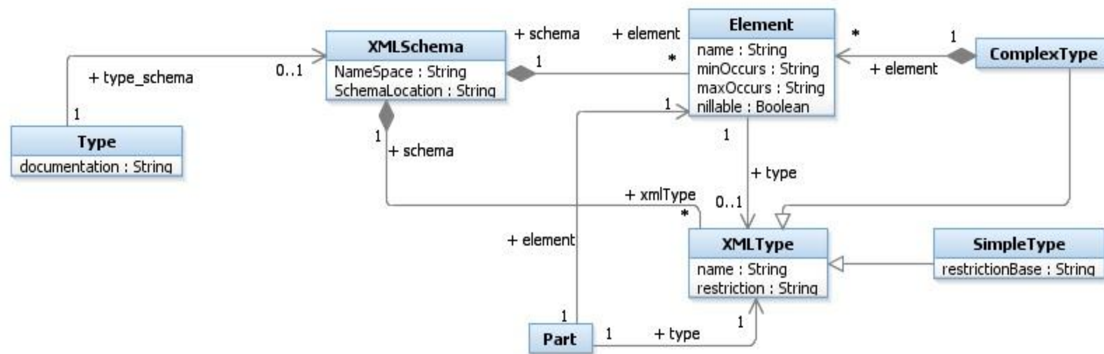

Figura 4.3: Diagrama del elemento "*Type"* de la definición WSDL (XMLSchema)

Clases y atributos.

- *Clase Type:* Esta clase permite la definición de tipos dentro del metamodelo SOAP.
    - *documentation*: Este atributo permite ver especificaciones e información que se ha omitido durante la definición del elemento y es permitido dentro de cualquier elemento del lenguaje WSDL
- *+ type_schema:* Esta asociación representa el vínculo entre el tipo de dato y el esquema XML.
- *Clase XMLSchema: Esta clase representa el esquema XML.*
    - *NameSpace:* Define el espacio de nombre de los elemento del esquema.
    - *SchemaLocation*: Representa la localización del esquema XML que será usado por el espacio de nombre definido.
- *+ schema…+ element:* Composición que permite definir los elementos del esquema XML.
- *Clase Element:* Representa un elemento dentro del esquema XML.
    - *name*: Representa el nombre del elemento.
    - *minOccurs y maxOccurs*: Estos atributos indican el número mínimo y máximo de veces que un elemento puede aparecer. Este valor puede ser un entero positivo tal como el 41, o el plazo ilimitado para indicar que no hay un número máximo de ocurrencias. El valor por defecto para ambos *minOccurs* y *maxOccurs* es 1 (W3C, 2004).
    - *nillable*: Este atributo representa el valor nulo, en ocasiones es conveniente para representar información desconocida de un elemento.

Por ejemplo, para representar el valor nulo de un elemento a ser enviado desde una base de datos relacional. El mecanismo para expresar valor nulo del esquema XML consiste en un fuera de banda o señal nula. En otras palabras, el valor real cero (0) no aparece como contenido del elemento, en su lugar hay un atributo para indicar que el contenido del elemento es nulo (W3C, 2004).

- *Clase XMLType:* Esta clase representa la información de tipo del esquema XML.
  - *name:* Especifica el nombre del tipo a definir.
  - *restriction:* Este atributo especifica exactamente cómo se desea restringir el tipo base, limitando una o varias de sus definiciones.
- *Clase SimpleType:* Representa los tipos simples en la definición de datos del esquema XML. Hereda los atributos de la clase progenitora "*XML Type*".
  - *restrictionBase:* Especifica un tipo base XSD cuyo espacio de valor se desea restringir, para la definición del nuevo tipo simple.
- *Clase ComplexType:* Representa los tipos complejos dentro de la definición de datos del esquema XML. Hereda los atributos de la clase progenitora "*XML Type*".
- + *element:* Esta composición relaciona los tipos complejos con un Elemento, haciendo posible disponer de diferentes elementos de tipos simples en una estructura, también conocido como un tipo complejo.
- *Clase Part:* Ver epígrafe 4.1.3.
- + *schema…+ element*: Relaciona el lemento con el esquema XML.

4.1.3. WSDL. Message

Descripción del elemento Message

El elemento de mensaje WSDL define un mensaje de resumen que puede servir como la entrada o salida de una operación. Es decir, agrupa los parámetros de dichas operaciones. Los mensajes consisten en uno o más elementos *"Part"* (parámetros), donde cada parte está asociada, con un elemento XML (cuando se utiliza un estilo de binding document) o un tipo XML (cuando se utiliza un estilo de binding RPC). Para más información sobre estos estilos ver epígrafe 4.1.5.1.

Pude aparecer, y normalmente aparecerán, múltiples elementos *Message* en un documento WSDL, definiendo las operaciones expuestas en cada servicio web. Un

ejemplo de una parte es el cuerpo de un mensaje SOAP, un parámetro que forma parte de una cadena de petición, un parámetro codificado en el cuerpo del mensaje SOAP o todo el cuerpo de un mensaje SOAP.

Diagrama del elemento Message

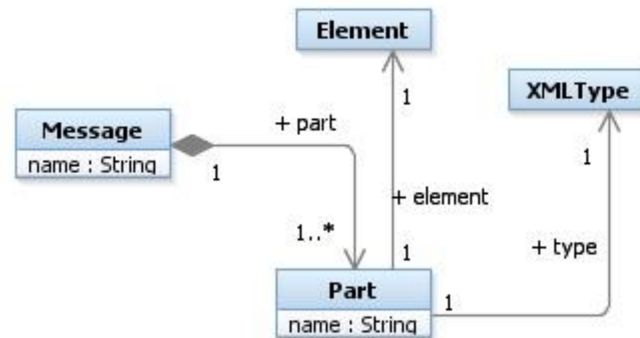

Figura 4.4: Diagrama del elemento "*Message*" de la definición WSDL

Clases y atributos.

- *Clase Message:* Define el mensaje dentro de la especificación WSDL.
  - *name*: Especifica el nombre del mensaje.
- + *part*: Vincula el mensaje al elemento *Part* mediante una composición.
- *Clase Part*: Representa el elemento parte.
  - *name*: Los mensajes y las partes deben ser nombrados de tal manera que sea posible referencia a ambos en la definición WSDL.
- + *element*: Esta asociación vincula la parte a un elemento, teniendo sentido cuando se utiliza estilo de documento (ver sección 4.1.5 para más detalles).
- + *type*: Representa el vinculo existente entre el elemento *Part* y *Types*, este asociación solo tiene sentido si se utiliza el estilo *RPC* (ver sección 4.1.5 para más detalles).
- *Clases XMLType* y *Element.* Ver especificación en el epígrafe 4.1.2.

4.1.4. WSDL. Port Type

Descripción del elemento Port Type

El elemento *PortType* también conocido como interfaz en la mayoría de los entornos, define un grupo de operaciones. Desafortunadamente, el término "*PortType*" es bastante confuso, WSDL 1.2 ya ha eliminado el término "*PortType*" sustituyéndolo por "interfaz" (Skonnard, 2003a).

Un elemento *PortType* contiene cero o más *operation*, que son las operaciones definidas por el interfaz del servicio web. La estructura básica de un *PortType* es la siguiente (* significa cero o más) (Skonnard, 2003a).

```
<definitions .... >
    <portType name="nmtoken">
        <operation name="nmtoken" .... /> *
    </portType>
</definitions>
```

Cada portType debe tener un nombre único por lo que es posible hacer referencia desde otra parte de la definición WSDL.

Cada elemento de la operación contiene una combinación de elementos de entrada y de salida, y cuando se tiene un elemento de salida también puede tener un elemento de error. El orden de estos elementos define el patrón de intercambio de mensajes, por sus siglas en inglés MEP (*message exchange pattern*), este patrón es apoyado por operaciones que se definen según su comportamiento, por ejemplo, un elemento de entrada, seguido de un elemento de salida define una operación de petición-respuesta (*request-response*), mientras que un elemento de salida seguido de un elemento de entrada define una operación de petición-respuesta (*solicit-response*). Una operación que sólo contiene un elemento de entrada define una operación de un solo sentido (*one-way*), mientras que una operación que sólo contiene un elemento de salida (*notification*) define una operación de notificación. La tabla 4.1 describe las cuatro primitivas MEP definidas por WSDL.

Tabla 4.1: Patrón de intercambio de mensajes (MEP)

MEP	Descripción
Request-response	El cliente realiza una petición y el servidor envía la correspondiente respuesta.
Solicit-response	El servidor envía una petición y el cliente envía una respuesta.
One-way	El cliente envía un mensaje pero no recibe una respuesta del servidor indicando el resultado del mensaje procesado.
Notification	El servidor envía un mensaje pero no recibe respuesta del cliente indicando el resultado del mensaje.

En este trabajo hemos utilizado el estilo petición-respuesta (*request-response*) para el modelado de las operaciones que componen este elemento. La definición de *portType* o

interfaz aún se considera abstracta, porque no sabemos cómo los mensajes se representan hasta que se aplique un *binding* (Ver sección 4.1.5).

Diagrama del elemento Port Type

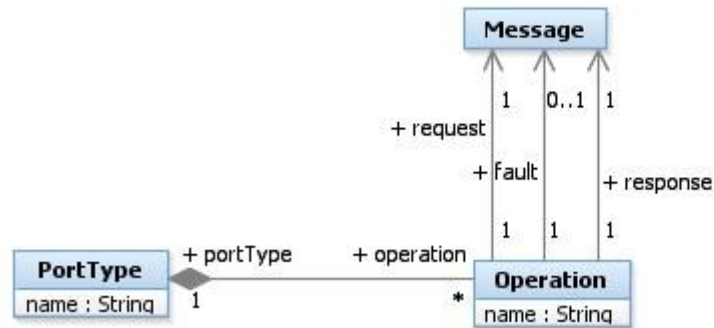

Figura 4.5: Diagrama del elemento "*PortType*" de la definición WSDL

Clases y atributos.

- *Clase PortType:* Define la interfaz dentro de la defición WSDL.
  - *name:* Especifica el nombre del PortType, este debe ser único, para poder ser especificado desde cualquier parte de la deficinición WSDL
- *+ portType…+ operation:* Relación existente entre los elementos *PorType* y *Operation*, esta relación está dada mediante una composición.
- *+ request, + response, + fault:* Definen el MEP de tipo *request-response.*
- *Clase Operation:* Esta clase representa el elemento *Operation*.
  - *name:* Especifica el nombre de la operación.
- *Clase Message:* Ver especificación en el epígrafe 4.1.3.

4.1.5. WSDL. Binding

Descripción del elemento Binding

El elemento *binding* describe los detalles concretos de la utilización de un *PortType* específico con un protocolo dado. Conteniendo varios elementos de extensibilidad que brindan detalles de su implementación, así como un elemento *operation* para cada operación del *PortType* descrito. La estructura básica del elemento *binding* es la siguiente (Skonnard, 2003a):

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element providing binding details --> *
        <wsdl:operation name="nmtoken"> *
           <-- extensibility element for operation details --> *
```

```
            <wsdl:input name="nmtoken"? > ?
                <-- extensibility element for body details -->
            </wsdl:input>
            <wsdl:output name="nmtoken"? > ?
                <-- extensibility element for body details -->
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <-- extensibility element for body details -->
            </wsdl:fault>
         </wsdl:operation>
     </wsdl:binding>
</wsdl:definitions>
```

binding es un elemento genérico, se utiliza como un *framewok* para definir los detalles de un enlace en WSDL. Los enlaces reales son definidos mediante los elementos de extensibilidad que lo componen. Esta arquitectura permite WSDL evolucionar en el tiempo, ya que cualquier elemento puede ser utilizado dentro de las etiquetas predefinidas, por ejemplo, la definición WSDL proporciona algunos elementos *binding* para describir enlaces SOAP, a pesar de estar en un espacio de nombres diferente. El siguiente ejemplo ilustra un *SOAP / HTTP binding* para el *portType MathInterface* (Skonnard, 2003a):

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
 ...
   <binding name="MathSoapHttpBinding" type="y:MathInterface">
      <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="Add">
         <soap:operation
                   soapAction="http://example.org/math/#Add"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
          ...
   </binding>
   ...
</definitions>
```

Este elemento también indica el estilo predeterminado del servicio (valores posibles incluyen documento o rpc) junto con el protocolo de transporte requerido (en este caso HTTP). El elemento *soap: operation* define el valor de encabezado *SOAPAction HTTP* para cada operación. Y el elemento *soap:body* define cómo las partes del mensaje aparecen en el interior del elemento *Body* de SOAP (valores posibles incluyen literal o

codificado). Hay otros detalles específicos del *binding* que pueden ser especificados de esta manera.
El atributo "*use*" especifica el uso de la codificación que se debe utilizar para traducir las partes de mensaje en una representación concreta. En el caso de *'encoded'*, es necesario reglas de codificación de SOAP para que las definiciones abstractas se conviertan en un formato concreto. En el caso de 'literal', las propias definiciones de tipo abstractas se convierten en definiciones concretas (son "literal" definiciones). Esto quiere decir que basta examinar las definiciones de esquema XML de tipo para determinar el formato de mensaje concreto.

Utilizando las definiciones literales el procesamiento del mensaje se hace simple, rápido y confiable. Para las definiciones rpc el uso de las reglas de codificación ha llevado a problemas de interoperabilidad significativas entre los conjuntos de herramientas. También conduce a situaciones extrañas como no ser capaz de validar el mensaje contra la definición del esquema original (ya que es abstracto y no es una representación fiel del mensaje). Para ayudar a aliviar la confusión y facilitar una mejor interoperabilidad, WS-I prohíbe el uso de codificaciones (*encode*), por tanto los estilos */user, rpc/encode* y *document/encode* no sonconsiderados en este trabajo.

Cabe destacar que aunque un mismo *portType* puede estar en distintos bindings, de facto, el tipo de *binding* limita el formato de los *messages* del *portType*. Así, no se puede definir un *message* en estilo RPC literal, sin hacer mención a elementos XML, y luego intentar hacer un *binding* del *portType* del estilo *document literal*. Por tanto, los distintos tipos de *binding* hacen mención a otras cuestiones (p.e. formato SOAP 1.1 o SOAP 1.2) distintas del posible estilo de *binding*. Además, aunque los elementos soap del *binding* son, en principio, independientes de los elementos *message wsdl,* debe haber una concordancia entre los elementos *wsdl message* y el *soap binding* correspondiente.

4.1.5.1. Binding. Style/User.

WSDL 1.1 se distinguen dos estilos del mensaje: documento y RPC. Estos estilos afectan el contenido de <soap:Body> de la siguiente manera (Shohoud, 2003):

- Documento (document): <soap:Body> contiene uno o más elementos secundarios denominados partes. No hay reglas de formato SOAP por lo que el

<soap:Body> contiene, por tanto contiene todo lo que el emisor y el receptor estén de acuerdo.

- RPC (rpc): Esto significa que el cuerpo de la solicitud SOAP (soap:Body) se compone de un elemento contenedor (wrapped) XML que indica el nombre de la función. Los parámetros de la función son luego incorporados dentro del elemento. Del mismo modo, el soap:Body de la respuesta incluirá un elemento XML que envuelve y refleja la petición de la función y los valores de retorno son luego incorporados dentro del elemento contenedor respuesta (Kamerkar, 2012).

Document/Literal y Document/Literal wrapped

Cada mensaje contiene cero o muchas partes. Parte (part) es una declaración de un elemento que describe el contenido del cuerpo del mensaje, es decir describe los parámetros de las operaciones.

Dentro del estilo Document literal existe un sub-estilo denominado Document /literal wrapped esta subconjunto del estilo document/literal permite agrupar los elementos del mensaje de tal manera que, un mensaje solo puede contener una parte (que solo puede pertenecer a un mensaje), permitiendo en ocasiones una mayor legibilidad del esquema.

En la declaración del enlace SOAP, este estilo se representa como "document" y el uso es "literal".

La característica principal del documento / literal, y su principal ventaja en comparación con RPC / literal, es el uso de una declaración de elemento de esquema para describir completamente el contenido de *soap: body*. Esto significa que el conjunto de información del cuerpo del mensaje es totalmente legible y puede ser entendido con tan sólo mirar el esquema y sin necesidad de normas adicionales. Por lo tanto, podemos tomar el esquema que describe un mensaje de documento / literal para validar el mensaje. Esto no es posible con rpc / literal.

RPC/Literal

Las características principales de este estilo es que cada mensaje contiene cero o más partes y en la declaración del *binding* SOAP este estilo se representa como *"rpc"* y el uso es "literal".

La característica principal (si se puede llamar así) de RPC / literal, y su principal desventaja en comparación con el documento / literal, es que no hay un mecanismo formalmente definido para saber qué contiene *soap: body*. Esto significa que el esquema

solo no nos dice cuál es el conjunto de información contenida en el cuerpo del mensaje, también debe conocerse las reglas RPC. Por lo tanto, el esquema que describe un mensaje de RPC / literal no es suficiente para validar el mensaje.

Así, podemos tener los distintos tipos de puertos, en base al estilo de binding (Hansen, 2007):

- **rpc literal wrapped**

```
<wsdl:message name="request">
      <wsdl:part name="startDate" type="xs:date"/>
      <wsdl:part name="endDate" type="xs:date"/>
</wsdl:message>
<wsdl:message name="response">
      <wsdl:part name="orders" element="oms:getOrdersType"/>
</wsdl:message>
```

- **document literal**

```
<wsdl:message name="request">
      <wsdl:part name="parameter1" element="getord:startDate"/>
      <wsdl:part name="parameter2" element="getord:endDate"/>
</wsdl:message>
<wsdl:message name="response">
      <wsdl:part name="parameter1" element="getord:orders"/>
</wsdl:message>
```

- **document literal wrapped**

```
<wsdl:message name="request">
      <wsdl:part name="parameters" element="getord:getOrdersDates"/>
</wsdl:message>
<wsdl:message name="response">
      <wsdl:part name="parameters"
element="getord:getOrdersDatesResponse"/>
</wsdl:message>

  <xs:schema elementFormDefault="qualified"
      targetNamespace="http://www.example.com/getord">
      <xs:element name="getOrdersDates">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="startDate" type="xs:date"/>
            <xs:element name="endDate" type="xs:date"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="getOrdersDatesResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="orders" type="oms:OrdersType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
```

Los mensajes SOAP que se corresponden con estos estilos de binding son:

- **rpc literal wrapped**

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Body>
      <getOrdersDates>
```

```
        <startDate>2005-11-19</startDate>
        <endDate>2005-11-22</endDate>
      </getOrdersDates>
    </env:Body>
</env:Envelope>
```

- **document literal**

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Body xmlns:getord="http://www.example.com/oms/getorders">
      <getord:startDate>2005-11-19</getord:startDate>
      <getord:endDate>2005-11-22</getord:endDate>
    </env:Body>
</env:Envelope>
```

- **document literal wrapped**

```
<env1:Envelope xmlns:env1="http://schemas.xmlsoap.org/soap/envelope">
    <env1:Body>
      <getord:getOrdersDates
xmlns:getord="http://www.example.com/oms/getorders">
        <getord:startDate>2005-11-19</getord:startDate>
        <getord:endDate>2005-11-22</getord:endDate>
      </getord:getOrdersDates>
    </env1:Body>
 </env1:Envelope>
```

Diagrama del elemento Binding

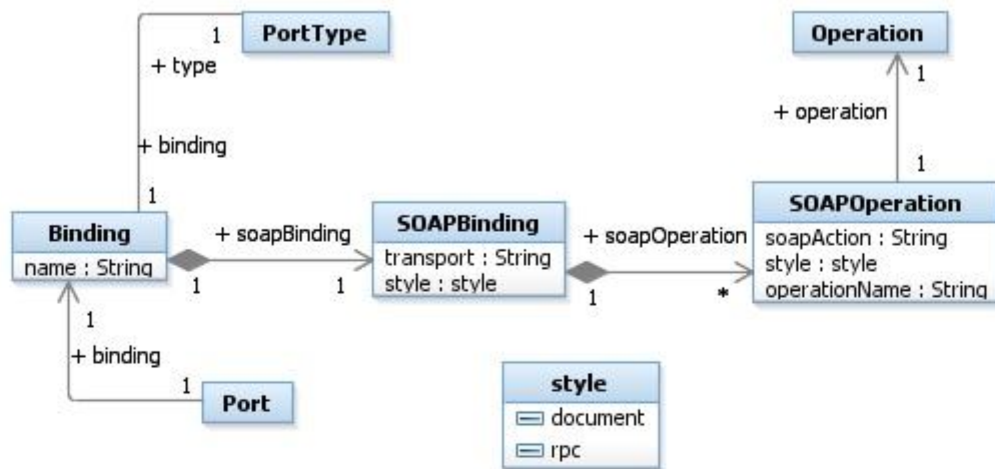

Figura 4.6: Diagrama del elemento *"Binding"* de la definición WSDL (SOAPBinding, SOAPOperation).

Clases y atributos.

- Clase Binding: Representa el componente Binding en la definición WSDL.
  - *name*: Define el nombre del *binding,* este debe ser único para que pueda ser identificado desde cualquier parte de la definición WSDL.
- *Clase SOAPBinding:* Define la información del enlace para el intercambio de mensajes SOAP.

    - *transport:* Indica el protocolo de transporte a utilizar para el intercambio de los mensajes SOAP
    - *style:* Especifica el estilo a utilizar en los mensajes SOAP (dos valores posibles “literal” o “rpc”)
- *Clase style:* Hace referencia a los posibles estilos a tener en cuenta (*document* y *rpc*).
- *Clase Port*: Ver especificación en el epígrafe 4.1.6.
- *Clase PortType*: Ver especificación en el epígrafe 4.1.4.
- *Clase Operation*: Ver especificación en el epígrafe 4.1.4
- *Clase SOAPOperation:* Define las operaciones (*input* y *output*) para el protocolo SOAP.
    - *soapAction:* Este atributo define el valor del encabezado HTTP para cada operación. Se requiere para SOAP 1.1 sobre HTTP binding y no es necesario para otros medios de transporte (Yang, 2013).

      El perfil básico de *Web Services Interoperability Organization (WS-I)* establece que el *soapAction* se debe utilizar con un valor fijo (cadena vacía) (Bayer, 2009).
    - *style:* Define el estilo de la operación SOAP, definidas en el *soapbody.*
    - *operationName:* Define el nombre de la operación SOAP definidas en el *SOAPOperation.*
- + *operation:* Asocia las operaciones SOAP (*SOAPOperation)* con el elemento *Operation* de la definición WSDL.
- + *type…*+ *binding*: Esta asociación especifica el *PortType* que se está describiendo en el *binding.*
- + *soapOperation:* Esta asociación vincula los elementos *SOAPBinding y SOAPOperation.*
- + *soapBinding:* Esta composición vincula el *Binding* con el *soapBinding* correspondiente.
- + *binding:* Ver especificación en el epígrafe 4.1.5.

4.1.6. WSDL. Service

Descripción del elemento Service

El elemento *Service* define una colección de puertos o puntos finales (*endpoint*), que exponen un *binding* determinado. La estructura básica del elemento de servicio es como sigue (Skonnard, 2003a):

```
<definitions .... >
    <service .... > *
        <port name="nmtoken" binding="qname"> *
           <-- extensibility element defines address details -->
        </port>
    </service>
</definitions>
```

Cada puerto contiene nombre, un *binding* y un elemento de extensibilidad para definir los detalles del *binding* al cual está asociado. Los puertos definidos en determinado servicio son independientes, por ejemplo, la salida de un puerto no puede utilizarse como una entrada de otro.

En este trabajo hemos agregado el atributo *className* al elemento *Port*, haciendo referencia a la clase que implementa las operaciones del *Binding*. Este atributo es necesario para vincular el interfaz con su clase de implementación (Hansen, 2007). Esta información por lo general, es transparente al usuario y en nuestro caso es fundamental para lograr instanciar el metamodelo SOAP en un metamodelo de conversión SOA (vea capítulo 5), y así, lograr el objetivo general de este trabajo, convertir servicios web descritos en WSDL a REST y viceversa.

Diagrama del elemento Service.

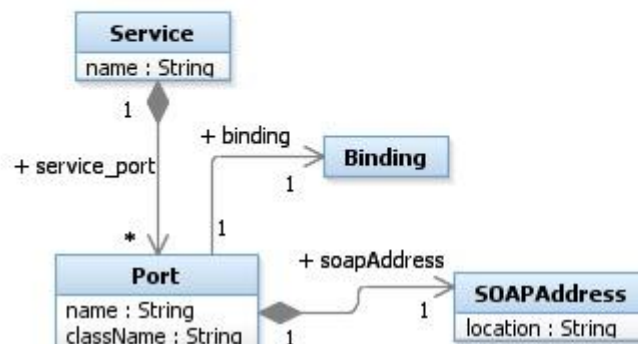

Figura 4.7: Diagrama del elemento *Service* de la definición WSDL

Clase y Atributos.

- *Clase Service:* Define el elemento *Service* de la definición WSDL
  - *name:* Define el nombre del servicio.
- *Clase Port:* Representa el elemento *Port* en la definición *WSDL*
  - *name:* Define el nombre del elemento *Port*.
  - *className:* Define el nombre de clase que implementa el *Binding*.
- *Clase Binding:* Ver especificación en el epígrafe 4.1.5.
- *Clase SOAPAddress:* Tiene la finalidad de indicar la dirección donde se puede acceder al servicio Web
  - *location:* Este atributo proporciona la dirección donde se puede acceder al servicio Web. Esta dirección debe ser única.
- + *service_port:* Esta composición vincula el elemento *Service* con *Port*.
- + *binding*: Esta asociación relaciona los elementos *Binding* y *Port* del metamodelo SOAP en términos de elementos *Ecore*, su función es definir la relación ente *Port* y *Binding* que describe cómo y dónde se interactúa con la interfaz de servicio (*PortType*). Cada *Port* contiene un único *Binding*, y un *Binding* podrá estar en muchos Port.
- + *soapAddress:* Esta composición vincula los elementos *Port* de *WSDL* con el *soapAddress.*

4.2. Instanciación del metamodelo SOAP

Para ejemplificar la instanciación del metamodelo SOAP se ha utilizado una vez más la interfaz para la gestión de productos definida en el capítulo 3. Recordemos que esta interfaz define la gestión de productos utilizando las clásicas operaciones CRUD (*Create, Read, Update* y *Delete*) permitiendo aceptar y devolver identificadores, así como elemento de transferencia que caracterizan los productos.

El metamodelo SOAP a diferencia del metamodelo RESTful posee una interfaz muy bien definida a través del elemento *Port Type* donde son especificadas las cuatro operaciones que componen la interfaz. Cada una de estas operaciones está asociada a un *SOAP Binding* donde se declara el estilo a tener en cuenta a la hora de manipular los parámetros. Estos parámetros son definidos en los mensajes (que pueden ser de entrada o de salida) a través del elemento *Part.* Esta parte estaría asociada a un elemento del esquema XML (*Schema XML*).

Una vez dentro del esquema XML se definen los tipos complejos asociados a cada elemento definido por cada parte o parámetro, así como, tipos simples de las propiedades o atributos (que en nuestro caso también son definidos como *Element*) de los tipos complejos definidos.

La figura 4.8 muestra la intancia del metamodelo SOAP contemplando estos conceptos y la relación entre ellos. Con el fin de obtener una vista más abstracta y compacta de la instancia se ha utilizado una representación *Ecore*. Para la instanciación de este metamodelo se ha elegido el estilo documento definido en la sección *Binding* de este capítulo.

4.2.1. Instancia del metamodelo SOAP.

Figura 4.8: Instancia del metamodelo SOAP.

5. Metamodelo Intermedio SOA. Reglas de Conversión

Para realizar la transformación entre metamodelos hemos diseñado un metamodelo intermedio que abstrae los conceptos principales de la arquitectura SOA, caracterizando el concepto principal de un servicio web. Aunque hay una gran complejidad en entender y trabajar con servicios web (Hansen, 2007), el concepto principal que subyace de estos es muy simple: una interfaz expuesta a través de la web, e implementada por una clase.

Esta visión de un servicio web casa con el concepto de servicios de aplicación (Alur, 2003) expuestos como un servicio web utilizando un *Web Service Broker (Alur, 2003)*. Por lo tanto, el proceso de conversión que proponemos supone la existencia de al menos una clase que implementa el servicio, mientras que el proceso de publicación en línea no es capturado, debido a que es muy dependiente del marco de aplicación del servicio web (Hansen, 2007).

Este metamodelo intermedio es utilizado como puente entre los metamodelos SOAP y REST, permitiendo la definición de transformaciones entre estos metamodelos y el metamodelo intermedio, tal y como describe la Figura 1.1. El metamodelo intermedio está inspirado en el diagrama Interfaces del paquete Classes de UML Superstructure 2.0 (OMG, 2005).

Aunque hay diversos estándares para la definición de interfaces (Wikipedia, 2013a), se ha elegido este metamodelo por dos razones: es muy sencillo y está descrito en MOF. No obstante, se han hecho algunas simplificaciones al metamodelo de UML Superstructure para facilitar su aplicación en este trabajo.

Para definir formalmente el proceso de conversión se ha definido un conjunto de transformaciones descritas según el estándar *Query View Transformation* (QVT) *Relations* (OMG, 2011) propuesto por la OMG (ver epígrafe 2.5).

5.1. Metamodelo intermedio SOA propuesto

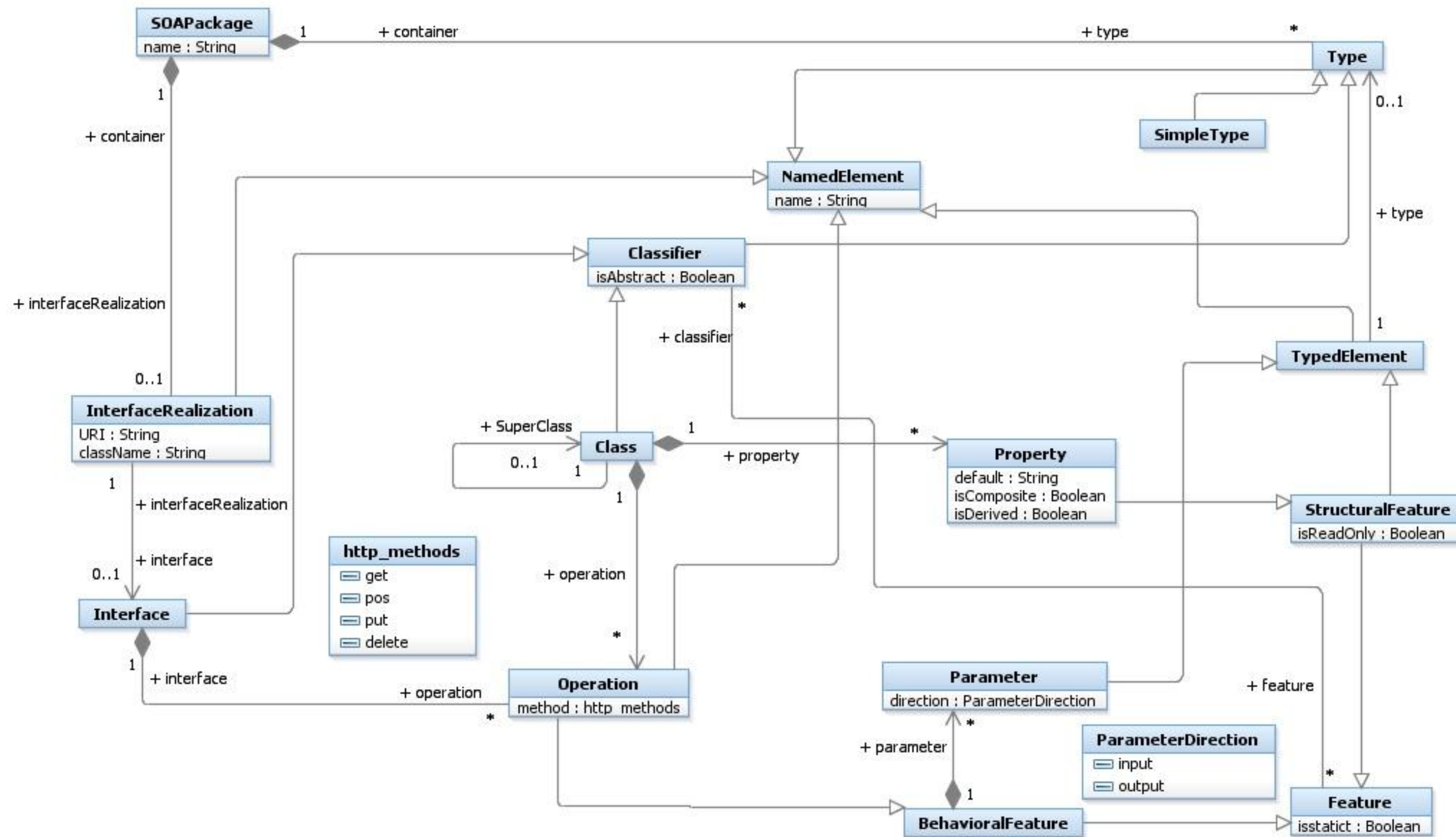

Figura 5.1: Metamodelo SOA

Esta sección describe nuestra propuesta de metamodelo para caracterizar los servicios web desde el punto de vista intermedio SOA en el contexto del modelo de conversión propuesto en este trabajo. La Figura 5.1 presenta el metamodelo SOA propuesto. Sus distintos elementos son analizados en profundidad a continuación.

5.1.1. SOA. SOAPackage

Descripción del elemento SOAPackage

Este paquete fue definido con el fin de agrupar los elementos necesarios para ser posible las reglas de transformaciones QVT. En el lenguaje *Relation*, una transformación entre modelos candidatos se especifica como un conjunto de relaciones que han de tener lugar para llevar a cabo la transformación. Los modelos candidatos tienen un nombre y los tipos de elementos que pueden contener, los cuales quedan restringidos al paquete (p.e. *SOAPackage*) al que se hace referencia en la declaración del modelo candidato.

Diagrama del elemento SOAPackage

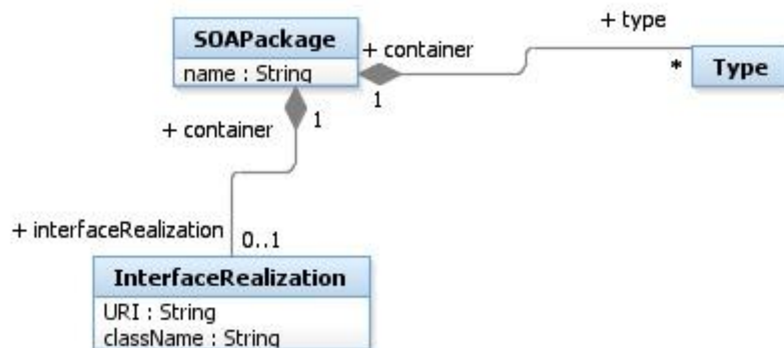

Figura 5.2: Diagrama del elemento *SOAPackage*

Clases y atributos.

- *Clase SOAPackage:* Esta clase agrupar los elementos necesarios para las transformaciones QVT.
- *Clase InterfaceRealization:* Vea epígrafe 5.1.4.
- *Clase Type:* Vea epígrafe 5.1.3.

5.1.2. SOA. NamedElement

Descripción del elemento NamedElement

NamedElement es una clase abstracta que representa los elementos nombrados dentro de la definición de SOA para servicios web. Un elemento nombrado contiene un nombre calificado que le permite ser identificado inequívocamente dentro de una jerarquía de espacios de nombres anidados. Los elementos *TypedElement* y *Type* se relaciona a través de una generalización con *NamedElement* permitiendo tipar los elementos nombrados y nombrar los tipos consecutivamente.

Diagrama del elemento NamedElement

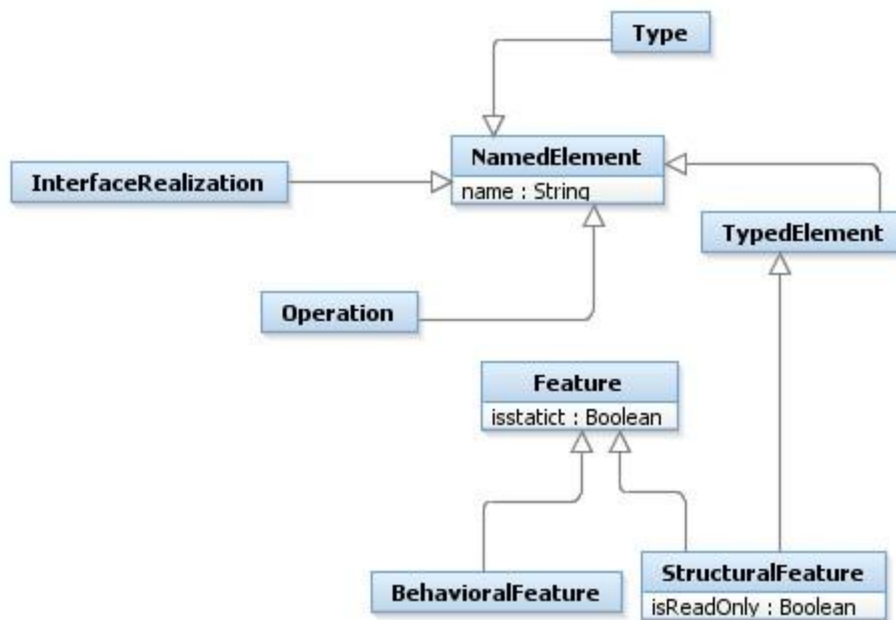

Figura 5.3: Diagrama del elemento *NamedElement*

Clases y atributos.

- *Clase NamedElement:* Define los elementos nombrados.
- *Clase Type:* Vea epígrafe 5.1.3.
- *Clase InterfaceRealization:* Vea epígrafe 5.1.4.
- *Clase Operation:* Vea epígrafe 5.1.4.
- *Clase TypedElement:* Esta clase define los elementos con tipos y se relaciona mediante una generalización con la clase *NamedElement*
- *Clase Classifier:* Vea epígrafe 5.1.3.

- *Clase Feature:* Define una característica que puede ser de comportamiento o de la estructura de un clasificador
  - *isstatic:* Este atributo define si la característica son resultado de instancias del clasificador (*not static*), o del propio clasificador (*static*).
- *Clase BehavioralFeature:* Esta clase define la característica de un clasificador que especifica un aspecto del comportamiento de sus instancias.
- *Clase StructuralFeature:* Define la característica estructural de un clasificador
  - *isReadOnly:* Este atributo permite saber si la característica estructural del clasificador puede o no ser modificada.

5.1.3. SOA. Type

Descripción del elemento Type

Para representar los tipos de datos del metamodelo SOA nos hemos apoyado de la definición que brinda *UML Superstructure* (OMG, 2010) para tipos simples, y para los tipos complejos hemos reutilizado los componentes necesarios de la definición del elemento *Class* de esta librería, asociando los tipos complejos a una clase.

Diagrama del elemento Type.

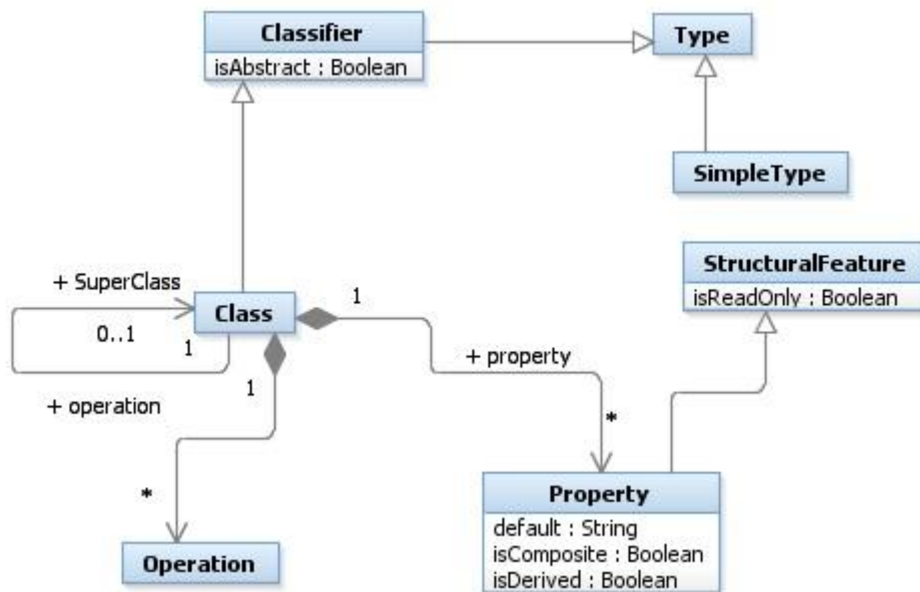

Figura 5.4: Diagrama del elemento *Type*.

Clases y atributos.

- *Clase Type:* Esta clase restringe los valores representados por un elemento con tipo.
- *Clase Classifier:* Un clasificador es un tipo y puede poseer generalizaciones, haciendo posible definir relaciones de generalización a otros clasificadores.
    - *IsAbstract:* Este atributo permite saber si el clasificador proporciona o no una declaración completa. En caso que la declaración sea completa (*true*) este no puede ser instanciada.
- *Clase SimpleType:* Esta clase permite la definición de los tipos de datos simples.
- *Clase Class:* El propósito de una clase es especificar una clasificación de los objetos. Esta característica nos permite utilizar este elemento para la definición de tipos complejos en el metamodelo SOA.
- *Clase Property:* Define las propiedades de una clase, para nuestro modelo nos referimos a las propiedades/atributos de un tipo complejo
    - *default:* Define un valor predeterminado de la propiedad.
    - *isComposite:* Indica si la agregación de la propiedad es compuesta o no.
    - *isDerived:* Especifica si la propiedad es derivado o no, es decir, si su valor o los valores se pueden obtener a partir de otra información.
- *Clase StructuralFeature:* Vea epígrafe 5.1.2.
- *Clase TypedElement:* Vea epígrafe 5.1.2.
- + *SuperClass:* Esta asociación nos da la posibilidad de referencia a la clase que posee la propiedad. Por ejemplo un Subconjunto *Feature :: FeaturingClassifier.*
- + *property:* Composición que vincula el elemento *Class* con sus propiedades.
- + *operation:* Esta composición relaciona el elemento *Class* con *Operation* (ver figura 4.1), a pesar de no aporta ninguna información para nuestra definición de tipos complejos, hemos decidido incluirla en el metamodelo por formar parte de la definición del elemento *Class*.

5.1.4. SOA. Interface

Descripción del elemento Interface

Una interfaz es un tipo de clasificador que representa la declaración de un conjunto de obligaciones y funciones públicas coherentes. Este elemento especifica un contrato, así

cualquier instancia de un clasificador que realiza la interfaz (*InterfaceRealization*) debe cumplir este contrato (OMG, 2010).

Dado que las interfaces son declaraciones, estas no son cuestionables. En su lugar, una especificación de interfaz se implementa mediante una instancia de un clasificador, lo que significa que el clasificador instanciable presenta una fachada pública que se ajusta a la especificación de la interfaz (OMG, 2010).

Diagrama del elemento Interface

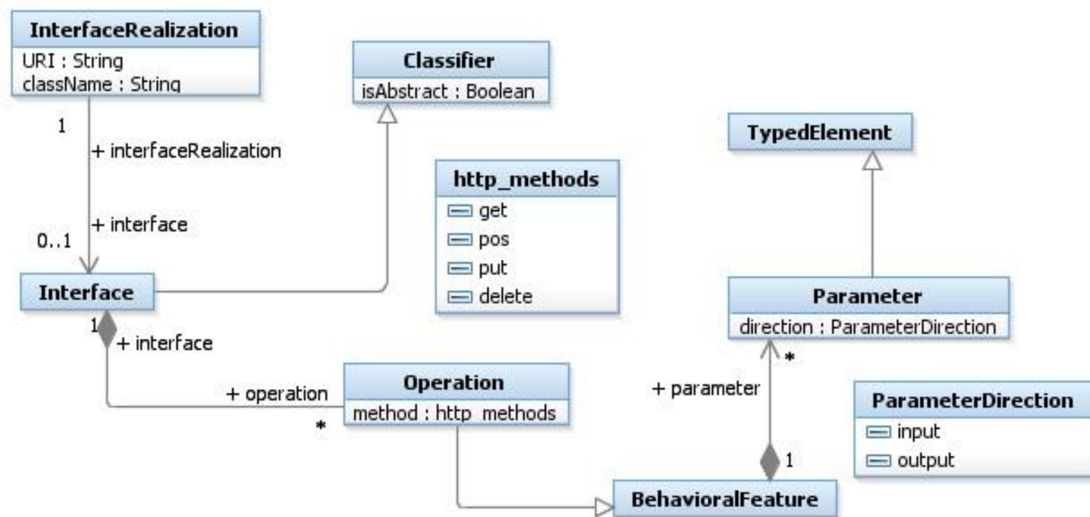

Figura 5.5: Diagrama del elemento *Interface*

Clases y atributos

- *Clase InterfaceRealization:* Define la clase que implementa la interfaz.
  - *URI:* Permite el acceso a la clase que implementación la interfaz.
  - *claseName:* Especifica el nombre de la clase que implementa o realiza la interfaz.
- *Clase NamedElement:* Vea epígrafe 5.1.2.
- *Clase Classifier:* Vea epígrafe 5.1.3.
- *Clase Interface:* Define el elemento interfaz dentro del metamodelo SOA
- *Clase Operation:* Define las operaciones de la interfaz.
  - *method:* Especifica el método que define la operación. Este atributo es de tipo *http_method.*
- *Clase BehavioralFeature:* Vea epígrafe 5.1.2.

- *Clase Parameter:* Define los parámetros presente en una operación.
    - *direction:* Especifica el tipo de parámetros a ser definido. Este atributo es de tipo *ParameterDirection.*
- *Clase TypedElement:* Vea epígrafe 5.1.2.
- *Clase ParameterDirection:* Especifica los posible tipos de parámetros (*input* y *output*).
- *Clase http_methods:* Especifica los posibles tipos de métodos admitido por nuestro metamodelo SOA (*GET, POST, PUT* y *DELETE*). Estos son necesarios para marcar las funciones a la hora de transformarlo a modelos RESTful. La filosofía es similar a la seguida por marcos como JAX-RS.
- *+ interfaceRealization…+ interface:* Asociación que vincula el elemento *Interface* con la clase que la realiza (*InterfaceRealization*)
- *+ interface…+ operation:* Generalización que vincula la interfaz (*Interface*) con sus operaciones (*Operation*).
- *+ parameter:* Composición que permite vincular las operaciones con sus parámetros. Note que el elemento *Operation* se relaciona con los parámetros a través de la generalización existente con la clase *BehavioralFeature.*

5.2. Instanciación del metamodelo SOA

Para ejemplificar la instanciación del metamodelo SOA se ha utilizado una vez más la interfaz para la gestión de productos definida en el capítulo 3. Veamos una representación UML de esta interfaz para el metamodelo SOA.

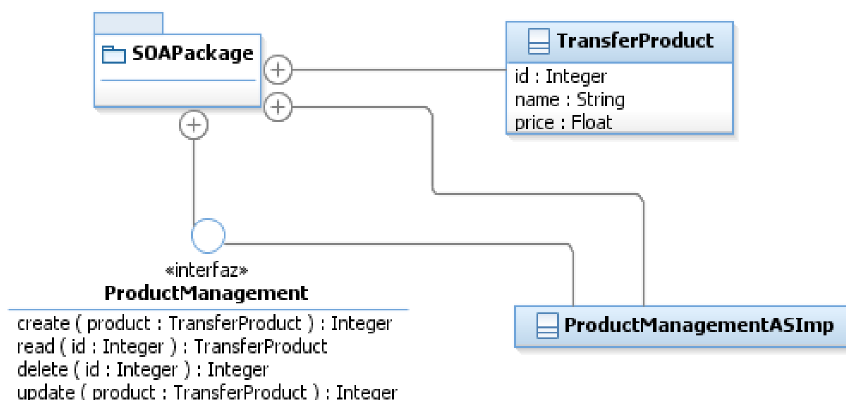

Figura 5.6: Representación en UML de la interfaz para la gestión de productos

Para instanciar el metamodelo SOA hemos cazado la clase *ProductManagementASImp* con la clase que gestiona la interfaz *(InterfazRealization)*, tipos simples se han declarado en base a las propiedades (*id, name* y *price*) definidas por el tipo complejo *TransferProduct*. Métodos CRUD asociados a las operaciones han sido asignados a las operaciones definidas por la interfaz *ProductManagement*, así como parámetros de entrada y de salida para cada una de estas operaciones. También se ha declarado un *SOAPackage* con el fin de envolver los elementos necesarios para efectuar la conversión. La figura 5.7 ilustra esta instancia.

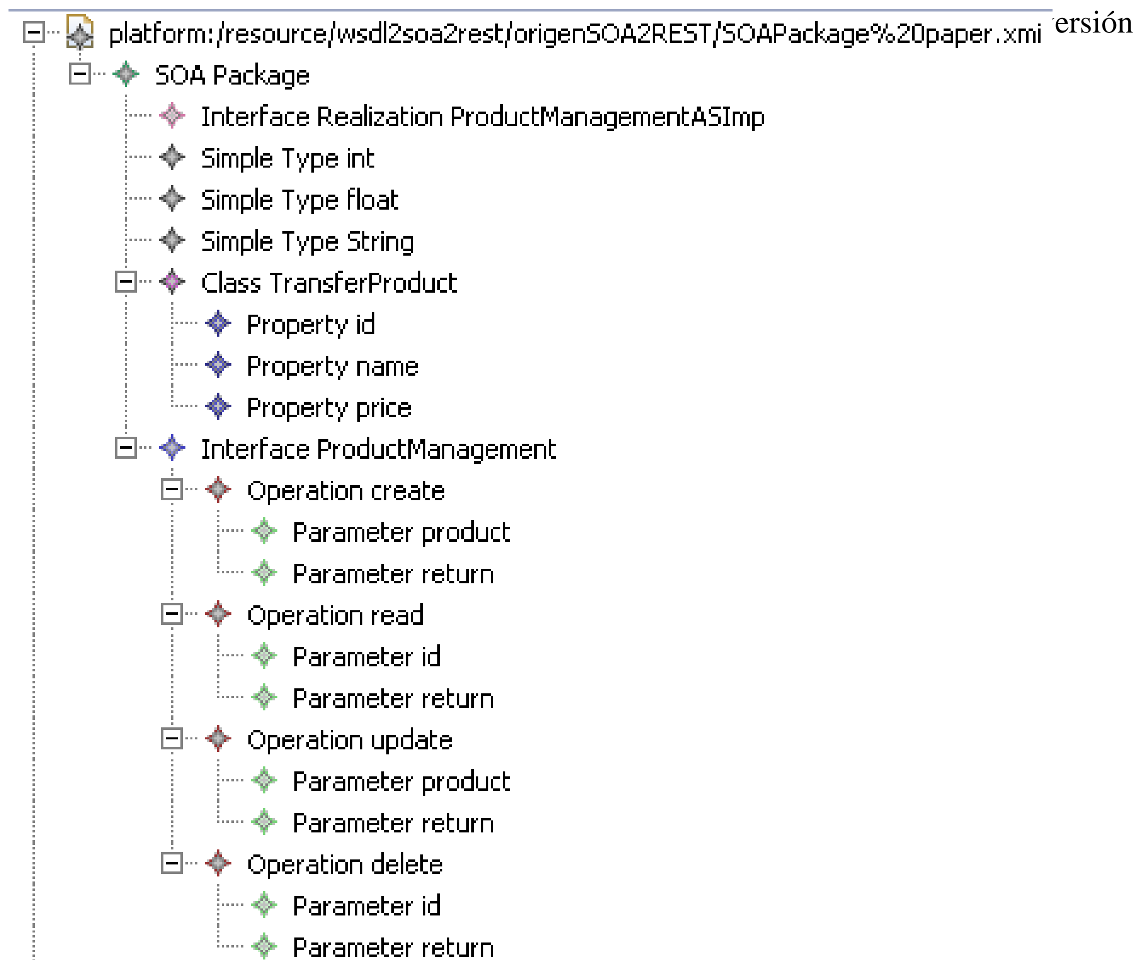

Figura 5.7: Instancia del metamodelo SOA

5.3. Reglas de Conversión

Hasta aqui la idea principal consiste en proporcionar un metamodelo MOF intermedio que abstrae los conceptos principales de un servicio web. Teniendo en cuenta que ya poseemos un metamodelo RESTful (vea capítulo 3) y un meta-modelo que caracteriza a la representación de WSDL de los servicios web SOAP (vea capítulo 4), solo nos queda establecer reglas de transformaciones QVT Relations que se definen con origen en los metamodelos SOAP y RESTful y con destino en el metamodelo intermedio SOA, así como, entre el metamodelo intermedio SOA y los metamodelos SOAP y RESTful, definiéndose formalmente un mecanismo de conversión. La figura 1.1 describe nuestro enfoque.

Usando este enfoque marcos de conversión puede definirse para la conversión entre los servicios *RESTful* y SOAP. Además, el mismo servicio puede ser publicado tanto como un SOAP o un servicio *RESTful*.

Es importante resaltar que nuestra aproximación es de alto nivel y no tiene en cuenta cuestiones tales como traducir en la práctica un parámetro XML RESTful en un parámetro Java SOAP. Para tal fin, pueden utilizarse marcos de conversión como JAXB, guiados por las reglas de alto nivel definidas en este trabajo. El uso de dichos marcos queda propuesto como trabajo futuro.

5.3.1. Conversión RESTful to SOA

La tabla 5.1 muestra los elementos involucrados en la lógica que subyace a las transformaciones *RESTful to SOA* definidas en el apéndice A.1.

Tabla 5.1: Transformaciones *RESTful to SOA*

RESTful	**SOA**
Package	Package
RESTService	Interface
	InterfaceRealization
Method	Operation
RequestParam	Param.direction='input'
ResponseParam	Param.direction='output'
SimpleType	SimpleType
ComplexType	Class
Property	Property

Básicamente, el servicio RESTful se transforman en interfaces de SOA, los parámetros de la petición (*Request*) en parámetros *input*, los parámetros de respuesta (*Response*) en parámetros *output*, tipos simples en tipos simples y tipos complejos en clases. Aplicando esta lógica de transformación obtendríamos que el modelo *RESTful* representado en la figura 3.9 se transforma en la interfaz SOA mostrada en la figura 5.7. La clase responsable de la implementación del servicio web *RESTful* almacenada en `REST::Method.className` es utilizada para definir el valor del atributo `SOA::InterfaceRealization.className`. Esta última definición obliga que los diferentes métodos del metamodelo *RESTful* tenga que utilizar la misma clase de implementación. Aunque esta restricción no está definida en el metamodelo, podemos

considerarla razonable de acuerdo a los *frameworks* actuales, tal como JAX-RS (Burke, 2009).

Una vez definida las transformaciones *RESTful to SOA,* son declaradas a continuación transformaciones *SOA to RESTful*, a pesar que las reglas QVT son bidireccionales las diferencias existente entre los metamodelos origen y destino nos obliga a definir este segundo conjunto de regla para completar la transformación bidireccional. La tabla 5.2 muestra los elementos involucrados en la lógica que subyace a las transformaciones *SOA to RESTful* definidas en el apéndice A.2.

Tabla 5.2: Transformaciones *SOA to RESTful*

SOA	**RESTful**
Package	Package
Interface	Service
Operation.method='get'	Method_GET
Operation.method='pos'	Method_POST
Operation.method='put'	Method_PUT
Operation.method='delete'	Method_DELETE
Parameter.direction='input'	Request.HeaderParam
Parameter.direction='output'	Response.HeaderParam
SimpleType	SimpleType
Class	ComplexType
Property	Property

Básicamente interfaces de SOA se transforman en servicios *RESTful*, operaciones en métodos (teniendo en cuenta el atributo que define el método HTTP en concreto), los parámetros de entrada en parámetros de la petición *header*, parámetros de salida en los parámetros de respuesta *header*, los tipos simples en tipos simples, y las clases en tipos complejos.

En los apéndices A.1 y A.2 se proporcionan una definición formal y detallada de las reglas de transformación implementadas en QVT Relations de los servicios web *RESTful* y las interfaces SOA.

5.3.2. Conversión SOAP to SOA.

Para efectuar las transformaciones *SOAP to SOA* los puertos (*Port*) WSDL se transforman en clases que realizan una interfaz (*InterfaceRealization*) SOA, el elemento

PortType en interfaz, operaciones en operaciones SOA, las partes de los mensajes de solicitud y respuesta en parámetros de estrada y salida respectivamente, tipos simples en tipos simples y tipos complejos en clases. Para la conversión sólo se consideraron los servicios SOAP con estilo *document literal/wrapped* debido a que hoy en día son más comunes en la industria web (Hansen, 2007). En la tabla 5.3 muestra los elementos involucrados en la lógica que subyace a las transformaciones *SOAP to SOA* definidas en el apéndice A.3.

Tabla 5.3: Transformaciones *SOAP to SOA*

SOAP/WSDL	SOA
Package	Package
Port	InterfaceRealization
PortType	Interface
Operation	Operation
(Request)Message.Part.Element.ComplexType.Element	Parameter.direction ='input'
(Response)Message.Part.Element.ComplexType.Element	Parameter.direction ='output'
SimpleType	SimpleType
ComplextType	Class
Element	Property

Aunque las transformaciones relaciones QVT son bidireccionales, debido a la estructura heterogénea de los metamodelos SOA y SOAP/WSDL, dos conjunto de relaciones QVT han sido definidos, *SOAP to SOA*, y otro para *SOA to SOAP*. La Tabla 5.4 describe los elementos involucrados en esta transformación.

Esencialmente el paquete *SOAPackage* se utiliza para definir los elementos generales del metamodelo WSDL involucrados en la transformación, las interfaces se traducen en *PortType* e informaciones asociadas al enlace SOAP (*Binding* y *soapBinding*), realizaciones de interfaces en puertos, tipos simples en tipos simples, las operaciones en elementos de entrada (*input*) y salida (*output*), además son generados tipos complejos y mensajes según la operación asociada con el fin de construir un servicio con estilo *document literal/wrapped,* clase se transforman en tipo complejo y propiedades en elementos. En los apéndices A.3 y A.4 se proporcionan una definición formal y

detallada de las reglas de transformación implementadas en *QVT Relations* entre los servicios web SOAP y los interfaces SOA.

Tabla 5.4: Transformaciones *SOA to SOAP*

SOA	**SOAP/WSDL**
SOAPackage	WSDLPackage
	XMLSchema
	Type
	Service
Interface	PortType
	Binding
	SOAPBinding
InterfaceRealization	Port
SimpleType	SimpleType
operation	2 Element (input/output wrappers)
	2 ComplexType(input/output wrappers types)
	2 Messages (request/response)
	operation
Class	ComplexType
Property	Element

Es importante resaltar que de acuerdo a las reglas de transformacion *QVT Relations* la relaciones entre: *Part* y *Element*, *Operation* y *Message*, *soapOperation* y *Operation*, no cruzan las instancias de metaclases, solo se almacenan los nombre de los elementos, esto simplifica las reglas de transformación permitiendo la generación de instancia XML WSDL, teniendo solo en cuenta la referencia en términos del nombre del elemento.

6. Conclusiones y trabajo futuro.

Con este trabajo hemos proporcionado una aproximación basada en MDA para definir un mecanismo de conversión entre servicios web SOAP y RESTful. Dicho mecanismo se basa en un metamodelo MOF intermedio que abstrae los principales conceptos de un servicio SOA, junto con otros dos metamodelos MOF, uno para los servicios web RESTful y otro para los servicios SOAP a través de su lenguaje descriptor (WSDL 1.1). Además, nuestra aproximación incluye reglas de transformación definidas en QVT *Relations* con su origen en los metamodelos SOAP y RESTful, con el metamodelo intermedio SOA como objetivo, así como, entre el metamodelo intermedio SOA y los metamodelos SOAP y RESTful.

El estudio y análisis de las diferentes implementaciones web y de los principios básicos de SOA para servicios web SOAP y RESTful nos permite concluir que a pesar de las diferencias existentes entre ambas implementaciones estas contienen un mismo progenitor, la arquitectura orientada a servicio (SOA). Para los servicios SOAP esto es evidente, pero para los servicios RESTful donde la idea fundamental es el recurso y no el servicio, esta conclusión no parece tan evidente. Teniendo en cuenta esto, cabe destacar que aunque el estilo RESTful centre su atención en el recurso, no nos podemos olvidar del entorno a que pertenece: los servicios web. Por tanto, en una arquitectura multicapa, sea cual sea la implementación de un servicio web, ésta no deja ser una interfaz expuesta a través de la web, e implementada por una clase. Esta idea fue la clave para lograr el objetivo fundamental de este trabajo.

Además de definir un mecanismo de conversión, proponemos modelos formales que caracterizan los servicios web RESTful y SOAP. A pesar de existir algunos metamodelos RESTful como WADL o WSDL 2.0, ninguno de ellos se ajustaba a las necesidades de nuestra aproximación, ya que, son muy complejos con elementos innecesarios para nuestros objetivos de conversión. Lo mismo sucedia en el caso SOAP y SOA. Por tanto, hemos optado por construir nuestros propios metamodelos MOF. Estos fueron concebidos centrándonos en los requisitos de cada implementación web. Así, algunas limitaciones o aportes se han tenido en cuenta a la hora de construir los metamodelos, con el fin de lograr la conversión, siendo cada uno de estos justificados y documentados en este trabajo.

El enfoque abstracto definido en este documento puede ser usado en ESBs, gateways SOA y entornos de desarrollo para la transformación entre servicios web SOAP y RESTful, o en la exposición de interfaces como servicios web. Sin embargo, el enfoque propuesto en este trabajo evita proporcionar los detalles del servicio así como los de su publicación, específicos para cada plataforma y *framework* concreto.

Vale la pena mencionar que, examinando las instancias del mismo servicio web según los tres metamodelos, la descripción SOA es más simple que la descripción RESTful (y la descripción RESTful es más simple que la descripción SOAP). Por lo tanto, si obviamos los detalles de la publicación, la definición de servicios web en términos del metamodelo SOA definido en este trabajo, podría ser utilizada para definir servicios RESTful y SOAP, con el fin de entender estos conceptos desde un punto de vista más legible para las personas.

Como trabajo futuro pretendemos incluir el desarrollo de un marco J2EE para la aplicación práctica de este marco formal. Así, podría llevarse a un nivel operacional, por ejemplo, la conversión entre un parámetro RESTful descrito en formato XML y su homólogo SOAP descrito como un objeto Java utilizando JAXB. También queda como trabajo futura la generalización de este trabajo, para hacer frente a los casos no cubiertos en la actualidad (por ejemplo, la presencia de los servicios web RPC en SOAP).

Bibliografía.

Abbas, A. (2009). Exposing RESTful services using an Enterprise Service Bus. http://www.ibm.com/developerworks/webservices/library/ws-RESTesb/index.html

Adell, J. (1994). *La internet como telaraña: el world-wide-web* Retrieved from http://www.uv.es/~biblios/mei3/Web022.html#ArquiWeb

Alur, D. (2003). *Core J2EE™ Patterns: Best Practices and Design Strategies* (pp. 528). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/book/programming/java/0131422464

APACHE. (2011). Apache ServiceMix. Retrieved from Service Mix website: http://servicemix.apache.org/

APACHE. (2012). Apache Synapse Enterprise Service Bus (ESB). *Version: 2.1.0*. Retrieved from Synapse website: http://synapse.apache.org/

Arias., J. H. (2005). ESB: Enterprise Services Bus “La siguiente generación de plataformas para la integración empresarial de aplicaciones”. http://www.google.es/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=1&ved=0CC8QFjAA&url=http%3A%2F%2Fwww.acis.org.co%2Ffileadmin%2FBase_de_Conocimiento%2FXXV_Salon_de_Informatica%2FESB-IntegrationNextGeneration-JorgeArias_light.ppt&ei=GxEnUv2JFPCO7QaSqICgCg&usg=AFQjCNG2yum8NUDyCOGGAYuhs9GbYZLlCw&bvm=bv.51495398,d.d2k&cad=rja

Barco, A. (2006). Artículo Tecnológico: "WSDL: El contrato de un Servicio". http://arquitecturaorientadaaservicios.blogspot.com.es/2006/12/artculo-tecnolgico-wsdl-el-contrato-de.html

Bayer, T. (2009). WSDL Reading, a Beginner's Guide. Retrieved from predic8 website: http://predic8.com/wsdl-reading.htm

Bolton, F. (2001). *Pure CORBA* (pp. 944). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/9780768683646

Brogden, W. (2009). Descriptive Languages for RESTful Services. http://searchsoa.techtarget.com/tip/Descriptive-Languages-for-RESTful-Services#content

Burke, B. (2009). *RESTful Java with JAX-RS* (pp. 320). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/9780596809300

C. Steel, R. N. a. R. L. (2005). *Core Security Patterns: Best Practices and Strategies for J2EE™, Web Services, and Identity Management* (pp. 1088). Retrieved from http://0 proquest.safaribooksonline.com.cisne.sim.ucm.es/book/networking/security/0131463071

Caponi, M. (2008). Mensajería en Sistemas de Información. http://www.google.es/url?sa=t&rct=j&q=componentes%20arquitectonicos%20de%20apache%20servicemix&source=web&cd=1&ved=0CCUQFjAA&url=http%3A%2F%2Fproyecto-grado-masi.googlecode.com%2Ffiles%2FInforme_Proyecto_Mensajeria_SI.pdf&ei=KEOhT4mkA-na0QXZ6LGQCA&usg=AFQjCNGcdValZOUT-7Cl_bFuv-LU2VgpCg

CISCO. (2013). Cisco ACE XML Gateways. Retrieved from CISCO website: http://www.cisco.com/en/US/products/ps7314/index.html

Eiton Brun, R. (2002). XML y Servicios Web. https://docs.google.com/viewer?a=v&q=cache:yQ92_VeK49YJ:www.forpas.us.es/aula/xml/doc/08.XML%2520y%2520Servicios%2520Web.ppt+ricardo+2002%2B+servicios+web&hl=es&gl=es&pid=bl&srcid=ADGEESip3n1WqE6Sr9dww1hojsv_zs3ux5cZHLlZINvfNKULiqNHoEdjhMnxdq_1kYtkS--2XZhjNnoCJLf9vT1_LLaiDHC_5nF6JPQUEjg5wiCqfnynMEYQxYKqINWvWSx0wQwnF_NA&sig=AHIEtbR0ky_eyi1ucPXCQ7OktSI1C47xLg

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design* (pp. 792). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/0131858580

Erl, T. (2008). *SOA Design Patterns* (pp. 864). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/9780136135166

F. Sáez, P. A. (2009). Un Análisis Crítico de la Aproximación Model-Driven Architecture. http://eprints.ucm.es/9880/

Fielding, R. (1999). Hypertext Transfer Protocol -- HTTP/1.1. Chapter 9 Method Definitions. Retrieved from W3 website: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Fielding, R. T. (2000). Architectural Stypes and the Desing of Network-based Software Architectural. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fielding, R. T. (2008). On software architecture. Retrieved from Untangled musings of Roy T. Fielding website: http://roy.gbiv.com/untangled/2008/on-software-architecture

Fiorano. (2013). Fiorano ESB (Enterprise Service Bus). Retrieved from Fiorano website: http://www.fiorano.com/products/ESB-enterprise-service-bus/Fiorano-ESB-enterprise-service-bus.php

Fowler, M. (2002). *Patterns of Enterprise Application Architecture* (pp. 560). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/book/software-engineering-and-development/patterns/0321127420

Ganatra, S. (2012). ColdFusion 10: Understanding REST parameters. Retrieved from http://www.sagarganatra.com/2012/02/coldfusion-10-understanding-rest.html

González., B. (2004). WSDL para la documentación de Servicios Web. http://www.desarrolloweb.com/articulos/1581.php

Graham, S., Simeonov, S., Boubez, T., Davis, D., Daniels, G., Nakamura, Y., & Neyama, R. (2001). *Building Web Services with Java™: Making Sense of XML, SOAP, WSDL, and UDDI (Chapter 6 and 7)* Retrieved from http://82.157.70.109/mirrorbooks/buildingwebserviceswithjava/0672321815_ch06.html

Guerra, J. (2009). Oracle Service Bus y Oracle Service Registry. Taller Técnico. http://www.google.es/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=1&ved=0CC8QFjAA&url=http%3A%2F%2Ftrac.ingenian.com%2Fproyectos%2Fmen-soa%2Fexport%2F30%2Fimpl%2Fosb%2Fdoc%2Fosb_.pdf&ei=oRQnUqrwKMvY7AbOm4DwDw&usg=AFQjCNHq7MgVZIT66dQXauz9DmsD_gZcXg&bvm=bv.51495398,d.d2k

Hansen, M. D. (2007). SOA Using Java™ Web Services (pp. 608): Prentice Hall.

Harmon, P. (2004). The OMG's Model Driven Architecture and BPM. *Volume 2, No. 5*. http://www.bptrends.com/publicationfiles/05-04%20NL%20MDA%20and%20BPM.pdf

IBM. (2008). REST to SOAP extension. http://publib.boulder.ibm.com/infocenter/wsmashin/v1r0/index.jsp?topic=/com.ibm.websphere.sMash.doc/assemble/zero.connection.soap/docs/en/overview.html

IBM. (2012). *What is SOAP?* Retrieved from http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/index.jsp?topic=%2Fcom.ibm.etools.mft.doc%2Fac55770_.htm

IBM. (2013a). Defining parameters for request representations to resources in RESTful applications. http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.nd.multiplatform.doc%2Fae%2Ftwbs_jaxrs_defresource_parmexchdata.html

IBM. (2013b). Defining resource methods for RESTful applications. http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.base.doc%2Fae%2Ftwbs_jaxrs_defresource_httpmeth.html&lang%3Dpt_BR

IBM. (2013c). IBM WebSphere Enterprise Service Bus (ESB). Retrieved from IBM website: http://www-01.ibm.com/software/ar/info/middleware/applications/

INTEL. (2013). Service Gateway - Application Security Made Easy. Retrieved from Intel-API Management, PCI Compliance, Enterprise Mobile Access website: http://cloudsecurity.intel.com/service-gateway

Juan M. Vara, V. D. C. a. E. M. (2005). WSDL Automatic Generation from UML Models in a MDA Framework. *Vol.1*, 12.http://nwesp.org/ijwsp/2005/vol1/1.pdf

Kamerkar, N. (2012). Understanding WSDL. http://www.techavalanche.com/2012/03/22/understanding-wsdl/

Kleppe, A. (2003). MDA Explained: The Model Driven Architecture™: Practice and Promise. In J. Warmer & W. Bast (Eds.), (pp. 192): Addison-Wesley Professional.

Lascelles, F. (2010). REST JSON to SOAP Conversion Tutorial. http://java.sys-con.com/node/1341004

LAYER7. (2013). SOA Gateways. Retrieved from LAYER 7 Technologies website: http://www.layer7tech.com/products/soa-gateway

Mandel, L. (2008). Describe REST Web services with WSDL 2.0. http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/

Microsoft. (2013). Service Gateways. Retrieved from msdn website: http://msdn.microsoft.com/en-us/library/ff650101.aspx#feedback

Microsoft. (2013). Enterprise Service Bus (ESB). Retrieved from Microsoft Biz Talk Server website: http://www.microsoft.com/en-us/biztalk/product-information/enterprise-service-bus.aspx

MuleSoft. (2013). Mule ESB. Retrieved from Mule Soft connecting the new enterprise website: http://www.mulesoft.com/mule-esb-open-source-esb

Muschett, B. (2011). Implementing a Web 2.0 RESTful facade enabled with JSON using WebSphere DataPower SOA Appliances. http://www.ibm.com/developerworks/websphere/library/techarticles/0912_muschett/0912_muschett.html

Navarro Marset, R. (2007). REST vs Web Services. *Modelo Diseño y Implementacion de Servicios Web*. http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf

O'Neill, M. (2008). How to convert from REST to SOAP. http://www.soatothecloud.com/2008/11/how-to-convert-from-rest-to-soap.html

OMG. (2003). MDA Guide Version 1.0.1. http://www.enterprise-architecture.info/Images/MDA/MDA%20Guide%20v1-0-1.pdf

OMG. (2005). UML 2.0 Infrastructure Specification. *version 2.0*. http://www.omg.org/spec/UML/2.0/

OMG. (2006). Meta Object Facility (MOF) Core Specification. *Version 2.0*. http://www.omg.org/spec/MOF/2.0/

OMG. (2006a). Object Constraint Language. *version 2.0*. http://www.omg.org/spec/OCL/2.0/PDF/

OMG. (2010). OMG Unified Modeling Language™ (OMG UML),Superstructure. *Version 2.3*. http://www.omg.org/spec/UML/2.3/

OMG. (2011). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. *Version 1.1*. http://www.omg.org/spec/QVT/1.1/

Onstine, W. (2011). Creación de SOA con servicios web usando WebSphere Studio, parte 1: Introducción a SOA y servicios web.

http://www.ibm.com/developerworks/ssa/webservices/tutorials/ws-soa1/section3.html

ORACLE. (2002). Core J2EE Patterns - Data Access Object. http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

ORACLE. (2008). Oracle Enterprise Service Bus Developer's Guide 10g (10.1.3.4.0). http://docs.oracle.com/cd/E11036_01/integrate.1013/b28212.pdf

ORACLE. (2010a). RESTful Web Services Developer's Guide. http://docs.oracle.com/cd/E19776-01/820-4867/ggqny/

ORACLE. (2010b). The Java EE 6 Tutorial. (Chapter 13 Building RESTful Web Services with JAX-RS). http://docs.oracle.com/cd/E19798-01/821-1841/giepu/index.html

ORACLE. (2011). Oracle® Fusion Middleware Concepts and Architecture for Oracle Service Bus 11*g* Release 1 (11.1.1.6.0). http://docs.oracle.com/cd/E14571_01/doc.1111/e15020/architecture_overview.htm

ORACLE. (2013a). Oracle API Gateway. Retrieved from ORACLE website: http://www.oracle.com/us/products/middleware/identity-management/api-gateway/overview/index.html

ORACLE. (2013b). Oracle Service Bus. Retrieved from ORACLE website: http://www.oracle.com/technetwork/middleware/service-bus/overview/index.html

Robinson, R. (2004). Understand Enterprise Service Bus scenarios and solutions in Service-Oriented Architecture. *Part 1*. Retrieved from Developer Works - IBM website: http://www.ibm.com/developerworks/webservices/library/ws-esbscen/#2.2

Rodriguez, A. (2008). RESTful Web services: The basics. http://www.ibm.com/developerworks/webservices/library/ws-restful/

Rubio, D. (2007). WADL: The REST answer to WSDL. http://searchsoa.techtarget.com/tip/WADL-The-REST-answer-to-WSDL

Ryan, J. (2011). Rethinking the ESB: Building a Simple, Secure, Scalable Service Bus with an SOA Gateway. Retrieved from Compiter World website: http://www.computerworld.com/s/article/9219205/Rethinking_the_ESB_Building_a_simple_secure_scalable_Service_Bus_with_an_SOA_Gateway

San Cristobal Ruiz, E. (2010). Metodología estructura y desarrollo de interfaces intermedias para la conexión de laboratorios remotos y virtuales a plataforma educativa. http://meteo.ieec.uned.es/www_Usumeteo7/tesis%20elio%20san%20cristobal%20ruiz%20(uned).pdf

Saugar, S. (2011). REST: La Arquitectura de la World Wide Web. http://zenon.etsii.urjc.es/grupo/docencia/as/material/tema6.pdf

Scrib. (2013). Best Practices in Deploying XML SOA Gateways. http://es.scribd.com/doc/43981774/Best-Practices-in-Deploying-XML-SOA-Gateways#

Shohoud, Y. (2003). RPC/Literal and Freedom of Choice. Retrieved from Microsoft Developer Network website: http://msdn.microsoft.com/en-us/library/ms996466.aspx

Skonnard, A. (2003a). Understanding WSDL. http://msdn.microsoft.com/en-us/library/ms996486.aspx#understand_topic3

Skonnard, A. (2003b). Understanding XML Schema. http://msdn.microsoft.com/pt-pt/library/aa468557.aspx#understandxsd_topic6

Soriano, J. (2013). REST: Un estilo arquitectónico para la WWW. http://pegaso.ls.fi.upm.es/sos/images/documentacion/material_aux_t2/tema4_RestfulWS.pdf

System, F. (2008). Introduction to SOA Gateways: Best Practices, Benefits and Requirements. Retrieved from SOA World Magazine website: http://es.scribd.com/doc/43981774/Best-Practices-in-Deploying-XML-SOA-Gateways#

T. Erl, B. C., C. Pautasso, R. Balasubramanian. (2012). *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST* (pp. 624). Retrieved from http://0-proquest.safaribooksonline.com.cisne.sim.ucm.es/9780132869904

Tilkov, S. (2007). A Brief Introduction to REST. Retrieved from Info Queve website: http://www.infoq.com/articles/rest-introduction

Valverde, F. (2013). Dealing with REST Services in Model-driven Web Engineering Methods. http://www.inf.udec.cl/~adweb/docs/ecweb2009/ValverdePastor_JSWEB09.pdf

VORDEL. (2011). Enterprise class application gateway provides integration, security, control, and acceleration across applications on-premises and in the Cloud. http://www.vordel.com/research/

VORDEL. (2013). Service Virtualization With The Vordel XML Gateway. Retrieved from axway website: http://www.vordel.com/resources/Service_Virtualization.html

Vázquez, J. J. (2009). Tutorial: primeros pasos con Apache ServiceMix 4. Retrieved 02 de mayo, 2011, from http://blogs.tecsisa.com/tutoriales-soa/tutorial-primeros-pasos-con-apache-servicemix-4/

W3C. (2001). Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl#_types)

W3C. (2003). W3C Recommendation. SOAP Version 1.2 Part 0: Primer. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/#L1165

W3C. (2004). XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/#ref6

W3C. (2008). Extensible Markup Language (XML) 1.0. Retrieved from W3C website: http://www.w3.org/TR/2008/REC-xml-20081126/

W3C. (2009). Web Application Description Language. Retrieved from W3C Member Submision website: http://www.w3.org/Submission/wadl/

W3Schools. (2013). JSON Tutorial. Retrieved from W3Schools website: http://www.w3schools.com/json/default.asp

Wikipedia. (2013a). Interface description language Retrieved from Wikipedia the free encyclopedia website: http://en.wikipedia.org/wiki/Interface_description_language

Wikipedia. (2013b). Localizador uniforme de recursos. Retrieved from Wikipedia la enciclopédia libre website: http://es.wikipedia.org/wiki/Localizador_Uniforme_de_Recursos

Wikipedia. (2013c). Uniform Resource Identifier. http://es.wikipedia.org/wiki/Uniform_Resource_Identifier

WS-I. (2006). Basic Profile. *version 1.1*. Retrieved from WS-I website: http://www.ws-i.org/profiles/basicprofile-1.1.html

WSO2. (2013). WSO2 Enterprise Service Bus. Retrieved from WSO2 website: http://wso2.com/products/enterprise-service-bus/

Yang, H. (2013). "soap:operation" - SOAPAction Header Line. http://www.herongyang.com/WSDL/WSDL-11-Extension-SOAP-11-operation-SOAPAction.html

Apéndice A. Reglas de transformaciones QVT.

A.1. RESTful to SOA.

```
transformation REST2SOA(rest:REST, soa:SOA) {

       top relation Package2Package {

              checkonly domain rest rp:REST::RESTPackage{ };

              enforce domain soa sp:SOA::SOAPackage{  };

       }

top    relation Service2Interface {

              mn, cn:String;

              checkonly domain rest rs:REST::RESTService { method = m:REST::Method { name= mn, className = cn },
                                                                                          container = rp:REST::RESTPackage {}
                                                                                          };


              enforce domain soa i:SOA::Interface {  name = 'GeneratedSOAInterface',
                                                                                 operation = op:SOA::Operation { name = mn },
                                                                                 container = sp:SOA::SOAPackage {}
                                                                            };

              when { Package2Package(rp, sp); }
              where { RequestParam2InputParam(m, op);
                             ResponseParam2ReturnParam(m, op);
                             }

    }

  top   relation Service2InterfaceRealization {

              cn:String;
              i:SOA::Interface;

           checkonly domain rest rs:REST::RESTService { method = m:REST::Method_POST { className = cn  },
                                                                                          container = rp:REST::RESTPackage {}
```

```
                                                                          };

        enforce domain soa  ir:SOA::InterfaceRealization { name = 'GeneratedSOAInterfaceImp',
                                                                                    className = cn,
                                                                                    interface = i,
                                                                                    container = sp:SOA::SOAPackage {}
                                                                                            };

        when { Service2Interface(rs, i);
                  Package2Package(rp, sp);
                 }

}



    relation RequestParam2InputParam {

        pn:String;


        checkonly domain rest method:REST::Method { request = rq:REST::Request { globalparameter = restParam:REST::Parameter {  name = pn,

                                                                                   type = restType:REST::Type {  }

                                                                           }
                                                                                                                                             }

                                                                                   };


        enforce domain soa operation:SOA::Operation { parameter = soaParam:SOA::Parameter { name= pn,

                direction=SOA::ParameterDirection::input,

                type = soaType:SOA::Type {  }

            }

    };


        when { SimpleType2SimpleType(restType, soaType) or ComplexType2Class(restType, soaType);
```

```
                    }

    }
        relation ResponseParam2ReturnParam {
                rpn:String;
                checkonly domain rest method:REST::Method { response = rs:REST::Response { globalparameter = restReturnParam:REST::Parameter{name=rpn,
                                                                                     type = restReturnType:REST::Type {}

                                                                                     }
                                                                                                                      }
                                                                                 };

                enforce domain soa operation:SOA::Operation { parameter = soaParam:SOA::Parameter { name = rpn,
                   direction=SOA::ParameterDirection::output,
                   type = soaReturnType:SOA::Type {}
                  }
                                                                                     };

                when { SimpleType2SimpleType(restReturnType, soaReturnType) or ComplexType2Class(restReturnType, soaReturnType);
                       RequestParam2InputParam(method, operation);
                      }

    }

top relation SimpleType2SimpleType {
                tn:String;
                checkonly domain rest rst:REST::SimpleType { name = tn, container = rp:REST::RESTPackage {}};
```

```
        enforce domain soa sst:SOA::SimpleType { name = tn, container = sp:SOA::SOAPackage {}
                                                                            }    ;
        when {Package2Package(rp, sp);}
    }

top relation ComplexType2Class {

        tn: String;

        checkonly domain rest ct:REST::ComplexType { name = tn, container = rp:REST::RESTPackage {} };

        enforce domain soa c:SOA::Class { name = tn, container = sp:SOA::SOAPackage {}
                                                                    };

        when {Package2Package(rp, sp);}
        where { ComplexProperty2ClassProperty(ct, c);
                SimpleProperty2SimpleType(ct, c);
                }

    }

    relation ComplexProperty2ClassProperty {

        checkonly domain rest ct:REST::ComplexType { property = p:REST::Property { type = t:REST::ComplexType {}}};

        enforce domain soa c:SOA::Class {};


        where { ComplexProperty2ClassPropertyDo(ct, c); }
     }

    relation ComplexProperty2ClassPropertyDo {

        pn:String;
        nestedT:REST::ComplexType;
        nestedC:SOA::Class;

        checkonly domain rest ct:REST::ComplexType { property = rp:REST::Property { name= pn, type = nestedT}};

        enforce domain soa c:SOA::Class { property = sp:SOA::Property {name= pn, type = nestedC}};

        when {ComplexType2Class(nestedT, nestedC);
                }
    }
```

```
        relation SimpleProperty2SimpleType {

                checkonly domain rest ct:REST::ComplexType { property = p:REST::Property { type = t:REST::SimpleType {}}};

                enforce domain soa c:SOA::Class {};


                where { SimpleProperty2SimplePropertyDo(ct, c); }
         }

        relation SimpleProperty2SimplePropertyDo {

                pn: String;
                restNestedT:REST::SimpleType;
                soaNestedT:SOA::SimpleType;

                checkonly domain rest ct:REST::ComplexType { property = rp:REST::Property { name = pn, type = restNestedT}};

                enforce domain soa c:SOA::Class { property = sp:SOA::Property {name = pn, type = soaNestedT}};

                when {SimpleType2SimpleType(restNestedT, soaNestedT);
                           }
        }


}
```

A.2. SOA to RESTful.

```
transformation SOA2REST(soa:SOA, rest:REST) {

        top relation Package2Package {

                checkonly domain soa o:SOA::SOAPackage {};

                enforce domain rest r:REST::RESTPackage { };

                }

        top relation Interface2Service {

                sp:SOA::SOAPackage;
                rp:REST::RESTPackage;

                checkonly domain soa i:SOA::Interface { container= sp};
```

```
        enforce domain rest  rs:REST::RESTService { container =rp, URI=uri:REST::URI {}};

         when { Package2Package(sp, rp); }
         where { GETOperation2Service (i, rs);
                 POSTOperation2Service (i, rs);
                 PUTOperation2Service (i, rs);
                 DELETEOperation2Service (i, rs);
                }

        }

 relation GETOperation2Service  {

        on, cn:String;

        checkonly domain soa  i:SOA::Interface { operation = o:SOA::Operation { name = on,
                                                                                 method= SOA::http_methods::get
                                                                               },
                                                  interfaceRealization= ir:SOA::InterfaceRealization { className= cn }
                                                };

        enforce domain rest  rs:REST::RESTService{ method= m:REST::Method_GET { className= cn}   } ;

        where {InParameter2Request(o, m);
               OutParameter2Response(o, m); }
}

relation POSTOperation2Service  {

        on, cn:String;


        checkonly domain soa  i:SOA::Interface { operation = o:SOA::Operation { name = on,
                                                                                 method= SOA::http_methods::pos
                                                                                },
                                                  interfaceRealization= ir:SOA::InterfaceRealization { className= cn }
                                                 };

        enforce domain rest  rs:REST::RESTService{ method= m:REST::Method_POST { className=cn} } ;

        where { InParameter2Request(o, m);
                OutParameter2Response(o, m); }
}


relation PUTOperation2Service  {
```

```
        on, cn:String;

        checkonly domain soa  i:SOA::Interface { operation = o:SOA::Operation { name = on,
                                                                     method= SOA::http_methods::put
                                                                    },
                                        interfaceRealization= ir:SOA::InterfaceRealization { className= cn }
                                       };

        enforce domain rest  rs:REST::RESTService{ method= m:REST::Method_PUT { className=cn} } ;

        where { InParameter2Request(o, m);
                OutParameter2Response(o, m); }
}

    relation DELETEOperation2Service {

        on, cn:String;

        checkonly domain soa  i:SOA::Interface { operation = o:SOA::Operation { name = on,
                                                                     method= SOA::http_methods::delete
                                                                    },
                                        interfaceRealization= ir:SOA::InterfaceRealization { className= cn }
                                       };

        enforce domain rest  rs:REST::RESTService{ method= m:REST::Method_DELETE { className=cn}        } ;

        where { InParameter2Request(o, m);
                OutParameter2Response(o, m); }
}

    relation InParameter2Request {

        on, pn:String;
        t1:SOA::Type;
        t2:REST::Type;

        checkonly domain soa o:SOA::Operation { name = on,
                                       parameter = ps:SOA::Parameter { name=pn,
                                                                    direction=SOA::ParameterDirection::input,
                                                                    type = t1
                                                                   }
```

```
                                        };
        enforce domain  rest m:REST::Method { name= on,
                                        request=req:REST::Request { globalparameter = pr:REST::Parameter { name =pn,
                                                                                                           type = t2
                                                                                                          }
                                                                   }
                                       };

    when {SimpleType2SimpleType(t1,t2) or Class2ComplexType(t1,t2);  }
}


    relation OutParameter2Response {

        on, pn:String;
        t1:SOA::Type;
        t2:REST::Type;

        checkonly domain soa o:SOA::Operation { name = on,
                                         parameter = ps:SOA::Parameter { name=pn,
                                                                         direction=SOA::ParameterDirection::output,
                                                                         type = t1}
                                        };
        enforce domain rest   m:REST::Method {  name= on,
                                         response = res:REST::Response { globalparameter = pr:REST::Parameter { name =pn,
                                                                                                                 type = t2
                                                                                                                }
                                                                         }
                                        };

    when { SimpleType2SimpleType(t1,t2) or Class2ComplexType(t1,t2);
         }
}


    top relation SimpleType2SimpleType {

        tn:String;
        sp:SOA::SOAPackage;
        rp:REST::RESTPackage;

        checkonly domain soa sst:SOA::SimpleType { name = tn, container =sp };

        enforce domain rest rst:REST::SimpleType { name = tn, container= rp, representation= x:REST::XMLRepresentation{container =rp,
                                                                                                              xsdElement =tn}
                                          };
```

```
        when { Package2Package(sp, rp); }
}



 top relation Class2ComplexType {

        cn: String;
        sp:SOA::SOAPackage;
        rp:REST::RESTPackage;

        checkonly domain soa c:SOA::Class { name = cn, container =sp  };


        enforce domain rest ct:REST::ComplexType { name = cn, container = rp,
                                            representation=x:REST::XMLRepresentation{container=rp, xsdElement =cn } };

        when { Package2Package(sp, rp); }

        where { ClassProperty2ComplexProperty(c, ct);
                SimpleType2SimpleProperty(c, ct);
               }

}



relation ClassProperty2ComplexProperty {

        checkonly domain soa c:SOA::Class {property = p:SOA::Property { type = t:SOA::Class {}}};

        enforce domain rest ct:REST::ComplexType { };

        where { ClassProperty2ComplexPropertyDo(c, ct); }
 }

relation ClassProperty2ComplexPropertyDo {

        cn:String;
        nestedT:REST::ComplexType;
        nestedC:SOA::Class;

        checkonly domain soa c:SOA::Class { property = sp:SOA::Property {name= cn, type = nestedC}};

        enforce domain rest ct:REST::ComplexType { property = rp:REST::Property { name= cn, type = nestedT}};
```

```
        when {Class2ComplexType(nestedC, nestedT);
                }
    }

    relation SimpleType2SimpleProperty {

        checkonly domain soa c:SOA::Class { property = p:SOA::Property { type = t:SOA::SimpleType {}} };

        enforce domain rest ct:REST::ComplexType { };

        where { SimpleProperty2SimplePropertyDo(c, ct); }
     }

    relation SimpleProperty2SimplePropertyDo {

        cn: String;
        restNestedT:REST::SimpleType;
        soaNestedT:SOA::SimpleType;

        enforce domain soa c:SOA::Class { property = sp:SOA::Property {name = cn, type = soaNestedT}};

        checkonly domain rest ct:REST::ComplexType { property = rp:REST::Property { name = cn, type = restNestedT}};

        when {SimpleType2SimpleType(soaNestedT, restNestedT);
                 }
    }

}
```

A.3. SOAP to SOA.

```
transformation WSDL2SOA(wsdl:WSDL, soa:SOA) {


    top relation Package2Package {


        checkonly domain wsdl wp:WSDL::WSDLPackage { xmlSchema= xmlS:WSDL::XMLSchema {  } };

        enforce domain soa sp:SOA::SOAPackage { };

        where { ChooseSimpleType2SimpleType(wp, sp);
                ChooseComplexType2Class(wp, sp);
```

```
               }
}

top relation Service2Interface {

       cn, pn:String;

       checkonly domain wsdl w:WSDL::WSDLDefinition { container= spp:WSDL::WSDLPackage {},
                                              definition_service = s:WSDL::Service { service_port= p:WSDL::Port { className= cn,
                                                                                                             binding=
                                                                                 b:WSDL::Binding { type = pt:WSDL::PortType {name=pn}
                                                                                                        }
                                                                                         }

                                                                          }
                                             };
       enforce domain soa sp:SOA::SOAPackage { type= i:SOA::Interface { name = pn },
                                        interfaceRealization= ir:SOA::InterfaceRealization {className = cn, interface = i }
                                       };

       when { Package2Package (spp, sp); }

       where { Operation2Operation(pt, i); }
}


relation Operation2Operation {

       on:String;

       checkonly domain wsdl pt:WSDL::PortType { operation= spo:WSDL::Operation { name= on,
                                                                          request= reqM:WSDL::Message{},
                                                                          response= resM:WSDL::Message {}
                                                                         }
                                        };
       enforce domain soa i:SOA::Interface { operation=  so:SOA::Operation { name = on } };

        where { RequestMessage2InputParam (spo, so);
                ResponseMessage2OutputParam (spo, so);
               }

}


relation RequestMessage2InputParam {
```

```
            en:String;
            soapT:WSDL::XMLType;
            soaT:SOA::Type;

            checkonly domain wsdl spo:WSDL::Operation { request= reqM:WSDL::Message { part = soapP:WSDL::Part{ element = e1:WSDL::Element {
                                                                          name=en,
                                                                          type = ct:WSDL::ComplexType { element= e2:WSDL::Element {type=soapT}
                                                                                                         }
                                                                                                                                        }
                                                                                                                 }
                                                                                            }
                                                          };


           enforce domain soa sp:SOA::Operation { parameter = soaP:SOA::Parameter { name = en,
                                                                                direction=SOA::ParameterDirection::input,
                                                                                      type=soaT
                                                                                    }
                                                  };

           when { SimpleType2SimpleType(soapT, soaT) or ComplexType2Class(soapT, soaT); }

      }



relation ResponseMessage2OutputParam {

            en:String;
            soapT:WSDL::XMLType;
            soaT:SOA::Type;

            checkonly domain wsdl spo:WSDL::Operation { response= reqM:WSDL::Message { part = soapP:WSDL::Part{ element = e1:WSDL::Element {
                                                                    name = en,
                                                                          type = ct:WSDL::ComplexType { element= e2:WSDL::Element {type=soapT}
                                                                                                         }
                                                                                                                                         }
                                                                                                                   }
                                                                                            }
                                                          };


            enforce domain soa sp:SOA::Operation { parameter = soaP:SOA::Parameter { name = en,
                                                                                      direction=SOA::ParameterDirection::output,
                                                                                      type=soaT
                                                                                    }
                                                 };
```

```
         when { SimpleType2SimpleType(soapT, soaT) or ComplexType2Class(soapT, soaT); }

}

relation ChooseSimpleType2SimpleType {


        checkonly domain wsdl wp:WSDL::WSDLPackage { xmlSchema= xmlS:WSDL::XMLSchema { xmlType = stsoap:WSDL::SimpleType {} } };

        enforce domain soa sp:SOA::SOAPackage{ type= stsoa:SOA::SimpleType {} };

        where {SimpleType2SimpleType(stsoap, stsoa); }
}


relation ChooseComplexType2Class {

        checkonly domain wsdl wp:WSDL::WSDLPackage { xmlSchema= xmlS:WSDL::XMLSchema { xmlType = dt:WSDL::ComplexType { } }
                                                                                                    };

        enforce domain soa sp:SOA::SOAPackage{ type= c:SOA::Class {}
                                                                                    };

        where { ComplexType2Class(dt, c); }  }

relation SimpleType2SimpleType {

        tn:String;

        checkonly domain wsdl wst:WSDL::SimpleType { name = tn   };

        enforce domain soa sst:SOA::SimpleType { name = tn   }     ;
}

relation ComplexType2Class {

        tn:String;

        checkonly domain wsdl soapct:WSDL::ComplexType { name = tn };

        enforce domain soa sc:SOA::Class { name = tn}      ;

        where { ComplexElement2ClassProperty(soapct, sc);
                      SimpleElement2SimpleProperty(soapct, sc);
                       }
```

```
}
relation ComplexElement2ClassProperty {

        checkonly domain wsdl ct:WSDL::ComplexType { element = e:WSDL::Element { type = nestedCT:WSDL::ComplexType {}}};

        enforce domain soa c:SOA::Class {};


        where { ComplexElement2ClassPropertyDo(ct, c); }
 }

relation ComplexElement2ClassPropertyDo  {

        en:String;
        nestedT:WSDL::ComplexType;
        nestedC:SOA::Class;

        checkonly domain wsdl ct:WSDL::ComplexType { element = e:WSDL::Element { name= en, type = nestedT}};

        enforce domain soa c:SOA::Class { property = sp:SOA::Property {name= en, type = nestedC}};

        when {ComplexType2Class(nestedT, nestedC);
                } }

relation SimpleElement2SimpleProperty {

        checkonly domain wsdl ct:WSDL::ComplexType { element = e:WSDL::Element { type = t:WSDL::SimpleType {}}};

        enforce domain soa c:SOA::Class {};


        where { SimpleElement2SimplePropertyDo(ct, c); }
 }

relation SimpleElement2SimplePropertyDo {

        pn: String;
        soapNestedT:WSDL::SimpleType;
        soaNestedT:SOA::SimpleType;

        checkonly domain wsdl ct:WSDL::ComplexType { element = e:WSDL::Element { name = pn, type = soapNestedT}};

        enforce domain soa c:SOA::Class { property = sp:SOA::Property {name = pn, type = soaNestedT}};

        when {SimpleType2SimpleType(soapNestedT, soaNestedT);
                 }
```

```
        }

}
```

A.4. SOA to SOAP.

```
transformation SOA2WSDL(soa:SOA, wsdl:WSDL) {

        top     relation Package2Package {

                cn:String;
                iName:String;

                checkonly domain soa sp:SOA::SOAPackage { interfaceRealization= ir:SOA::InterfaceRealization {className= cn},
                                                          type= i:SOA::Interface { name = iName }
                                                     };

                enforce domain wsdl wp:WSDL::WSDLPackage { xmlSchema= xmlS:WSDL::XMLSchema {},
                                                           wsdlDefinition = wsdlDef:WSDL::WSDLDefinition { definition_types= t:WSDL::Type
                                                                                                   { type_schema= xmlS},
                                                                                                     definition_service= s:WSDL::Service {
                                                                                                                  name= iName+'GeneratedSOAPService',
                                                                                                                  iName+'GeneratedSOAPPort',
                                                                                                                  soapAddress= sa:WSDL::SOAPAddress {
                                                                                                                  location='GeneratedSOAPPortLocation'
                                                                                                                                                  },
                                                                                                                  className = cn
                                                                                                                                        }

                                                                                                      },
                                                            definition_portType= pt:WSDL::PortType { name= iName+'GeneratedPortType'},
                                                            definition_binding= b:WSDL::Binding { name= iName+'GeneratedSOAPBinding',
                                                                                                    type= pt,
                                                                                                     soapBinding = soapBinding:WSDL::SOAPBinding { }
                                                                                                  }
                                                                                                          }
                                                        };

                where {  ChooseSimpleType2SimpleType(sp, xmlS);
                         ChooseClass2ComplexType(sp, xmlS);
                         GenerateRequestWrappers(i, xmlS);
                         GenerateResponseWrappers(i, xmlS);
                         Operation2RequestMessages(i, wsdlDef);
                         Operation2ResponseMessages(i, wsdlDef);
```

```
                Operation2Operation(i, pt);
                Operation2SOAPOperation(i, soapBinding);
              }
}

relation GenerateRequestWrappers {

    on, en:String;
    soaT:SOA::Type;
    soapT:WSDL::XMLType;


    checkonly domain soa i:SOA::Interface { operation = sp:SOA::Operation { name= on,
                                                                            parameter = soaP:SOA::Parameter { name = en,
                                                                                               direction=SOA::ParameterDirection::input,
                                                                                               type=soaT
                                                                                                              }
                                                                         }
                                            };

     enforce domain wsdl schema:WSDL::XMLSchema { xmlType= wrapperCT:WSDL::ComplexType { name=on,
                                                                                          element= e1:WSDL::Element { name = en,
                                                                                                                       type = soapT
                                                                                                                     }
                                                                                          },
                                                                                          element = wrapper:WSDL::Element { name = on,
                                                                                                                             type= wrapperCT
                                                                                                                           }
                                                                                          }
                                                 };
      when {  SimpleType2SimpleType(soaT, soapT) or Class2ComplexType(soaT, soapT); }

}


relation GenerateResponseWrappers {

        on, en:String;
        soaT:SOA::Type;
        soapT:WSDL::XMLType;


        checkonly domain soa i:SOA::Interface { operation = sp:SOA::Operation {  name= on,
                                                                                 parameter = soaP:SOA::Parameter { name = en,
                                                                                                direction=SOA::ParameterDirection::output,
                                                                                                type=soaT
```

```
                                                                                                    }
                                                                               }
                                                    };

      enforce domain wsdl schema:WSDL::XMLSchema { xmlType= wrapperCT:WSDL::ComplexType { name=on+'Response',
                                                                                           element= e1:WSDL::Element { name = en,
                                                                                                                       type = soapT
                                                                                                                     }
                                                                                          },
                                                       element = wrapper:WSDL::Element { name = on+'Response',
                                                                                          type= wrapperCT }
                                                 };

      when {  SimpleType2SimpleType(soaT, soapT) or Class2ComplexType(soaT, soapT); }

}

relation Operation2RequestMessages {

      on:String;

      checkonly domain soa i:SOA::Interface { operation = sp:SOA::Operation { name= on
                                                                                                                    }
                                                                        }           ;

      enforce domain wsdl wsdlDef:WSDL::WSDLDefinition { messages = m:WSDL::Message { name=on,
                                                                                     part= p:WSDL::Part {name=on}
                                                                                    }
                                                        };
}

relation Operation2ResponseMessages {

      on:String;

      checkonly domain soa i:SOA::Interface { operation = sp:SOA::Operation { name= on } };

      enforce domain wsdl wsdlDef:WSDL::WSDLDefinition { messages = m:WSDL::Message { name=on+'Response',
                                                                                      part= p:WSDL::Part { name= on+'Response' }
                                                                                    }
                                                            };

}
```

```
    relation Operation2Operation {

            on:String;

            checkonly domain soa i:SOA::Interface { operation=  so:SOA::Operation { name = on } };

            enforce domain wsdl pt:WSDL::PortType { operation= spo:WSDL::Operation { name= on } };


    }

    relation Operation2SOAPOperation {

            on:String;

            checkonly domain soa i:SOA::Interface { operation=  so:SOA::Operation { name = on } };
            enforce domain wsdl sb:WSDL::SOAPBinding { soapOperation= spo:WSDL::SOAPOperation { operationName= on } };

}

    relation ChooseSimpleType2SimpleType {

            checkonly domain soa sp:SOA::SOAPackage{ type= stsoa:SOA::SimpleType {} };

            enforce domain wsdl xmlS:WSDL::XMLSchema { xmlType = stsoap:WSDL::SimpleType {} } ;

            where {SimpleType2SimpleType(stsoa, stsoap); }
    }

    relation ChooseClass2ComplexType {

            checkonly domain soa sp:SOA::SOAPackage{ type= c:SOA::Class {} };

            enforce domain wsdl xmlS:WSDL::XMLSchema { xmlType = ct:WSDL::ComplexType {} } ;

            where { Class2ComplexType(c, ct); }
    }

    relation SimpleType2SimpleType {

            tn:String;
```

```
        checkonly domain soa sst:SOA::SimpleType { name = tn   };

        enforce domain wsdl wst:WSDL::SimpleType { name = tn   };


}


relation Class2ComplexType {

        tn:String;

        checkonly domain soa c:SOA::Class { name = tn} ;

        enforce domain wsdl ct:WSDL::ComplexType { name = tn };

        where { ClassProperty2ComplexElement(c, ct);
                SimpleProperty2SimpleElement(c, ct);
              }

}


relation ClassProperty2ComplexElement {

        checkonly domain soa c:SOA::Class { property = p:SOA::Property { type = nestedC:SOA::Class {} } };

        enforce domain wsdl ct:WSDL::ComplexType { };

        where { ClassProperty2ComplexElementDo(c, ct); }
 }

relation ClassProperty2ComplexElementDo  {

        pn:String;
        soaNestedC:SOA::Class;
        soapNestedT:WSDL::ComplexType;

        checkonly domain soa c:SOA::Class { property = sp:SOA::Property {name= pn, type = soaNestedC}};

        enforce domain wsdl ct:WSDL::ComplexType { element = e:WSDL::Element { name= pn , type = soapNestedT}};

        when { Class2ComplexType( soaNestedC, soapNestedT);  }
}
```

```
	relation SimpleProperty2SimpleElement {

		checkonly domain soa c:SOA::Class {property = p:SOA::Property { type = sts:SOA::SimpleType {}
																		}};
		enforce domain wsdl ct:WSDL::ComplexType { };

		where { SimpleProperty2SimpleElementDo(c, ct); }
	 }

	relation SimpleProperty2SimpleElementDo {

		pn: String;
		soapNestedT:WSDL::SimpleType;
		soaNestedT:SOA::SimpleType;

		checkonly domain soa c:SOA::Class { property = sp:SOA::Property {name = pn, type = soaNestedT}};

		enforce domain wsdl ct:WSDL::ComplexType { element = e:WSDL::Element { name = pn, type = soapNestedT}};

		when {SimpleType2SimpleType(soaNestedT, soapNestedT); }
	}
}
```