UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Automatización de la reconfiguración dinámica de servicios Cloud

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Carlos Cano Genovés

Tutor: Dr. Emilio Insfrán Pelozo

2015-2016

[Página intencionalmente en blanco]

Agradecimientos

A mi tutor Emilio Insfrán por todo el apoyo, consejos y ayuda prestada en este trabajo.

A Miguel Ángel por las cosas que me ha enseñado, los consejos, la ayuda prestada y porque sin él este trabajo no sería posible.

A los compañeros del grupo ISSI que me han ayudado a resolver algunos problemas.

Y por último pero no por ello menos importante a mi familia por todo el apoyo mostrado.

[Página intencionalmente en blanco]

Resumen

Debido al auge de las tecnologías de cloud computing existe una gran demanda de servicios cloud por lo que el uso de metodologías de desarrollo ágiles es muy habitual. En este tipo de metodologías se promueve el desarrollo incremental del software lo que implica una continua integración de nuevos servicios a los ya existentes, siendo necesarios procesos y herramientas bien definidos que faciliten, y en lo posible, automaticen esta tarea. En este Trabajo de Fin de Grado se sigue una enfoque dirigido por modelos para la generación automática de la orquestación de servicios cloud y también de los scripts necesarios que permitan actualizar los enlaces entre los servicios actuales y los nuevos produciendo por tanto una reconfiguración arquitectónica de los servicios cloud en tiempo de ejecución. El trabajo se ha realizado en el contexto del grupo de investigación ISSI (DSIC-UPV) que cuenta actualmente con el *Microsoft Azure Research Award* por lo que hará uso de la plataforma Windows Azure© con WCF Workflow para la orquestación de servicios. Esta plataforma emplea archivos XML (eXtensible Markup Language) para su configuración por lo que será además necesario el uso de XDT (XML Document Transform) para poder modificar las configuraciones actuales de la plataforma a las nuevas configuraciones.

Palabras clave: servicios cloud, desarrollo dirigido por modelos, integración, desarrollo ágil e incremental, arquitectura cloud

Abstract

With the rise of cloud computing technologies there is a high demand for cloud services so that the use of agile development methodologies is very common. This type of methodologies promotes incremental development which implies a continuous integration of new services to the existing ones. This task requires well-defined processes and tools to facilitate, and where possible to automate this continuous integration. This Final Degree Work follows a model-driven development approach for the automatic generation of the orchestration of cloud services and the necessary scripts for updating the links among the existing and new services thus producing a run-time architectural reconfiguration of cloud services. The work was done in the context of the ISSI research group (DSIC-UPV) that currently holds the *Microsoft Azure Research Award* so we will use the Windows Azure© platform with the WCF Workflow for service orchestration. This platform uses XML files (eXtensible Markup Language) for its settings so it will be necessary to use XDT (XML Document Transform) to modify the current settings of the platform to the new settings.

Keywords: cloud services, model driven development, integration, agile development and incremental development, cloud architecture

Tabla de contenidos

1. Introducción

1.1. Contexto

Hoy en día el desarrollo de aplicaciones en la nube (*cloud*) está en auge debido a las prestaciones que brindan permitiendo a las empresas ofrecer sus servicios mediante la web, escalarlos cuando sea necesario y pagar únicamente por aquellos recursos que se utilicen, reduciendo así los costes y aumentando su disponibilidad a los clientes.

Debido al continuo mantenimiento de las aplicaciones y la adición de nuevas funcionalidades es necesario que las aplicaciones sigan un paradigma de *desarrollo continuo*, donde la aplicación se va desarrollando de forma continua a lo largo del tiempo, de *integración continua*, donde la aplicación se desarrolla por incrementos y de *despliegue continuo* donde la nueva aplicación se debe desplegar tras cada integración.

Con tal de agilizar este proceso de desarrollo continuo y de integración continua se ha desarrollado un método **Dynamic Incremental Architectural Reconfiguration** (DIARy) [45][44] el cual facilita la gestión de los incrementos de la aplicación, la generación de los esqueletos de las clases y la automatización la nueva configuración necesaria para el despliegue de los nuevos incrementos.

El método DIARy utiliza un paradigma de Desarrollo de Software Dirigido por Modelos (DSDM) con tal de gestionar los distintos incrementos mediante el empleo de modelos y de transformaciones de modelo a modelo, así como automatizar la generación de código y de configuraciones a través de transformaciones de modelo a texto.

Actualmente el método DIARy está siendo aplicado para el desarrollo integración y despliegue de servicios *cloud* en la plataforma Azure, ya que es la plataforma de la que disponemos, pero en un futuro se pretende dar soporte a más plataformas.

1.2. Objetivo

En este trabajo de fin de grado se pretende implementar una parte del método DIARy relacionada con la generación de la configuración necesaria para desplegar aplicaciones en entornos *cloud*, con tal motivo el objetivo principal es **diseñar e implementar un entorno de desarrollo** que permita automatizar dos tareas importantes del método DIARy:

- Una transformación de tipo modelo a modelo (M2M) que utiliza un metamodelo extensión de SoaML (explicado más adelante) y un metamodelo específico de las plataformas *cloud*.

  El propósito de esta transformación es pasar de un modelo orientado a la arquitectura de servicios a un modelo más específico de las plataformas *cloud*. Para poder llevar a cabo esta tarea es necesario diseñar los metamodelos necesarios que representen la arquitectura correctamente con todos los datos necesarios, corroborar que los metamodelos son correctos, implementar una transformación empleando el lenguaje ATL para realizar la transformación entre modelos y realizar pruebas para demostrar que todo funciona correctamente.

- Una transformación de tipo modelo a texto (M2T) que utiliza el metamodelo específico de plataformas *cloud* y la configuración necesaria para desplegar el proyecto en la plataforma *cloud* Microsoft Azure.

  El propósito de esta tarea es generar la nueva configuración de la aplicación para poder desplegar la aplicación *cloud* tras la integración del nuevo incremento. Para realizar esta tarea es necesario comprender adecuadamente el metalenguaje XDT empleado por Microsoft Azure para poder generar la configuración. Además será necesario implementar una transformación empleando el lenguaje Acceleo para la transformación de modelo a texto y realizar pruebas para corroborar que todo funciona adecuadamente.

1.3. Trabajos relacionados

Hoy en día la cantidad de métodos que han sido desarrollados específicamente para ayudar con el desarrollo de aplicaciones *cloud* es muy reducido, ya que *cloud* es una tecnología bastante nueva y por lo tanto carece de mucho soporte.

Otra de las cosas que cabe destacar de las tecnologías *cloud* es que a pesar de que se han establecido unos estándares para el desarrollo los principales proveedores de servicios *cloud* (Google, Microsoft, Amazon, etc.) no implementan estos estándares y utilizan unos estándares propios por lo que la migración de una aplicación *cloud* de un proveedor a otro puede ser algo costoso y por lo tanto es necesario tener en cuenta el proveedor que se va a utilizar a la hora de desarrollar la aplicación.

Algunas de las metodologías que se han desarrollado con tal de dar soporte al desarrollo de aplicaciones *cloud* son las siguientes:

MULTICLAPP

MULTICLAPP [12] es un *framework* de desarrollo para aplicaciones *multilcloud* que permite a los desarrolladores generar aplicaciones *cloud* independientes de la plataforma donde cada uno de los componentes de la aplicación puede estar en un proveedor distinto.

Este *framework* está dividido en tres fases:

- Fase modelado: En esta fase los arquitectos y desarrolladores modelar la aplicación *multicloud* de una forma gráfica sencilla, identificando los artefactos *cloud* que componen la aplicación. Emplea un *profile UML* de SoaML (Service Oriented Architecture Model Language) como metamodelo para modelar.
- Fase de codificación funcional del código: En esta fase se genera el código de la aplicación en Java (actualmente MULTICLAPP solo da soporte para Java) junto con Maven para las dependencias necesarias. Para realizar esta transformación de modelo a texto se emplea JET (Java Emitter Templates).

- Fase de generación de artefactos *cloud* y despliegue: En esta fase se genera y configura los artefactos *cloud* y genera unos adaptadores que permiten integrar el artefacto *cloud* en la plataforma deseada.

Como puede observarse, esta metodología tiene muchas características en común con DIARy:

- Empleo de MDA (Model Driven Architecture) para la automatización de las tareas.
- Empleo de un *profile UML* extendido de SoaML
- Permite el desarrollo de aplicaciones *multicloud* (aplicaciones donde varios componentes pueden estar en *clouds* distintos que pueden pertenecer a proveedores distintos).
- Dan soporte durante todo el desarrollo de la aplicación, desde la generación de la arquitectura hasta el despliegue pasando por la generación de código y artefactos *cloud*.

Pese a que MULTICLAPP tiene muchos aspectos en común con DIARy existen algunas diferencias.

La principal diferencia entre DIARy y MULTICLAPP es que DIARy da soporte al paradigma de desarrollo ágil, permitiendo que la aplicación se desarrolle por incrementos y teniendo las ventajas de un desarrollo ágil que son el desarrollo continuo, la integración continua y el despliegue continuo. Debido a que MULTICLAPP no da soporte a este paradigma de desarrollo ágil no trabaja con incrementos y por lo tanto no tiene una de las características más interesantes de DIARy, la **automatización de la reconfiguración dinámica de la arquitectura**.

TOSCA (Topology and Orchestration Specification for Cloud Applications)

TOSCA [43] es una metodología cuyo objetivo es ofrecer portabilidad y gestión de aplicaciones *cloud*. TOSCA se basa principalmente en el modelado de la arquitectura de la aplicación *cloud* mediante el uso del metalenguaje de XML (eXtensible Markup Language).

TOSCA está orientado al paradigma de desarrollo *DevOps*. *DevOps* [42] es un conjunto de valores y prácticas que dirigen estas nuevas prácticas en las organizaciones. *DevOps* tiene como principio la automatización completa de los procesos desde el entorno de desarrollo (*Dev*) hasta el entorno de operaciones (*Ops*), esto se logra con la implementación de herramientas que automatizan los procesos e incrementan su confiabilidad [38].

Las principales características que TOSCA tiene en común con DIARy son las siguientes:

- Empleo de MDA para la gestión de la arquitectura de la aplicación
- Automatización del despliegue y la reconfiguración de la aplicación
- *DevOps* es un paradigma de desarrollo ágil por lo que implementa desarrollo continuo, integración continua y despliegue ágil, así como también trabaja por incrementos.

La principal diferencia entre DIARy y TOSCA radica en que TOSCA está orientado principalmente a la parte *Ops* de *DevOps*, es decir, TOSCA está dirigido principalmente al despliegue y reconfiguración de la arquitectura de la aplicación *cloud*. DIARy a parte del despliegue y reconfiguración también automatiza parte de la generación de código (parte *Dev* de *DevOps*).

1.4. Estructura del documento

El contenido de este trabajo está estructurado de la siguiente manera:

La primera sección corresponde con la introducción, en la que se habla del contexto del proyecto, así como de los objetivos que se han propuesto en el proyecto y de los trabajos relacionados.

En la sección dos se indica el marco tecnológico del proyecto, así como también se explica las distintas herramientas y tecnologías que se han utilizado para el desarrollo de este.

A continuación, en la sección tres se explica los distintos metamodelos empleados en el proyecto, cuáles son las correspondencias entre ellos y las transformaciones realizadas.

En la sección cuatro se presenta un caso de estudio donde se ha aplicado el método DIARy mostrando todo el proceso necesario para realizar la reconfiguración dinámica de la arquitectura.

La sección cinco corresponde con las conclusiones del trabajo, el trabajo futuro donde se indican las ampliaciones que podrían realizarse en futuros proyectos y las publicaciones que se han realizado en el transcurso de este trabajo de fin de grado.

Para finalizar, la última sección muestra las referencias utilizadas para poder realizar este trabajo.

2. Marco Tecnológico

2.1. Ingeniería basada en modelos (MDE)

La ingeniería basada en modelos (MDE Model Driven Engineering) es una disciplina de la ingeniería del software que se basa en modelos como artefactos de primera clase y que tiene como objetivo desarrollar, mantener y evolucionar software por medio de transformaciones de modelo. El proceso de desarrollo bajo MDE es llamado Desarrollo de Software Dirigido por Modelos (DSDM)[4].

MDE ofrece un enfoque más efectivo; los modelos son partes activas del proceso de desarrollo de software. Los modelos son abstractos y formales, al mismo tiempo. La abstracción no destaca aquí por su vaguedad, pero para la esencia de la compacidad. Los modelos MDE tienen el significado exacto del código del programa, en el sentido de que la mayor parte de la aplicación final, no sólo la clase y los esqueletos de los métodos pueden ser generados a partir de ellos [40]. En este caso los modelos ya no son sólo la documentación, sino partes del software, lo que constituye un factor decisivo para aumentar la velocidad y calidad de desarrollo de software.

Un enfoque basado en modelos requiere lenguajes para la especificación de modelos, definición de transformación y descripción del metamodelo. MDE propone el uso de transformaciones de modelos con el fin de transformar un modelo en otro, y también para producir el producto final.

MDE es aceptado por diversas organizaciones y empresas que incluyen a la OMG, IBM y Microsoft.

Existen cinco cosas que una infraestructura de soporte MDE debe definir:

- Conceptos disponibles para la creación de modelos y reglas claras que rigen su uso.
- La notación a utilizar para describir los modelos.
- Tiene que ser claro, cómo los elementos del modelo representan los elementos del mundo real y los artefactos de software.

- Conceptos para facilitar las extensiones de usuarios dinámicos para modelar conceptos, modelos de notación y los modelos creados a partir de ellos.
- Conceptos para facilitar el intercambio de conceptos de modelos y notación, y los modelos creados a partir de conceptos definidos por el usuario para facilitar las asignaciones de los modelos a otros artefactos.

2.1.1. Desarrollo de software dirigido por modelos (DSDM)

El Desarrollo de Software Dirigido por Modelos (DSDM) es un enfoque de desarrollo de software basado en modelos. Un modelo puede definir la funcionalidad, estructura o comportamiento de un sistema. Los modelos permiten trabajar con un nivel de abstracción más cerca a los conceptos de dominio, en lugar de centrarse en conceptos orientados a la plataforma como el desarrollo del software tradicional. El objetivo de esta propuesta es maximizar la productividad, incrementar la interoperabilidad entre sistemas, facilitando la reutilización y adaptación del cambio tecnológico.

Además cualquier artefacto software es considerado un modelo o un elemento del modelo. Mientras que los enfoques anteriores utilizan modelos para documentación o la comunicación de ideas, que son entidades de primera clase durante todo el ciclo de vida ingeniería todo basado en modelos [7]. DSDM es una aproximación abierta e integradora no vinculada a una norma especial, por lo tanto, existen diferentes implementaciones.

El DSDM intenta capturar lo que con frecuencia se expresa de manera informal como especificaciones formales basadas en modelos. Los conceptos clave para mitigar el problema actual de la ingeniería de software son modelos, metamodelos, espacios tecnológicos, lenguajes de modelado de dominio específico y diferentes tipos de transformaciones de modelo como modelo a modelo o modelo a texto.

2.1.2. Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA), como se ha dicho antes, está incluido en la definición de MDE. El estándar MDA desde el Object Management Group (OMG) es una implementación específica de la ingeniería dirigida por modelos (MDE).

MDA es un joven modelo establecido por la OMG. La OMG se fundó en el año 1989 y es actualmente un consorcio abierto de aproximadamente 800 empresas en el mundo. La OMG crea especificaciones independientes de fabricantes para mejorar la interoperabilidad y portabilidad de sistemas de software.

MDA está relacionado al uso de lenguajes de modelado como lenguajes de programación más que meramente como lenguajes de diseño. La programación con lenguajes de modelado puede mejorar la calidad y la velocidad de desarrollo del software.

El objetivo de MDA es separar la forma en que los sistemas de aplicación se definen a partir de la tecnología en la que se ejecutan.

MDA inicia con la idea bien conocida y establecida desde hace mucho tiempo de separar las especificaciones operacionales del sistema de los detalles de la forma en que los sistema utilizan las capacidades de su plataforma.

La independencia de la plataforma software es análoga a la independencia de la plataforma hardware. Un lenguaje independiente de la plataforma hardware, tales como C o Java, permite la escritura de una especificación que se puede ejecutar en una amplia variedad de plataformas de hardware con ningún cambio. De la misma forma que un lenguaje software independiente de la plataforma permite la escritura de una especificación que puede ejecutarse en una amplia variedad de plataformas de software, o los diseños de la arquitectura software, sin tener que realizar ningún cambio. Por lo tanto, un software independiente de la plataforma podría ser asignado a un multiprocesador o entorno multitarea CORBA, o un ambiente cliente-servidor de base de datos relacional, sin cambio en el modelo.

En general, la organización y procesamiento de datos implicada un modelo conceptual que puede no ser igual a la organización de los datos y el procesamiento de la implementación. Si se consideran dos conceptos, los del "cliente" y "cuenta", modelarlos como clases, utilizando UML se sugiere que la solución de software debe expresarse en términos de clases de software llamados Cliente y Cuenta. Sin embargo, hay muchos diseños de software posibles que pueden cumplir con estos requisitos, muchas de las cuales ni siquiera están orientados a objetos.

Entre el concepto y la aplicación, un atributo puede ser un referente, una clase puede dividirse en conjuntos de instancias de objetos de acuerdo con algunos criterios de clasificación, las clases se pueden combinar o dividir; el estado de gráficos puede ser aplanado, combinado, o separado, y así sucesivamente. Un lenguaje de modelado que permite este tipo de asignaciones es un software independiente de la plataforma.

Elevando el nivel de abstracción cambia la plataforma de la cual depende cada capa de abstracciones. El desarrollo basado en modelos se basa en la construcción de modelos que son independientes de sus plataformas de software, que incluyen entornos cliente servidor de base de datos relacional y la estructura de código final [24].

MDA proporciona un enfoque, y permite que las herramientas sean provistas para:

- La especificación de un sistema independiente de la plataforma que lo soporta.
- La especificación de las plataformas.
- La elección de una plataforma en particular para el sistema.
- Y finalmente, la transformación de la especificación del sistema en una particular plataforma.

Los tres objetivos principales de MDA son portabilidad, interoperabilidad y reutilización mediante separación arquitectónica de diversos puntos de vista.

El ciclo de vida de desarrollo de MDA, no parece muy diferente del ciclo de vida tradicional. Se identifican las mismas fases. Una de las principales diferencias se encuentra en la naturaleza de los artefactos que se crean durante el proceso de desarrollo. Los artefactos son modelos formales, es decir, los modelos que pueden ser entendidos por las computadoras [20].

Como conclusión MDA es una aproximación al desarrollo de sistemas, que aumenta la potencia de los modelos en ese trabajo. Está basado en modelos porque proporciona un medio para que el uso de modelos para dirigir el curso del entendimiento, diseño, construcción, implementación, operación, mantenimiento y modificación.

2.1.2.1. Puntos de vista MDA

Un punto de vista sobre un sistema es una técnica para la abstracción utilizando un conjunto seleccionado de conceptos arquitectónicos y reglas de estructuración, con el fin de centrarse en preocupaciones particulares dentro de ese sistema. La arquitectura basada en modelos especifica tres puntos de vista sobre un sistema, un punto de vista independiente de cómputo, un punto de vista independiente de plataforma y un punto de vista específico de plataforma [27].

Punto de Vista Independiente de Cómputo

El punto de vista independiente de cómputo se centra en el entorno del sistema y los requisitos para el sistema; los detalles de la estructura y procesamiento están ocultos o aún indeterminados.

Punto de Vista Independiente de Plataforma

El punto de vista independiente de plataforma se centra en el funcionamiento de un sistema al tiempo que oculta los detalles necesarios para una determinada plataforma. Una vista independiente de plataforma muestra parte de la especificación completa que no cambia de una plataforma a otra. Una vista independiente de la plataforma puede utilizar un lenguaje de modelado de propósito general o un lenguaje específico para el área en la cual se utilizará el sistema.

Punto de Vista Específico de Plataforma

El punto de vista específico de plataforma combina el punto de vista independiente de plataforma con un enfoque adicional en el detalle del uso de una plataforma específica por un sistema.

2.1.2.2. Modelos en MDA

Los artefactos de primera clase en la ingeniería basada en modelos son los modelos. La primera cosa a hacer es definir que es un modelo. Un modelo es una simplificación (o una descripción abstracta) de una parte del mundo llamado sistema, construido con un objetivo previsto en la mente. Un modelo debería ser más fácil de usar y entender que el sistema original, debe ser capaz de responder a preguntas sobre el sistema. La respuesta proporcionada por el modelo debe ser exactamente las mismas que las dadas por el mismo sistema [39], [3].

Los modelos pueden consistir en un conjunto de elementos con una representación gráfica o textual. Mientras que esto sirve como un punto de partida, Kleppe et al. [20] da una definición aún más dirigida al DSDM. Un modelo es una descripción de un (parte de) sistema escrito en un lenguaje bien definido. Un lenguaje bien definido es un lenguaje con una forma bien definida (sintaxis), y significado (semántica), lo que es conveniente para la interpretación automatizada por un computador.

La idea de MDA es la creación de diferentes modelos de un sistema en diferentes niveles de abstracción. Cada modelo representa un aspecto determinado del sistema.

De acuerdo con estas definiciones, el código fuente es un modelo también. El código fuente es una representación simplificada de la estructura inferior de la máquina y las operaciones que son necesarias para automatizar las tareas en el mundo real. Por otra parte, el correcto código fuente es un modelo muy útil, ya que le dice a la máquina qué le indica a la maquina las medidas necesarias para mantener el objetivo del sistema.

Tipos de modelos

El estándar MDA [27] define cuatro tipos de modelos en el ciclo de vida del desarrollo de software dirigido por modelos:

Modelo Independiente de Cómputo (CIM)

Un modelo independiente de cómputo es una vista del punto de vista independiente de cómputo, que se centra en el entorno y en los requisitos del sistema; los detalles estructurales o de procesamiento del sistema son ocultados o todavía no son determinados. Un CIM no muestra los detalles de la estructura del sistema. Un CIM algunas veces es llamado un modelo de dominio y un vocabulario que es familiar para los profesionales del dominio en cuestión y utilizan en su especificación [27]. Se centra en el entorno del sistema y en los requisitos que el usuario tiene en el sistema, la descripción proporciona lo que se espera que el sistema debe hacer.

Modelo Independiente de Plataforma (PIM)

El modelo independiente de plataforma se centra en el funcionamiento de un sistema al mismo tiempo ocultando los detalles necesarios para una determinada plataforma. Un PIM exhibe un determinado grado de independencia de la plataforma con el fin de ser aptos para su uso con un número de diferentes plataformas de tipo similar. También es una representación de la funcionalidad y comportamiento del negocio, sin distorsión por los detalles de la tecnología. Además, muestra la parte de la especificación completa que no cambia desde una plataforma a otra.

El objetivo es posponer el proceso de desarrollo de creación de modelos que tengan en cuenta los aspectos tecnológicos de una plataforma tanto como posible. La principal ventaja es ser capaz de reaccionar eficientemente y con bajos costos a los cambios de tecnología.

Modelo Específico de Plataforma (PSM)

Un modelo específico de plataforma es una combinación de un PIM con el detalle adicional de la plataforma específica del sistema.

Modelo de Plataforma

Finalmente los modelos de plataforma son la representación de conceptos técnicos de partes de la plataforma, los servicios prestados por esa plataforma y para su uso en un posterior PSM, los conceptos que modela el uso de la plataforma por las aplicaciones.

2.1.2.3. Metamodelos

La palabra "meta" es griega y significa "arriba", por lo tanto el término metamodelo puede interpretarse como un modelo que describe otro modelo. Un lenguaje consiste en palabras cuya combinación es restringida por una gramática. Si una sentencia en un lenguaje es vista como un posible modelo, la definición de su estructura, la gramática puede ser vista como su metamodelo. Anteriormente se dijo que en el DSDM un metamodelo define como un modelo puede ser visto, esto puede ser formulado de manera más precisa como: un metamodelo define los constructores y las reglas que se utilizan para crear una clase de modelos.

Esto es consistente con la siguiente definición:

"*Un metamodelo es un modelo de un conjunto de modelos* [26]."

"*Un metamodelo es un modelo que define el lenguaje para expresar un modelo* [37]."

Un metamodelo es un modelo de especificación que describe sus modelos en un cierto lenguaje de modelado. Un metamodelo dice lo que se puede expresar en un modelo válido del lenguaje de modelado.

La interpretación de un metamodelo es el mapeo de elementos del metamodelo a elementos del lenguaje de modelado. El valor de verdad de las declaraciones

en el metamodelo puede determinarse por cualquier modelo expresado en el lenguaje de modelado. Puesto que el metamodelo es la especificación de modelos, un modelo en el lenguaje de modelado es válido sólo si ninguna de estas afirmaciones son falsas. Los metamodelos precisos son un requisito previo para la realización de transformación de modelos automáticas y para definir modelos precisos.

Puesto que un metamodelo es también un modelo, podría ser expresado en algunos lenguajes de modelado. Un metamodelo para un lenguaje de modelado podría utilizar el mismo lenguaje de modelado. Las afirmaciones en el metamodelo se expresan en el mismo lenguaje que está siendo descrito por el metamodelo. Esto es llamado metamodelo reflexivo.

2.1.2.4. Meta-Object Facility (MOF)

Meta-Object Facility (MOF) es un estándar propuesto y definido por la OMG [34] para soportar a MDA. Esta norma propone cuatro niveles de arquitectura metamodelos con cuatro meta capas como se muestra en la **Figura 1**. Cada meta capa es el metamodelo de los constructores en las capas anteriores. Las capas se describen a continuación:

1. **Capa M3 Metametamodelo**: En esta capa reside el metametamodelo, un lenguaje para definir los metamodelos del nivel M2. En el estándar de OMG, MOF es el lenguaje definido en esta capa.
2. **Capa M2 Metamodelo**: Los metamodelos M2 se utilizan para describir los modelos M1. El metamodelo de UML de la OMG describirá los constructores de UML.
3. **Capa M1 Modelo**: En este nivel es donde se definen los modelos. Un modelo es una instancia de un metamodelo M2. Es decir, si en la capa M2 reside el metamodelo de UML, en M1 podríamos tener uno de sus modelos: diagrama de clases, diagrama de actividad, diagrama de secuencia, y así sucesivamente.
4. **Capa Mo Instancia**: En esta capa se definen objetos del mundo real.

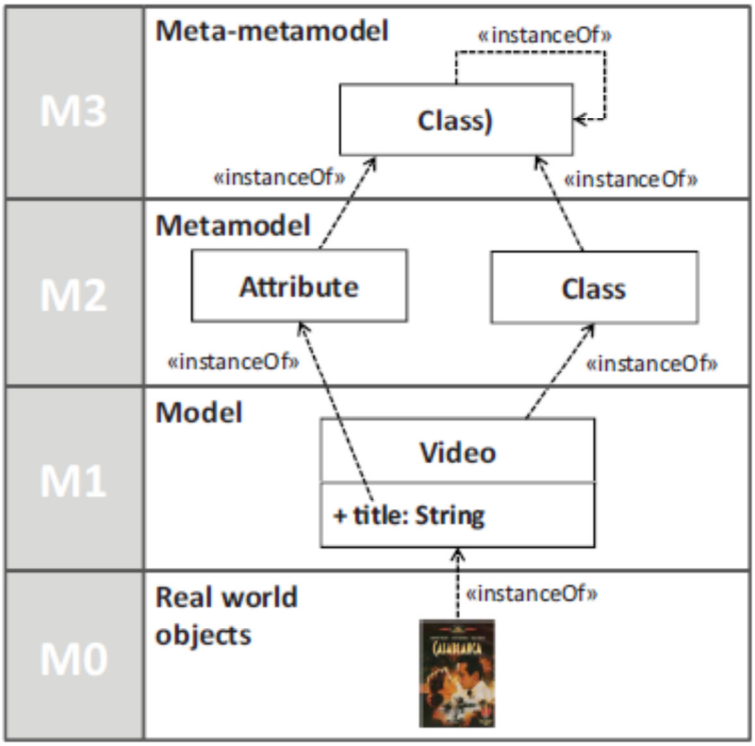

Figura 1: Capas de Arquitectura MOF [4].

MOF es una arquitectura de metamodelado cerrada. Esto significa que el nivel M3 podría definirse con instancias de elementos de M3. Esto implica que podemos definir MOF.

Además MOF proporciona conceptos para definir un lenguaje:

- Clases, que son metaobjetos del modelo MOF.
- Tipos de datos (propiedad), que son necesarios para el modelo de datos descriptivos (es decir, los tipos primitivos).
- Asociaciones (propiedad), que son las relaciones binarias entre los metaobjectos del modelo.
- Paquetes, que son modularizan los modelos.

2.1.2.5. Transformación de Modelos

Transformación de modelos: es el proceso de conversión de un modelo a otro modelo del mismo sistema [27]. Un modelo puede ser transformado a varios modelos alternativos que pueden mantener la semántica, pero con diferente sintaxis.

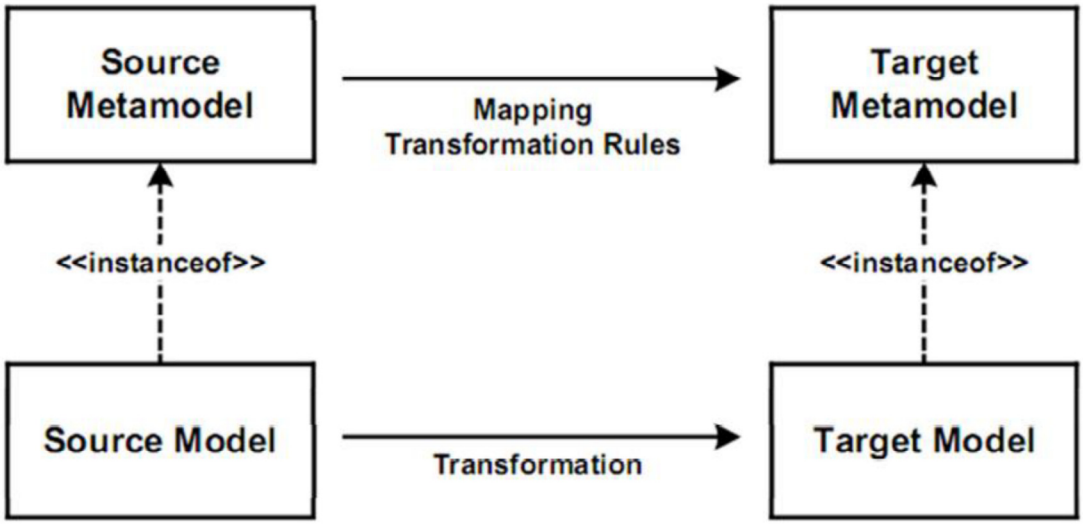

Figura 2: Definición de una transformación de modelos.

Una definición de transformación consiste en una colección de reglas de transformación que son especificaciones inequívocas de la manera que (parte de) un modelo puede utilizarse para crear una pieza de otro modelo. Las transformaciones son definidas en términos de los metamodelos involucrados en el proceso de transformación.

Una transformación, una definición de transformación y sus reglas de transformación pueden definirse de la siguiente manera [16]:

- Una transformación es la generación automática de un modelo destino desde un modelo de fuente, según una definición de transformación.

- Una definición de transformación es un conjunto de reglas de transformación que juntas describen cómo un modelo origen puede ser transformado en un modelo destino.
- Una regla de transformación es una descripción de cómo una o más constructores en la lenguaje origen pueden ser transformados en uno o más constructores de un lenguaje destino.

La característica más importante de una transformación es el hecho de que una transformación de modelos debe mantener el significado entre el modelo origen y el modelo destino. En este punto hay que decir que el significado del modelo sólo puede ser preservado y expresado en el modelo origen y destino. Parte de la información pueden perderse si el lenguaje de destino es menos expresivo que el lenguaje de origen.

Un mapeo se define como una transformación unidireccional en contraste a una relación que define una transformación bidireccional.

2.1.2.6. Lenguaje de restricción de objetos (OCL)

Lenguaje de restricciones de objetos (OCL) [33] es un lenguaje formal usado para describir las expresiones de modelos UML. Estas expresiones suelen especificar condiciones invariables que se deben mantener en el modelado del sistema o consultas sobre los objetos descritos en un modelo. Tenga en cuenta que cuando OCL evalúa las expresiones, que no tienen efectos secundarios; es decir, su evaluación no puede alterar el estado de ejecución del correspondiente sistema.

Las expresiones OCL pueden ser usadas para especificar operaciones/acciones que, cuando son ejecutadas, no alteraran el estado del sistema. Los modeladores UML pueden utilizar OCL para especificar las restricciones específicas de la aplicación en sus modelos. Los modeladores UML pueden también utilizar OCL para especificar las preguntas sobre el modelo de UML, que son totalmente lenguaje de programación independiente.

OCL no sólo se puede aplicar en modelos UML, pero también puede ser aplicado en UML o metamodelos MOF ya que están expresados en UML o un subconjunto de UML.

Por lo tanto OCL puede ser utilizado para restringir la semántica del metamodelo, por ejemplo, por medio de estereotipos o lenguajes de dominio específicos (LDE).

En el contexto MDA, OCL se puede utilizar de tres formas:

- Modelos precisos en M1 a nivel MOF.
- Definición de lenguajes de modelado.
- Definición de transformaciones.

2.1.2.7. ATL (Atlas Transformation Language)

Es un lenguaje de dominio específico para la especificación de transformaciones de un modelo a otro [1], [17], [18]. ATL está inspirado en los requerimientos del estándar QVT [31] de la OMG, y se basa en el formalismo de OCL [33].

La decisión de utilizar OCL está motivada por su amplia adopción en MDE y el hecho de que es un lenguaje estándar apoyado por OMG y de las principales herramientas de los proveedores.

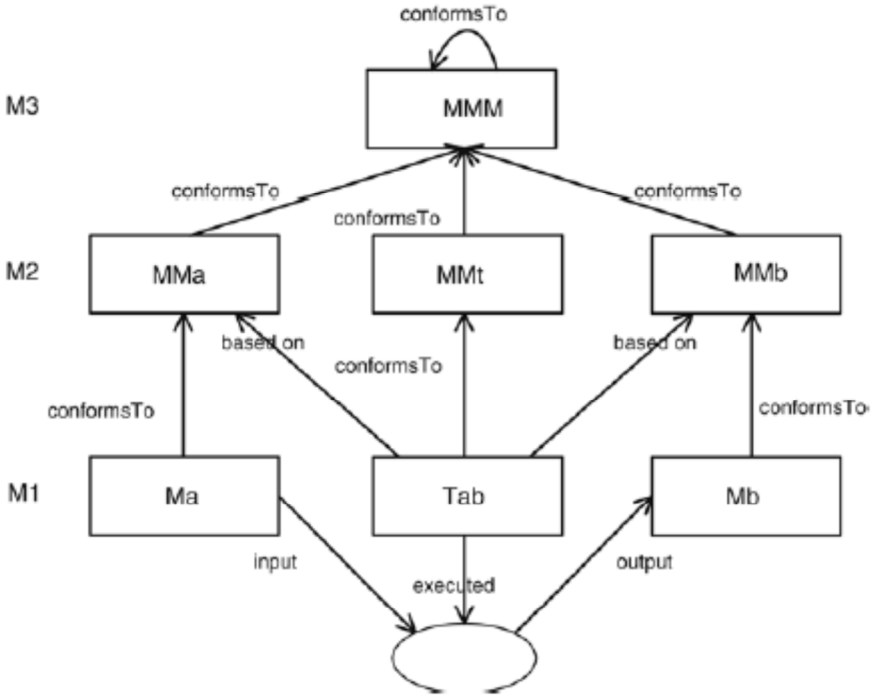

Figura 3: Patrón de transformación de modelos.

ATL es aplicado en el contexto del patrón de transformación que se muestra en la **Figura 3**. Este patrón es un modelo origen *Ma* que se transforma en un modelo destino *Mb* según una definición de transformación *mma2mmb.atl* escrita en el lenguaje ATL. La definición de transformación es un modelo conforme al metamodelo de ATL [17]. Todos los metamodelos son conformes a MOF.

ATL es un lenguaje híbrido, es decir, que ofrece una mezcla de construcciones declarativas e imperativas. El estilo declarativo de especificación de transformación tiene un número de ventajas. Generalmente está basado en la especificación de las relaciones entre los patrones origen y destino, y por lo tanto tiende a estar más cerca de la forma en que los desarrolladores perciben intuitivamente una transformación [17].

Las transformaciones ATL son unidireccionales, es decir, operan en modelos origen sólo para lectura y en modelos de destino para escritura. Durante la ejecución de una transformación, el modelo origen puede ser navegado, pero los cambios en él no están permitidos, y en un modelo destino es imposible ser navegado [17]. Una transformación bidireccional se implementa como un par de transformaciones: una para cada dirección. Los modelos origen y destino de

ATL pueden expresarse en el formato de serialización XMI de la OMG. Los metamodelos origen y destino también pueden expresarse en XMI o en la notación más conveniente [17].

Las definiciones de transformación en ATL se realizan en módulos (*module*). Un módulo contiene una sección obligatoria de cabecera (*header*), una sección de importación (*import*), y un número de ayudantes (*helpers*), y reglas de transformación (transformation rules). La sección de cabecera da el nombre al módulo de transformación y declara los modelos origen y destino. A continuación se detalla un ejemplo de la sección de cabecera:

module Class2Relational;

create OUT : Relational from IN : Class;

La sección ***header*** comienza con la palabra clave *module* seguida por el nombre del módulo. Luego, los modelos origen y destino se declaran como variables introducidas por sus metamodelos [17]. La palabra clave *create* indica los modelos de destino. La palabra clave *from* indica los modelos origen. En este ejemplo, el modelo destino junto a la variable *OUT* es creado del modelo origen *IN*. Los modelos origen y destino se ajustan a los metamodelos *Class* y *Relational* respectivamente. En general, más de un modelo origen y destino pueden ser enumerados en la sección de cabecera [17].

Un ***helper*** define las operaciones en el contexto de los elementos de un modelo o en el contexto de un módulo. Ellos pueden tener parámetros de entrada y pueden utilizar la recursividad. Un *helper attribute* se utiliza para asociar valores de sólo lectura nombrados en los elementos de un modelo. De igual forma las operaciones tienen un nombre, un contexto y un tipo. La diferencia es que ellas no pueden tener parámetros de entrada. Sus valores se especifican mediante una expresión OCL. Para ilustrar la sintaxis de los *helpers* se presenta el siguiente ejemplo [17]:

helper context String def: firstToLower() : String =

self.substring(1, 1).toLower() + self.substring(2, self.size());

Las reglas de transformación es la construcción básica en ATL que se utiliza para expresar la lógica de transformación. Las reglas ATL pueden especificarse en un estilo declarativo o en un estilo imperativo [17].

Matched Rules: Las *matched rules* son reglas ATL declarativas. Una *matched rule* se compone de un patrón de origen y un patrón de destino. La regla del patrón de origen regla especifica un conjunto de tipos de destino (procedentes de los metamodelos de origen y desde el conjunto de tipos disponibles de colecciones en OCL) y un *guard* (una expresión OCL booleana). Un patrón de origen se evalúa a un conjunto de coincidencias en el modelo origen [17].

El patrón de destino se compone de un conjunto de elementos. Cada elemento especifica un tipo de destino (del metamodelo destino) y un conjunto de enlaces. Un enlace se refiere a una característica del tipo de destino (es decir, un atributo, una referencia, o un extremo de la asociación) y especifica una expresión de inicialización para el valor de característica [17]. El siguiente fragmento de código muestra una simple *matched rule*. Esta regla implementa la lógica para la transformación de las clases a las tablas.

```
rule Class2Table
        {
                from c : Class!Class to out : Relational !Table
                        (
                                name <- c.name,
                                col <- Sequence {key}->union(c.attr->select(e
                                | not e .multiValued)),
                                key <- Set {key}
                        ),
                key : Relational!Column
                        (
                                name <- 'objectId',
                                type <- thisModule.objectIdType
                        )
        }
```

Tipos de *mathed rules*: Hay varios tipos de *matched rules* que difieren en la forma en que se desencadenan [17].

- ***Standard rule*** son aplicadas una vez para todas las combinaciones que se puedan encontrar en los modelos origen [17].
- ***Lazy rule*** son provocados por otras reglas. Se aplican en una sola combinación tantas veces como es referida por otras reglas, cada vez que produzca un nuevo conjunto de elementos de destino [17].
- ***Unique lazy rule*** también son provocados por otras reglas. Se aplican una sola vez por una combinación determinada. Si una *unique lazy rule*

se activa más tarde en la misma combinación, esta utiliza los elementos de destino ya creados [17].

Herencia de reglas. En ATL, la herencia de reglas se puede utilizar como un mecanismo de reutilización de código, y también como un mecanismo para especificar reglas polimórficas [17].

Una regla (llamada subregla) puede heredar de otra regla (regla padre). Una subregla coincide con un subconjunto de los combinaciones de la regla padre. Los tipos patrón de origen de la regla padre podrán ser sustituidos por sus subtipos en el patrón de origen de la subregla [17].

ATL tiene una parte **imperativa** en base a dos conceptos principales:

- ***Called rules***: Una *called rule* es básicamente un procedimiento: se invoca por su nombre y puede tomar argumentos [17].

- ***Action block***: Un *action block* es una secuencia de sentencias imperativas, y se puede utilizar en lugar de o en una combinación con un patrón de destino en combinación o en *called rules* [17].

Las declaraciones imperativas disponibles en ATL son las construcciones conocidas para la especificación de un flujo de control, tales como las condiciones, bucles, asignaciones, etc [17].

2.1.2.8. Transformaciones de Modelo a Texto (M2T)

MDA coloca el modelado en el centro del proceso de desarrollo de software. Varios modelos son utilizados para capturar diversos aspectos del sistema de una manera independiente de la plataforma [32]. Luego, se aplican conjuntos de transformaciones a estos modelos independientes de plataforma (PIM) para derivar modelos específicos de la plataforma (PSM). Estos PSMs necesitan ser finalmente transformados en artefactos de software como código, especificaciones de implementación, informes, documentos, etc [32].

El estándar *MOF model to text* (*mof2text*) aborda cómo traducir un modelo a varios artefactos de texto como código, especificaciones de implementación, informes, documentos, etc [32]. En esencia, el estándar *mof2text* debe abordar la manera de transformar un modelo en una representación de texto. Una forma intuitiva de abordar este requerimiento es un enfoque basado en plantillas (*templates*) en la que el texto que se generen a partir de los modelos se especifica como un conjunto de plantillas de texto que se configuran con los elementos del modelo [32].

Un enfoque basado en plantillas es usado donde un *template* específica una plantilla de texto con marcadores de posición para los datos que se extraen de los modelos. Estos marcadores de posición son esencialmente expresiones especificadas sobre las entidades del metamodelo con consultas, siendo los principales mecanismos para la selección y la extracción de los valores de los modelos. Estos valores se convierten en fragmentos de texto usando un lenguaje de expresión aumentado con una biblioteca de manipulación de cadenas [32].

Un *template* puede estar compuesto para abordar requerimientos de complejas transformaciones. Las grandes transformaciones pueden ser estructurados dentro de módulos (*module*) que tienen partes públicas y privadas [32].

Por ejemplo, la **Figura 4** presenta la especificación de un *template* para generar una definición de Java para una clase UML [32].

```
[template public classToJava(c : Class)]
class [c.name/]
{
    // Constructor
    [c.name/]()
    {
    }
}
[/template]
```

Figura 4: Plantilla de generación de código [32].

Un *template* puede invocar a otros *templates*. La invocación de un template es equivalente a to *in situ* del texto producido por el template que se invoca [32].

```
[template public classToJava(c : Class)]
class [c.name/]
{
    // Attribute declarations
    [attributeToJava(c.attribute)/]

    // Constructor
    [c.name/]()
    {
    }
}
[/template]

[template public attributeToJava(a : Attribute)]
[a.type.name/] [a.name/];
[/template]
```

Figura 5: Plantilla con invocación a otras plantillas de generación de código [29].

La **Figura 5**, presenta el *template classToJava* que invoca al *template attributeToJava* para cada atributo de la clase y pone ';' como separador entre

los fragmentos de textos producidos por cada invocación del *template attributeToJava* [32]. Un *template* puede iterar sobre una colección mediante el uso de un bloque *for*, por ejemplo *[for]..[/for]* [32].

Las navegaciones sobre modelo complejo pueden ser especificadas mediante *queries*. El siguiente ejemplo en la **Figura 6** muestra el uso de un *query allOperations* para recoger las operaciones de todas las clases padres abstractas de una clase en una jerarquía de clases [32].

```
[query public allOperations(c: Class) : Set ( Operation ) =
c.operation->union( c.superClass->select(sc|sc.isAbstract=true)-
>iterate(ac : Class;
         os:Set(Operation) = Set{}| os->union(allOperations(ac))) /]

[template public classToJava(c : Class) ? (c.isAbstract = false)]
class [c.name/]
{
// Attribute declarations
   [attributeToJava(c.attribute)/]
   // Constructor
   [c.name/]()
   {
   }
   [operationToJava(allOperations(c))/]
}

[/template]

[template public operationToJava(o : Operation)]
[o.type.name/] [o.name/] ([for(p:Parameter | o.parameter)
separator(',')] [p.type/] [p.name/] [/for]);
[/template]
```

Figura 6: *Query* en la generación de texto [32].

2.2. Espacios Tecnológicos

En esta sección se describen las áreas tecnológicas relacionadas con el trabajo.

Eclipse

La comunidad Eclipse [41] es una comunidad de código abierto cuyos proyectos se centran en la creación de una plataforma de desarrollo abierta compuesta por marcos extensibles, herramientas para la creación, implementación y gestión de software a través del ciclo de vida.

Uno de sus proyectos más importantes es Eclipse. Eclipse es un proyecto cogido abierto, robusto, con múltiples características, plataforma industrial de calidad comercial para el desarrollo de herramientas altamente integradas.

Entorno de desarrollo integrado (IDE) utiliza diferentes módulos para mejorar su funcionalidad; Esto es una ventaja sobre otros ambientes monolíticos donde la funcionalidad no es configurable.

En definitiva, la naturaleza de esta herramienta es un IDE abierto y ampliable para múltiples propósitos. Este trabajo será de particular interés el Eclipse Modeling Framework (EMF), un marco para la gestión de modelos y generación de código de modelos que se describen en XMI. EMF es descrito en detalle en el apartado siguiente.

Eclipse Modeling Framework (EMF)

El proyecto EMF es un marco de modelado y generación de código para la creación de herramientas y otras aplicaciones basadas en modelos de datos estructurados. Todo inicia con una especificación del modelo descrito en XMI. EMF proporciona las herramientas necesarias para producir un conjunto de clases Java para el modelo.

Los modelos se pueden especificar utilizando Java anotado, documentos XML, o las herramientas de modelado como Rational Rose a través del cual pueden ser importados a EMF. Lo más importante es que EMF proporciona la base para

establecer la interoperabilidad con otras herramientas y aplicaciones basadas en EMF. Con respecto a la relación de EMF a la OMG y MOF, EMF empezó como una implementación de la especificación madura MOF de la experiencia adquirida en el desarrollo de herramientas por los desarrolladores de Eclipse. EMF puede verse como una implementación eficiente de utilización conjunta de la API de MOF. Sin embargo, para evitar confusión, el metamodelo de EMF está basado en el núcleo de MOF y es llamado Ecore [41].

ATL - una tecnología de transformación de modelos

ATL es un componente del proyecto MMT [28] que tiene como objetivo proporcionar un conjunto de herramientas de transformación de modelo a modelo. Estos incluyen algunas transformaciones ATL, que muestran un motor de transformación ATL y un IDE para ATL [1].

ATL (Atlas Transformation Language) es un lenguaje de transformación de modelos y kit de herramientas [1]. En el campo de la Ingeniería Dirigida por Modelos (MDE), ATL proporciona formas para producir un conjunto de modelos destino de un conjunto de modelos origen. Desarrollado sobre la plataforma Eclipse, proporciona una serie de herramientas de desarrollo estándar (resaltado de sintaxis, depurador, etc.) que tiene como objetivo facilitar el desarrollo de las transformaciones ATL [1].

Acceleo

Acceleo es una implementación pragmática del estándar *MOF Model to Text Language (MTL)* del *Object Management Group (OMG)*. Acceleo es el resultado de varios años de investigación y desarrollo. Se inició en la empresa francesa Obeo [30], la unión entre el estándar MTL OMG, y la experiencia de su equipo con la generación de código industrial y los últimos avances en el campo de investigación M2T, ofrecen ventajas excepcionales: alta capacidad personalización, interoperabilidad, gestión de la trazabilidad, etc [29].

2.3. Arquitectura Orientada a Servicios

La computación orientada a servicios (*Service Oriented Computing*) es una nueva generación de plataformas para la computación distribuida. Esta incluye muchos conceptos, como su propio paradigma de diseño (orientación a servicios) que ofrece normas y directrices para realizar ciertas características de diseño en una aplicación. La computación orientada a servicios tiene sus propios principios y patrones de diseño, un propio modelo de arquitectura que es conocida como Arquitectura Orientada a Servicios, junto a muchos conceptos, tecnologías y marcos de trabajo asociados. El término computación orientada a servicios y arquitectura orientada a servicios no son sinónimos, pero la literatura muy a menudo no siempre los trata como a uno [6].

La computación orientada a servicios es altamente afectada por diferentes paradigmas y tecnologías tales como: orientada a objetos, computación distribuida, gestión de procesos de negocios, servicios Web, e integración de aplicaciones empresariales (EAI). La computación orientada a servicios es guiada por numerosos estándares de la industria. Tanto por organizaciones de normalización (incluyendo OMG, W3C, OASIS y WS-I) y organizaciones de la industria (tales como Microsoft, IBM, Sun Microsystems, Oracle, Hewlett-Packard, Fujitsu-Siemens y SAP) han desarrollado activamente la computación orientada a servicios, y están en constante desarrollo de nuevas normas y especificaciones [6], [19].

2.3.1. Definición

La arquitectura orientada a servicios (*Service Oriented Architecture*) es un modelo arquitectónico que tiene como objetivo aumentar la eficiencia, agilidad y productividad de una empresa, haciendo hincapié en los servicios como el principal medio, a través del cual, la lógica del negocio es representada en soporte de la realización de los objetivos estratégicos, asociados con la computación orientada a servicios (SOC). Como una forma de arquitectura de tecnología, una implementación de SOA puede consistir en una combinación de tecnologías, productos, APIs, soportando extensiones de infraestructura, y algunas otras partes [6]. SOA también puede ser visto como un estilo de arquitectura para la construcción de aplicaciones de software que utilizan los servicios de una red [13].

Existen numerosas definiciones para SOA: Linthicum define a SOA como: "Un marco estratégico de tecnología que permite a todos los sistemas interesados, dentro y fuera de una organización, exponer y tener acceso a servicios bien definidos, e información ligada a aquellos servicios, además que pueden ser abstraídos y procesados por capas y aplicaciones compuestas para el desarrollo de soluciones [22]. En esencia, SOA añade el aspecto de la agilidad con la arquitectura, lo que nos permite tratar con cambios en el sistema utilizando una capa de configuración en vez de constantemente tener que volver a desarrollar estos sistemas" [22]. OASIS define a SOA como: "Un paradigma para organizar y utilizar capacidades distribuidas que pueden estar bajo el control de diferentes dominios de propiedad. Proporciona un medio uniforme para ofrecer, descubrir, interactuar y utilizar las capacidades para producir los efectos deseados en consonancia con las condiciones previas y expectativas medibles" [29]. Y por último Josuttis la define como: "Un paradigma arquitectónico para hacer frente a los procesos de negocio distribuidos en un amplio panorama de los sistemas heterogéneos existentes y las nuevos que se encuentran bajo el control de diferentes propietarios" [19].

SOA es principalmente para la construcción de aplicaciones de negocios [16]. Los procesos de software en SOA están estrechamente ligados al modelado de procesos de negocio, y los procesos de negocio manejan el diseño de SOA. La lógica de la aplicación de una solución SOA está débilmente acoplada y está residiendo en varios equipos que pueden estar dentro o fuera de la organización.

La sustitución y el rediseño de los componentes y la reutilización de la lógica de aplicación existente en una solución SOA debe ser más bien una tarea fácil. En SOA, los componentes se comunican a través de una única interfaz y los sistemas normalmente sirven para múltiples usuarios remotos [1]. De acuerdo con Josuttis, con el fin de establecer SOA con éxito, se necesitan tres componentes: la infraestructura, la arquitectura y los procesos. Una plataforma técnica, tales como los servicios Web son necesarios para formar la infraestructura [19]. Con base en los conceptos de SOA, estándares y herramientas, algunas decisiones tienen que hacerse para formar la arquitectura del sistema SOA. Diferentes procesos incluyen el modelado de procesos de negocio, los ciclos de vida de servicios y gobernabilidad SOA, es decir, una de las tareas más importantes en una SOA con éxito [19].

2.3.2. Elementos

La ideología fundamental detrás de SOA, es que existe un gran problema que necesita ser resuelto con el software al dividirlo en trozos más pequeños, en unidades individuales de la lógica conocida como **servicios** que están diseñados con los principios de la Computación Orientada al Servicio.

Un **servicio** es una entidad independiente que encapsula una lógica relacionada con una determinada tarea de negocio u otra agrupación lógica, e intercambia información con otros servicios para lograr la meta deseada. Un servicio puede combinarse con otros servicios con el fin de crear composiciones de servicios. Los servicios pueden ser distribuidos, lo que significa que ellos pueden residir dentro o fuera de la empresa.

Los servicios existen autónomamente pero no están aislados, mantienen un grado de uniformidad y estandarización [1], [6]. Cuando se utilizan los servicios, los servicios no se compran como un paquete de software [13].

La comunicación entre servicios requiere que los servicios estén conscientes de los otros. Esto sucede con la descripción de los servicios que contiene el nombre y la ubicación del servicio y, entradas y salidas de datos requisitos del servicio de intercambio. Los servicios son combinados libremente a través de descripciones del servicio. Se necesita un marco de comunicación para establecer la comunicación entre servicios [6]. Este marco define la estructura de los

mensajes, las políticas y protocolos que son usados en la comunicación. Como los servicios los mensajes también son autónomos. Ellos también contienen suficiente inteligencia para gobernar sus partes de la lógica de procesamiento. Después de enviar un mensaje, el remitente no es capaz de determinar, lo que va a suceder con el mensaje [1], [6].

Las **interfaces de aplicación** son los actores activos en la arquitectura, su función es iniciar y controlar todas las actividades de los sistemas de la empresa.

Las interfaces de aplicación más utilizadas son:

- **Interfaces gráficas de usuario**: Éste tipo de interfaz permite a los usuarios finales interactuar directamente con la aplicación, las interfaces gráficas pueden ser aplicaciones web o clientes ricos.
- **Programas de lotes o procesos**: Los programas o procesos de larga vida invocan su funcionalidad de manera periódica o son el resultado de acontecimientos concretos.

Sin embargo, es posible que una interfaz de aplicación delegue gran parte de la responsabilidad a servicios o procesos de negocio.

Los **servicios de contrato** proporcionan una especificación informal de la finalidad, funcionalidad, restricciones y el uso del servicio [21]. La forma de esta especificación puede variar, dependiendo del tipo de servicio. Un elemento no obligatorio de los servicios de contrato es una definición de interfaz formal basada en lenguajes como son el lenguaje de definición de Interface (IDL) o el lenguaje de descripción del servicio Web (WSDL) [21]. Estos elementos proporcionan abstracción e independencia de tecnología, incluyendo el lenguaje de programación, el protocolo de middleware de la red y su entorno de ejecución.

El contrato puede imponer la semántica detallada en las funciones y parámetros que no están sujetos a las especificaciones IDL o WSDL [21]. Es importante comprender que cada servicio requiere un servicio de contrato en particular si no hay una descripción formal basada en una norma como WSDL o IDL.

La funcionalidad de los servicios es expuesta a los clientes por el **servicio de interfaz**, los clientes deben estar conectados al servicio utilizando una red [21].

Aunque la descripción de la interfaz es parte del servicio de contrato, la implementación física de la interfaz consta de esbozos del servicio, que están incorporados en los clientes de un servicio y un despachador [21].

Los **servicios de implementación** proporcionan físicamente la lógica de negocios requerida y los datos que son apropiados. Esto es la realización técnica que cumple con servicio de contrato. El servicio de implementación consiste de uno o más artefactos como son programas, datos de configuración y bases de datos [21].

Los **servicios de lógica de negocio** son los encargados de encapsular la lógica de negocio como parte de su implementación. Ésta se encuentra disponible a través de interfaces de servicios. Sin embargo, la programación en contra de las interfaces es deseable, si se aplica un planteamiento orientado al servicio [21].

Un servicio también puede incluir datos. En particular, este es el propósito de los servicios de datos céntricos [21].

2.3.3. Principios de diseño

Varios principios de diseño pueden ser seguidos en el diseño de una solución orientada a servicios. Al diseñar un servicio orientado a solución y construyendo SOA, algunas preguntas importantes son: ¿Cómo los servicios deben ser diseñados?, ¿Cómo debe ser definida la relación entre servicios?, ¿Cómo deben diseñarse las descripciones de los servicio? y ¿Cómo deben diseñarse los mensajes?. Ocho principios de diseño de SOA se han definido [6]. No todos los principios de diseño pueden realizarse simultáneamente, porque los principios se interrelacionan. Esto significa que para lograr un principio se puede requerir la realización de otro principio. Por ejemplo, para alcanzar el acoplamiento flexible, el servicio de contrato debe estar presente. Cinco principios establecen las características de diseño servicio concreto y los tres restantes actúan más como influencias reguladoras [1], [6].

2.3.4. Abstracción de la capa de servicios

Con el fin de lograr la reutilización, no es posible crear servicios ágiles que se adaptan tanto para negocios como para aplicaciones de consideraciones lógicas simultáneamente. Por lo tanto, las capas especializadas de servicios tienen que ser construidas. Estas capas son: la *capa de servicios de aplicación*, la *capa de servicios de negocio* y la *capa de orquestación de servicios*. La **Figura 7** muestra estas capas entre los procesos de negocio y la capa de aplicación [1].

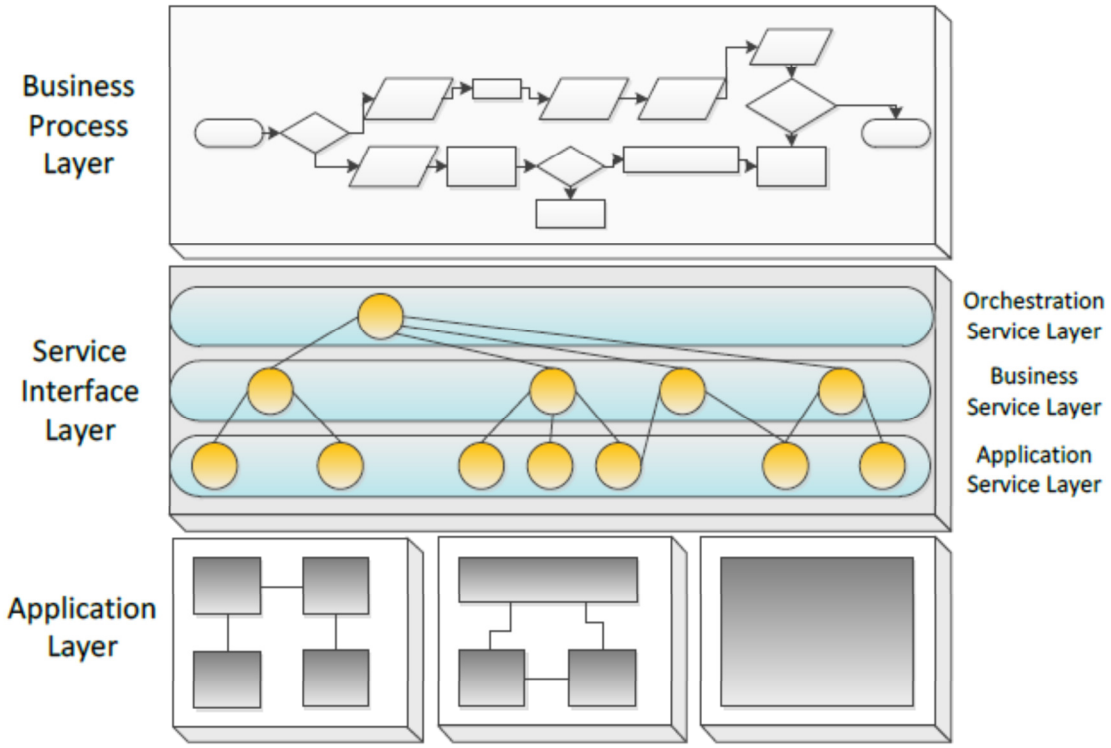

Figura 7: Capas de servicio en SOA [10].

La ***capa de servicios de aplicación*** proporciona funciones reutilizables que están relacionadas con el procesamiento de datos en un entorno de aplicaciones. Los servicios de aplicación son altamente específicos de tecnología. Los tipos de servicios de aplicaciones más comunes son *servicios de utilidad* y *servicios de envoltura* [6]. Los servicios de utilidad son servicios genéricos que

proporcionan las operaciones reutilizables para servicios de negocio permitiendo completar las tareas centrada en el negocio. Los servicios de envoltura se utilizan típicamente para los propósitos de integración. Ellos encapsulan la lógica de los sistemas heredados. Los servicios que contienen tanto aplicación como lógica del negocio se denominan *servicios híbrido*s [1].

El propósito de la ***capa de procesos de negocio*** es introducir servicios que se concentran en la representación de la lógica de negocio [1]. Los servicios de negocios siempre son implementaciones del modelo del servicio de negocio. Sin embargo, los servicios de negocio también pueden clasificarse como un *servicio de controlador* o un *servicio de utilidad*. Es muy probable que los servicios empresariales actúen como reguladores que se componen de varios servicios de aplicaciones para ejecutar su lógica de negocio.

Los modelos de servicios de negocio incluyen los modelos de negocio *centrados en las tareas* y los modelos de negocio *centrados en la entidad* [1]. Los servicios de negocio centrados en las tareas encapsulan una lógica de negocio que se específica de un determinado proceso de trabajo o negocio. Los servicios de negocios centrados en la entidad encapsulan una entidad de negocios específicos (por ejemplo, una hoja o una factura) teniendo así un mayor potencial de reutilización [1].

La ***capa de orquestación*** consiste en uno o más servicios de proceso que componen los servicios en los niveles inferiores (servicios de negocio y aplicaciones) según las reglas de negocio y la lógica empresarial que está alojada dentro de las definiciones del proceso [1].

La capa de orquestación evita la necesidad de otros servicios, administra los detalles relacionados con la interacción que se requieren para garantizar la correcta ejecución de las operaciones de servicio. Los servicios de procesos residen en la capa de orquestación y otros servicios que proporcionan conjuntos específicos de funciones, independientes de las reglas de negocio y la lógica específica del escenario a componer [1]. Los servicios de procesos pueden clasificarse como *servicios de control* porque componen otros servicios para ejecutar la lógica del proceso de negocio. Los servicios de procesos también pueden convertirse en servicios públicos si el proceso que se ejecuta puede considerarse reutilizable. Las reglas de negocio y servicios de ejecución de la lógica de la secuencia se abstraen de otros servicios de orquestación [1].

2.3.5. Servicios Web

La evolución de los servicios web puede ser vista como una fuerza conductora en la evolución del servicio de orientación, y ha dado forma considerablemente a la computación orientada al servicio. Aunque SOA como una arquitectura de tecnología y la orientación de servicio como un paradigma ambos están en una implementación neutral, los servicios Web son muy a menudo asociados con ellos [1], [19]. Con el fin de construir una solución SOA, es necesaria una plataforma de aplicación. La tecnología de los servicios Web ofrece este tipo de plataforma [1]. Los servicios Web se utilizan para implementar tareas empresariales compartidas entre las empresas, y son la tecnología que permite conectar diferentes sistemas desacoplados a través de varias plataformas, lenguajes de programación y aplicaciones. También pueden ser utilizados para la integración de aplicaciones empresariales [36]. La definición de un servicio Web es: *"Un servicio Web es una plataforma independiente, acoplamiento flexible, autónomo, programable, habilitado para aplicaciones Web, que puede ser descrito, publicado, descubierto, coordinado y configurable utilizando artefactos XML (estándares abiertos) con el fin de desarrollar aplicaciones interoperables distribuidas"* [36].

2.3.5.1. Roles de los servicios web

Los servicios Web difieren entre sí en función de las tareas que realizan y el alcance de la tarea. Un servicio Web puede ser una tarea de negocio (por ejemplo, los fondos de retiro de un servicio de depósito), un completo proceso de negocio (una compra automatizada de suministros de oficina), una aplicación (una previsión de demanda o solicitud de reposición) o puede ser un recurso que es activado por un servicio Web (por ejemplo, el acceso a una base de datos que contiene registros médicos) [36]. Las funciones de los servicios Web pueden variar desde simples peticiones para completar aplicaciones de negocio que accedan y combinan información de muchas fuentes diferentes.

Los servicios Web a veces se mezclan con las aplicaciones Web. Cuatro diferencias clave entre los servicios Web y aplicaciones Web pueden realizarse [36]:

1. Los servicios Web pueden actuar como recurso para otras aplicaciones, de manera que los servicios Web puedan utilizar otros servicios sin la intervención humana [36].

2. Un servicio Web conoce sus propiedades funcionales y no funcionales, y puede decirles a usuarios y otros servicios web. Las propiedades funcionales incluyen funciones que el servicio Web puede realizar, que entradas son requeridas para producir sus salidas [36]. Las propiedades no funcionales incluyen costos de utilización del servicio, medidas de seguridad, características de rendimiento (representación) e información de contacto [36].

3. El estado del servicio Web puede monitorizarse mediante el uso de sistemas de control y gestión de aplicaciones externas. De esta forma los servicios Web son más visibles y manejables que las aplicaciones basadas en la Web [36].

4. Los servicios Web también pueden ser negociados o subastados. Un corredor puede elegir un servicio Web conveniente para sus fines entre varios servicios realizando la misma tarea. Una opción puede basarse, por ejemplo, en el costo, rapidez y grado de seguridad del servicio Web [36].

2.3.5.2. Interfaz e implementación

La interfaz y la implementación de los servicios Web están separados. La interfaz del servicio es visible para los usuarios del servicio. Define la funcionalidad del servicio y cómo acceder a ella [6]. La interfaz también es denominada como un contrato de servicios. Consiste en la definición del WSDL (lenguaje de descripción de servicios Web) y la definición del esquema XML (eXtended Markup Language) y puede incluir definición de políticas de WS (servicio Web) [6]. Los detalles de implementación del servicio están ocultos a los usuarios del servicio. La implementación del servicio se divide en lógica de programación lógica y el procesamiento del mensaje. La lógica de programación puede ser lógica heredada que es envuelta por un servicio web o puede ser lógica que especialmente desarrollada para el servicio Web [6]. En este caso, la lógica es a menudo llamada lógica de servicio básico o lógica de negocios. Los depuradores, procesadores y agentes de servicio componen la lógica de procesamiento de mensajes de un servicio Web. Ellos pueden manejar los mensajes enviados o recibidos por los servicios Web. Algunas de estas lógicas son proporcionadas por el entorno de ejecución pero también puede ser hecha a la medida. La **Figura 8** representa los componentes de un servicio Web [6].

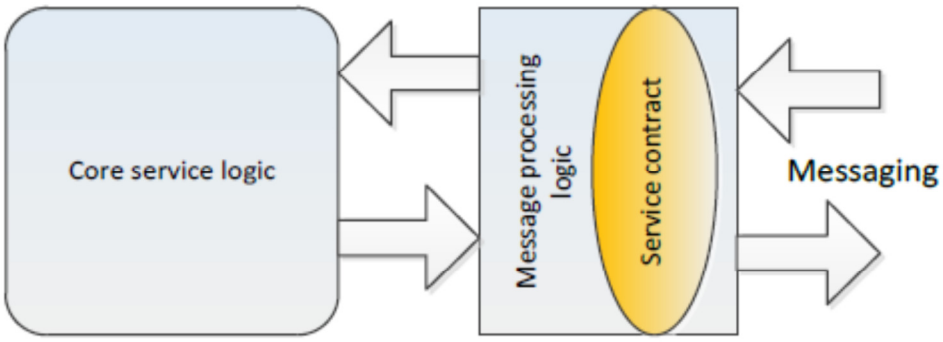

Figura 8: Componentes de un servicio Web [9].

La misma interfaz del servicio puede implementarse por diferentes prestadores de servicios mediante el uso de cualquier lenguaje de programación [36]. Las implementaciones pueden ser diferentes: una implementación puede proporcionar cierta funcionalidad mientras que otra implementación puede usar una combinación de otros servicios para proporcionar la misma

funcionalidad [36]. La separación entre la interfaz del servicio e implementación puede ser tan radical que no son organizaciones, las cuales proporcionan interfaces de servicio no son necesarias en las mismas organizaciones que implementan los servicios [36]. Diferentes proveedores de SOA han desarrollado sus plataformas para utilizar la tecnología de los servicios Web. Actuales plataformas y tecnologías industriales incluyen potentes soporte de servicios Web y XML. Estas tecnologías incluyen IBM Websphere Toolkit, Sun's Open Net Environment y JINITM technology, Microsoft .NET y Novell's One Inititatives, HP e-speak y BEA's WebLogic Integration [15].

2.3.6. Service-Oriented Architecture Modeling Language (SoaML)

La Arquitectura Orientada a Servicios (SOA) es una forma de describir y entender las organizaciones, las comunidades y los sistemas para maximizar la agilidad, la escala y la interoperabilidad. El enfoque SOA es simple: las personas, las organizaciones y los sistemas proporcionan servicios entre sí. Estos servicios nos permiten hacer algo sin hacerlo nosotros mismos o incluso sin saber cómo hacerlo; que nos permite ser más eficientes y ágiles. Los servicios también nos permiten ofrecer nuestras capacidades a otros en cambio de algún valor, estableciendo así una comunidad, proceso o mercado. El paradigma SOA funciona igual de bien para la integración de las capacidades existentes, así como la creación y la integración de nuevas capacidades [35].

SoaML (Lenguaje de Modelado de la Arquitectura Orientada a Servicio) es una especificación que proporciona un metamodelo y un perfil UML para la especificación y diseño de servicios dentro de una arquitectura orientada a servicios. Abarca extensiones de UML 2.1.2 para soportar el modelado de las siguientes capacidades [35]:

- Identificación de servicios, los requisitos que deben cumplir, y las dependencias anticipadas entre ellos.

- Especificación de servicios incluyendo las capacidades funcionales que proporcionan, capacidades que se espera que los consumidores proporcionen, los protocolos o reglas para el uso de ellos, y la información de servicio intercambiada entre los consumidores y proveedores.

- Definición de servicios de consumidores y proveedores, solicitudes y servicios que consumen y proveen, cómo están conectados y cómo las capacidades funcionales del servicio son utilizadas por los consumidores e implementadas por los proveedores de una manera compatible con los protocolos de especificación de servicio y los requisitos cumplidos.
- Las políticas para el uso y prestación de servicios.

- La habilidad para definir esquemas de clasificación teniendo aspectos para apoyar una amplia gama arquitecturas, esquemas de particionamiento físico y organizacional, y restricciones.
- Definición de servicio y requerimientos de uso de servicio, y vincularlos a metamodelos relacionados a la OMG, tales como el BMM, BPDM, UPDM o UML.

- SoaML se centra en los conceptos de modelado de servicios básicos y la intención es usar esto como una base para las extensiones relacionadas con la integración con otros metamodelos de la OMG como BPDM y BPMN 2.0, así como SBVR, OSM, ODM y otros.

El concepto clave de SoaML es el *servicio*. Un servicio es el intercambio de valor desde un participante a otro. El conocimiento de cómo usar el servicio y el acceso a utilizar es proporcionado por un contrato de servicio. En un servicio existe un proveedor de servicios y un consumidor de servicios, uno de los cuales es el iniciador de la interacción de servicios. Un servicio viene dado por un participante, y el participante puede ser cualquier entidad: un ser humano, una organización o un sistema informático.

2.3.7. Beneficios de SOA

SOA organiza los sistemas de modo que ejecuten necesidades del negocio de manera eficiente. Si fue implementado correctamente, SOA resulta una arquitectura empresarial que es ágil y reutilizable.

Esto significa que se puede utilizar una y otra vez la misma lógica de aplicación y los procesos de negocio pueden cambiarse sin los cambios necesarios en todos los sistemas [1], [6]. En la **Figura 9**, se muestra tanto el modelo tradicional y el modelo de la automatización de un proceso de negocios SOA.

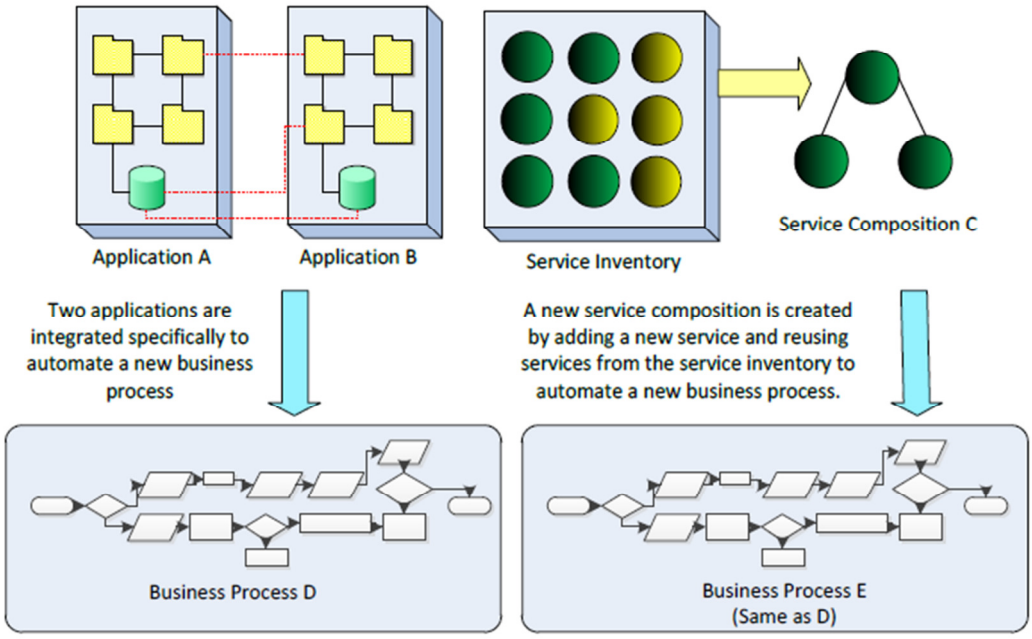

Figura 9: Modelo Tradicional vs. Modelo SOA [9].

Para automatizar tradicionalmente los procesos de negocio D, se integran dos aplicaciones con varias conexiones punto a punto. Esto es problemático si el proceso del negocio cambia y aparece la necesidad de nuevas funcionalidades en el futuro. Como resultado, habrá un sistema que es difícil de administrar. Con el fin de automatizar los procesos de negocio E (que son los mismo que el D) pueden utilizarse los servicios existentes encontrados en inventario de servicios [1], [6].

La cantidad de nuevo código escrito es probablemente menos que en el modelo tradicional tan sólo unas poca cantidad de nueva funcionalidad es necesario ser creado. Como resultado, a la cuestión de la reacción rápida a los cambios en la actividad empresarial se ha cumplido con el sistema con el que es menos complicado de gestionar [1], [6].

Es importante tener en cuenta que SOA no es una arquitectura técnica compuesta por servicios Web, o simplemente implementar los servicios web no conduce a todos los beneficios que pueden lograrse a través de una verdadera SOA. En su lugar, SOA requiere un cambio de lógica de negocios en el que se ve desde un contexto orientado hacia el servicio [1], [6]. Los factores claves para del éxito de SOA son: la comprensión, la gobernanza, la gestión y el apoyo [1].

Siguiendo una transformación *top-down* y la aprobación de un cambio cultural con visión y compromiso es la forma de lograr una verdadera arquitectura orientada a servicios con sus beneficios. Las siete metas (representado en la **Figura 10** de SOA pueden categorizarse en dos grupos, los objetivos estratégicos y beneficios resultantes [1], [6].

Los objetivos estratégicos son aumento de la federación, aumento de la interoperabilidad intrínseca, incremento de las opciones de diversidad de proveedores, aumento de negocios y alineación de la tecnología. Las tres ventajas resultantes pueden llegar, dependiendo de cómo sean realizadas [1], [6].

Estos beneficios incrementan el ROI (*Return of Invesment*), la reducción de la carga de TI y un incremento en la agilidad organizacional [1], [6].

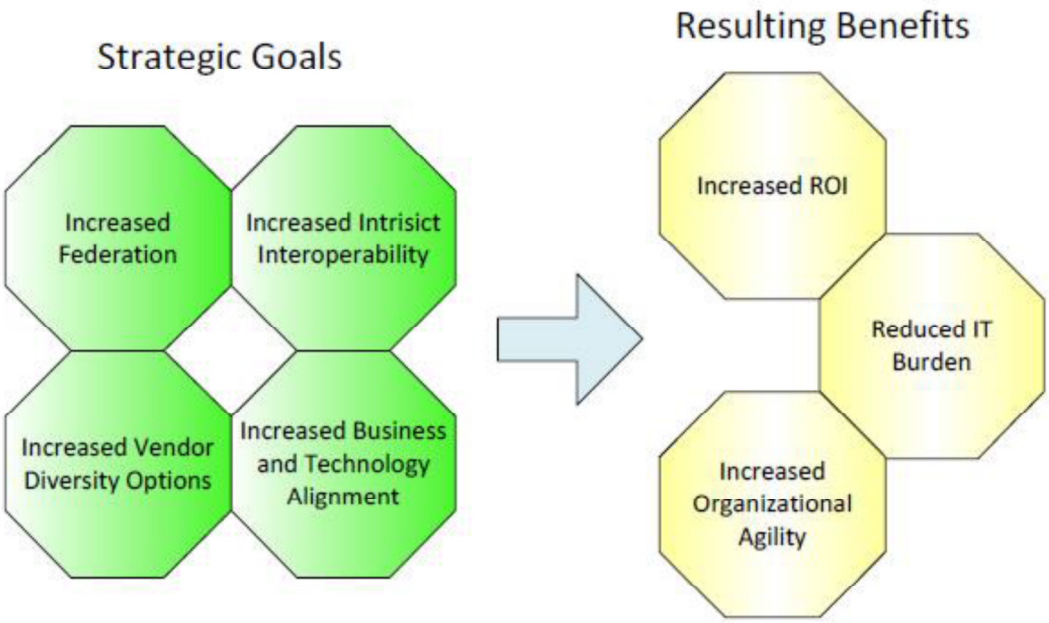

Figura 10: Beneficios de SOA [9].

SOA requiere un cambio significativo del clima organizacional, así como una cantidad significativa de tiempo y esfuerzo. SOA no es una panacea y no es conveniente para todas las situaciones [19]. Algunas organizaciones han sufrido decepciones con sus proyectos SOA [13].

También, se pueden encontrar críticas contra SOA en algunos blogs. Algunos expertos [8], [23] dicen que SOA no ha podido cumplir las promesas y expectativas que se han establecido para ello. Todavía creen que aunque SOA no exista ya sus principios se quedarán, y serán adoptados por nuevas tecnologías, como SaaS y *Cloud Computing*.

Según Haines y Rothenberg, la pregunta más importante es cómo SOA puede hacerse bien en una situación actual o es posible o razonable hacerlo en absoluto [13].

2.3.8. Limitaciones de SOA

En un mundo tecnológico de constantes cambios, se espera que las empresas provean más con menos recursos. SOA provee a las empresas de desarrollo de software la habilidad de responder rápida y eficientemente a las solicitudes de servicio. Sin embargo, SOA no es compatible con todas las aplicaciones. A continuación se detallan las principales limitaciones de esta arquitectura:

- SOA depende de la implementación de estándares. Sin estándares, la comunicación entre aplicaciones requiere de mucho tiempo y código.

- SOA no es para: aplicaciones con alto nivel de transferencia de datos, aplicaciones que no requieren de implementación del tipo *request/response* y para aplicaciones que tienen un corto periodo de vida.

- Incrementalmente se hace difícil y costoso el ser capaz de cumplir con los protocolos y hablar con un servicio.

- Implica conocer los procesos del negocio, clasificarlos, extraer las funciones que son comunes a ellos, estandarizarlas y formar con ellas capas de servicios que serán requeridas por cualquier proceso de negocio.

- En la medida en que un servicio de negocio, vaya siendo incorporado en la definición de los procesos de negocio, dicho servicio aumentara su nivel de criticidad. Con lo cual cada vez que se requiera efectuar una actualización en dicho servicio (por ejemplo, un cambio en el código, una interfaz nueva, etc.), deberá evaluarse previamente el impacto y tener mucho cuidado con su implementación. Sin embargo, parte de la problemática anterior, puede ser solventada en virtud a un buen diseño del servicio.

2.4. Microsoft XML Document Transformation (XDT)

Microsoft Azure emplea documentos en formato XML para las configuraciones como puede observarse en [26]. Para poder modificar estos archivos de configuración, Microsoft permite a los desarrolladores el empleo de ficheros XDT [25] para modificar los archivos XML de la configuración.

Los ficheros XDT utilizan el metalenguaje XML pero con un **namespace** propio (XDT) que entiende el motor que utiliza Azure. Este *namespace* es muy simple ya que únicamente agrega dos atributos **Locator** y **Transform**.

El atributo **Locator** permite indicar que parte del archivo de configuración va a modificarse, para ello *Locator* provee de varias funciones:

- **Condition**: Especifica una expresión XPath que se combina con la expresión XPath del elemento actual. Aquellos elementos que coinciden con la expresión XPath combinada son seleccionados.
- **Match**: Selecciona aquellos elementos que tienen el mismo valor indicado en uno de los atributos indicados.
- **Xpath**: Permite especificar una expresión XPath absoluta, a diferencia de Condition esta expresión no se combina con la expresión XPath actual.

El atributo **Transform** permite a los desarrolladores indicar cuál va a ser la acción a realizar sobre el elemento seleccionado (mediante *Locator*). Para ello el atributo *Transform* tiene varias opciones:

- Replace
- Insert
- InsertAfter
- InsertBefore
- Remove
- RemoveAll
- RemoveAttributes
- SetAttributes

2.5. El método DIARy (Dynamic Incremental Architectural Reconfiguration)

2.5.1. Introducción

El método DIARy permite la construcción de aplicaciones *cloud* como una composición de servicios, donde cada diseño de servicio es incluido dentro de un proceso incremental de integración que permite especificar cómo el servicio será integrado dentro de la aplicación *cloud*. Los desarrolladores utilizan la especificación de la integración del incremento para generar artefactos software tales como esqueletos de lógica de los microservicios, protocolos de interacción, y scripts para desarrollar, desplegar y reconfigurar la arquitectura actual de la aplicación *cloud* [46].

2.5.2. Descripción del proceso

Como puede observarse en la **Figura 11**, el método DIARy está compuesto principalmente por tres fases:

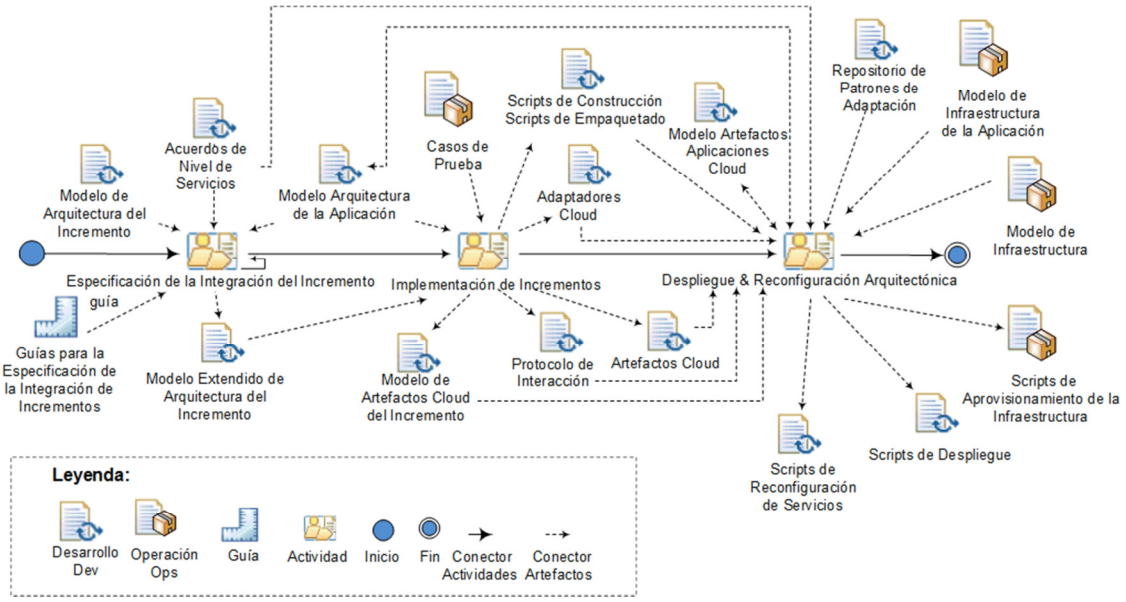

Figura 11: Extracto del método DIARy [38].

- **Especificación de la Integración del Incremento**: Esta actividad tiene como objetivo permitir a los desarrolladores especificar como integrar un incremento de software en la arquitectura de un servicio *cloud*. Para ello permite a los desarrolladores especificar tanto la lógica de integración como el impacto arquitectónico de la integración, sin tomar en consideración las características específicas de un entorno cloud. En esta actividad los desarrolladores toman como entrada un incremento de software y lo integran como parte de un servicio, siguiendo las *Guías para la Especificación de la Integración de Incrementos*. Las decisiones de diseño se toman en base a los términos de un *Acuerdo de Nivel de Servicios*. Esta es una actividad iterativa por lo que los desarrolladores podrían especificar la integración de incrementos de software compuesto por varios servicios. Con el fin de

apoyar esta actividad los desarrolladores utilizan el *Perfil de Especificación DIARy* [27], que es un lenguaje de Dominio Especifico para especificar las decisiones de diseño de la integración, produciendo un *Modelo Extendido de la Arquitectura del Incremento*.

- **Implementación del Incremento**: Esta actividad tiene como objetivo apoyar el proceso de integración mediante la generación de artefactos *cloud* específicos para la plataforma de despliegue. Los artefactos a generar implementan: i) el *Protocolo de Interacción* entre los servicios *cloud* a ser integrados y la aplicación destino, ii) *Adaptadores Cloud* que corrigen incompatibilidades entre interfaces de servicios, y iii) *Artefactos Cloud* que implementan la lógica de *servicios cloud* y archivos de configuración como *scripts* de construcción y empaquetado.

- **Despliegue y Reconfiguración de la Arquitectura**: Los desarrolladores seleccionan del *Repositorio de Patrones de Adaptación* los patrones de adaptación apropiados para integrar la arquitectura del incremento en la arquitectura actual. Luego ejecutan las transformaciones M2T con el fin de operacionalizar los patrones de adaptación seleccionados de acuerdo al *Modelo Extendido de la Arquitectura del Incremento*, el *Modelo de Artefactos Cloud del Incremento*, y la tecnología de despliegue seleccionada. Los *artefactos Cloud* generados son: i) *Scripts* de despliegue y aprovisionamiento de paquetes previamente generados conjuntamente con los archivos de configuración correspondientes, y ii) *Scripts* para la reconfiguración de la arquitectura de aplicaciones [38].

3. Automatización de la reconfiguración dinámica de arquitecturas

La **reconfiguración dinámica de la arquitectura** consiste en cambiar la estructura y comportamiento de la arquitectura de la aplicación, en tiempo de ejecución, sin interrumpir los servicios de la aplicación. La estructura de la aplicación se actualiza ya sea incluyendo, modificando o eliminando servicios y dependencias entre servicios; mientras que el comportamiento se actualiza incluyendo, modificando o eliminando servicios que implementan el protocolo de interacción entre los servicios de la aplicación. Además, DIARy promueve la generación automática de artefactos de software o scripts que agilitan y disminuyen costos de desarrollo durante la implementación, aprovisionamiento, despliegue y reconfiguración. En este trabajo se han refinado los meta-modelos propuestos en DIARy, y sean definido e implementado las trasformaciones necesarias para automatizar reconfiguración.

3.1. Metamodelos

3.1.1. Metamodelo origen de la transformación: EIAM

El metamodelo Extended Increment Architecture Model (EIAM) extiende el metamodelo de SoaML el cual permite describir la arquitectura de la aplicación o del incremento de software a través de sus distintas capas: *capa de servicios de aplicación*, la *capa de servicios de negocio* y la *capa de orquestación de servicios*. EIAM extiende el metemodelo de SoaML incluyendo metaclases, atributos y relaciones que permiten describir información del impacto de la integración de un incremento sobre la arquitectura actual y los requerimientos de gestión de las variaciones de carga de los recursos *cloud*, retrasos en la atención de servicios y configuraciones de servicios. Como puede apreciarse en la **Figura 12** los principales son (las metaclases que incluyen un icono junto a su nombre corresponden a las metaclases del metamodelo UML que extendidas tanto por SoaML como por DIARy):

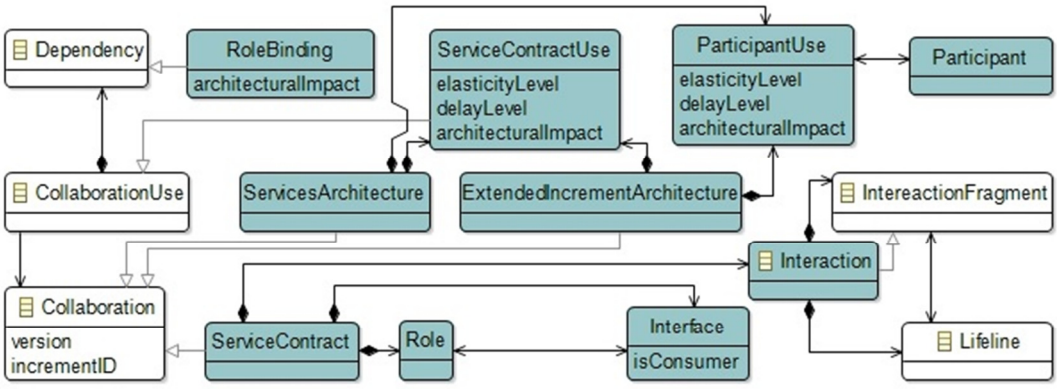

Figura 12: Extracto del metamodelo Extended Increment Architecture Model.

- **incrementId** y **versionID**: Son identificadores para saber a qué incremento corresponde cada uno de los elementos del modelo.
- **architecturalImpact** : Indica el impacto que tendrá este elemento al integrarlo (*Add, Modify, Delete*).
- **representCloudArtifact**: Indica si los cambios en elementos arquitectónicos deben ser propagados a los artefactos *cloud* relacionados durante la implementación.
- **elasticityLevel**: Indica el nivel de escalado necesario, por ejemplo el número de máquinas que deberían asociarse a este elemento
- **delayLevel**: Indica si el servicio *cloud* requiere ejecución inmediata o si el servicio de la plataforma *cloud* requiere retrasar las peticiones de procesamiento (e.j. colas de mensajes).
- **participantUse**: Hacen referencia a un *Participant* y representan las diferentes implementaciones que estos pueden tener dependiendo del contexto en el que se encuentren.
- Un **ServiceContract** es cada uno de los servicios que se pueden ofrecer. En el *ServiceContract* se describe la orquestación entre los participantes del servicio. Entre los componentes internos del ServiceContract están las interfaces y el diagrama de actividad que describe la orquestación.
- **ServiceContractUse**: Hace referencia a un *ServiceContract*, se usa para reutilizar la información de un *ServiceContract*.
- **Role:** Permiten relacionar los *Participant* con los ServiceContract e indican a los primeros que interfaces deberían implementar.
- **Role Binding:** Indican cómo se enlazan los distintos *CollaborationUse* (*ServiceContractUse* y *ParticipantUse*) entre sí.

3.1.2. Metamodelo destino de la transformación: CAM

El metamodelo CAM (Cloud Artifacts Model) permite describir, de manera independiente de la tecnología de implementación y del entorno *cloud* de despliegue, los artefactos de software (ej., código de aplicación, interfaces, archivos de configuración, etc.) que deben ser implementados por cada uno de los elementos arquitectónicos del EIAM. La metaclase *DIARyArchitecturalElement* permite establecer la correspondencia entre los elementos de este modelo con los elementos arquitectónicos de EIAM. Para un mayor detalle de la descripción de CAM ver [46].

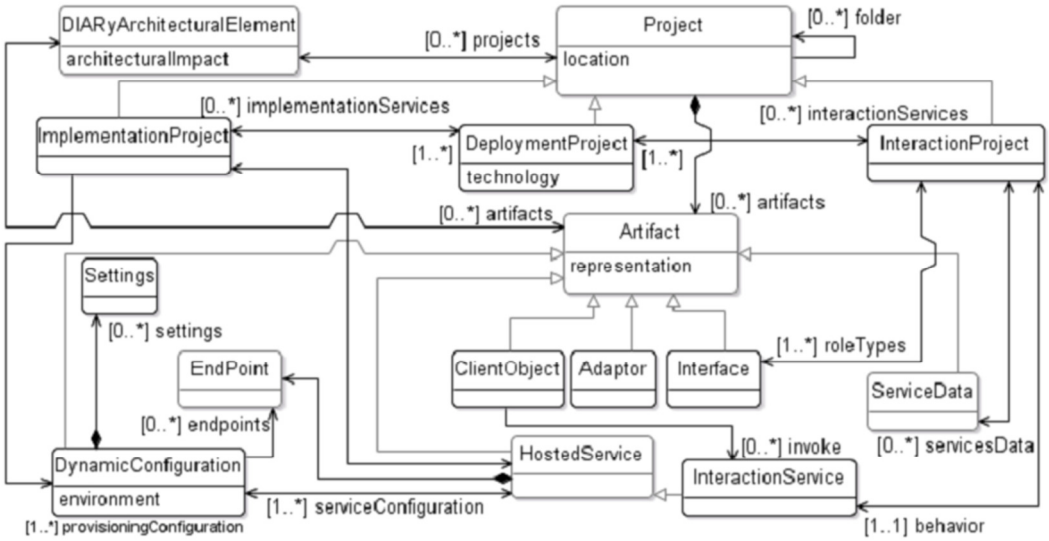

Figura 13: Extracto del metamodelo Cloud Artifacts Model.

Como puede observarse en la **Figura 13**, el metamodelo CAM agrupa los artefactos de software que deben ser implementados en tres tipos de proyectos:

- **InteractionProject**: Incluye los artefactos de software necesarios para implementar los protocolos de interacción (orquestación), descritos como *ServiceContract* en el EIAM. La implementación y despliegue de este proyecto permitirá ofrecer la orquestación de servicios como otro servicio. Entre estos artefactos se están las interfaces y diagrama de secuencia (metaclase *Interaction*) que forman parte de un *ServiceContract*.
- **ImplementationProject**: Incluye los artefactos de software necesarios para implementar el diseño de los servicios descritos como *Participant* en el EIAM.
- **DeploymentProject**: Incluye proyectos de implementación o interacción que será desplegados en un mismo nodo (ej., máquina virtual) de un entorno *cloud*. Todos los servicios incluidos en los proyectos de implementación o interacción compartirán los recursos del nodo. Estos proyectos también incluyen información de aprovisionamiento de recursos *cloud* descritos a través de elementos *DynamicConfiguration*.

Algunos de los elementos que caben destacar de este metamodelo son:

- **incrementID** y **elementVersion**: Indican el incremento y la versión a la que corresponde cada elemento.
- **architecturalImpact**: Tiene la misma funcionalidad que en el metamodelo EIAM, indica el impacto que tendrá este elemento al integrarlo (*Add, Modify, Delete*).
- **technology**; Indica la tecnología con la cual ha sido desarrollado este elemento; esto es debido a que cada elemento puede estar desarrollado en distintas tecnologías.
- **EndPoint**: Corresponde con localización en la que se puede encontrar un servicio y/o participante o un elemento. Es necesario conocerlo con tal de poder interactuar con el elemento en cuestión.
- **DynamicConfiguration**: Separa la información de configuración de un servicio de la implementación del servicio favorece la reconfiguración dinámica ([11][14]), por lo tanto este artefacto describe información de las configuraciones de los servicios que podrán cambiar mientras éste se encuentra en ejecución.

3.2. Definición de Transformaciones

Debido a que este trabajo de fin de grado está orientado a la reconfiguración dinámica de la arquitectura los elementos clave que hemos seleccionado para nombrar están relacionados con la configuración arquitectónica.

3.2.1. Reglas de transformación entre EIAM y CAM

En este apartado se definen las correspondencias y reglas de transformación entre los elementos arquitectónicos de EIAM y los artefactos de software que los implementaran descritos en CAM. Las principales correspondencias en las transformaciones empleadas en el trabajo han sido las siguientes:

Por cada **Participant** origen generamos un **DeploymentProject** (indicando en este las características para desplegar) con sus configuraciones (**DynamicConfiguration**), así como también generamos un **Participant**.

```
rule Participant {
	from
		ParticipantInput : EIAM!Participant
	to

	Participant : CAM!Participant (
			name <- ParticipantInput.name,
			cloudartifactsmodel <-
ParticipantInput.extendedincrementarchitecturemodel
		),
		DeploymentProject : CAM!DeploymentProject
		(
			name <- ParticipantInput.name,
			cloudartifactsmodel <-
ParticipantInput.extendedincrementarchitecturemodel,
			belongsToParticipant <- Participant,
			artifacts <- Set{ServiceConfiguration}
		),
		ServiceConfiguration : CAM!DynamicConfiguration
		(
			name <- 'ServiceConfiguration.cloud.cscfg',
			projectConfiguration <- DeploymentProject,
			project <- DeploymentProject
		)
}
```

Extracto de la regla de transformación de Participant

Por cada **ParticipantUse** (instancia de un **Participant** dependiendo del contexto) generamos un **ImplementationProject** (una implementación del participante en un contexto concreto) el cual está enlazado con el **DeploymentProject** del **Participant** al que pertenece (cabe destacar que en el caso de que para este contexto en concreto fuese necesario una configuración de despliegue específica, podría generarse un **DeploymentProject** para una única instancia de un **Participant** en el caso de que fuese necesario). Así mismo también se generan la configuración específica para el **ImplementationProject** así como los **EndPoint**(a partir de los **RoleBinding** que posea) necesarios para poder interaccionar con este. En caso que la interface que implementa el participante tiene el atributo *isConsumer* = *true*, también se generará un **ClientObject**.

```
rule ParticipantUse {
	from
		ParticipantUseInput : EIAM!ParticipantUse
	to
		ImplementationProject : CAM!ImplementationProject (
			name <- ParticipantUseInput.name,
			cloudartifactsmodel <-
ParticipantUseInput.extendedincrementarchitecture.extendedincrementarchitectur
emodel,
			belongsToParticipant <-
thisModule.resolveTemp(ParticipantUseInput.participantType, 'Participant'),
			deployment <-
thisModule.resolveTemp(ParticipantUseInput.participantType,
'DeploymentProject'),
			artifacts <- Set{ImplementationProjectConfiguration,
FrontEndService, FrontEndConfiguration}
		),
		ImplementationProjectConfiguration : CAM!DynamicConfiguration
		(
			name <- 'ServiceConfiguration.cloud.cscfg',
			projectConfiguration <- ImplementationProject
		),
		FrontEndService : CAM!FrontEndService (
			project <- ImplementationProject

		),
		FrontEndConfiguration : CAM!DynamicConfiguration (
			name <- 'Web.config',
			configurationService <- FrontEndService
		)
}
```

Extracto de la regla de transformación de ParticipantUse

Por cada **ServiceContract** se genera un **ServiceContract** (SC), un **InteractionProject**, la configuración propia del SC (**DynamicConfiguration**). Al interior del InteractionProject se genera un InteractionService (a partir del diagrama de secuencia del SC indicando así el comportamiento de este), sus **EndPoints** a partir de los **RoleBindings** que posea, y la Configuración del **InteractionService**. Además en el **InteractionProject** se generan también los **MessageType** que utiliza el SC para comunicarse a partir de los **MessageType** originales.

```
rule ServiceContract {
	from
		ServiceContractInput : EIAM!ServiceContract
	to
		ServiceContract : CAM!ServiceContract (
			name <- ServiceContractInput.name,
			cloudartifactsmodel <-
ServiceContractInput.extendedincrementarchitecturemodel
		),
		InteractionProject : CAM!InteractionProject
		(
			name <- ServiceContractInput.name,
			cloudartifactsmodel <-
ServiceContractInput.extendedincrementarchitecturemodel,
			artifacts <- Set{InteractionProjectConfiguration,
InteractionService, InteractionServiceConfiguration}
		),
		InteractionProjectConfiguration : CAM!DynamicConfiguration
		(
			name <- 'ServiceConfiguration.cloud.cscfg',
			projectConfiguration <- InteractionProject
		),
		InteractionService : CAM!InteractionService
		(
			name <- ServiceContract.name,
			interactionProject <- InteractionProject
		),
		InteractionServiceConfiguration : CAM!DynamicConfiguration
		(
			name <- 'Web.config',
			configurationService <- InteractionService
		)
}
```

Extracto de la regla de transformación de ServiceContract

Por cada **ServiceContractUse** (son como punteros a un SC, permitiendo reutilizar el SC sin tener que duplicarlo) se genera un **ServiceContractUse** que está relacionado con el SC al que apunta.

```
rule ServiceContractUse {
	from
		ServiceContractUseInput : EIAM!ServiceContractUse
	to
		ServiceContractUse : CAM!ServiceContractUse (
			name <- ServiceContractUseInput.name,
			cloudartifactsmodel <-
ServiceContractUseInput.extendedincrementarchitecture.extendedincrementarchite
cturemodel,
			implementationProjects <-
ServiceContractUseInput.rolebinding->iterate(Rolebinding; participantUses:
Set(soaml!ParticipantUse) = Set {} |
                            Rolebinding.supplier->iterate(participantUse; res2:
Set(soaml!ParticipantUse) = Set{} |
                                  participantUses->including(participantUse)
                                  ))->collect(participantUse |
thisModule.resolveTemp(participantUse, 'ImplementationProject'))

		)
}
```

Extracto de la regla de transformación de ServiceContractUse

Por cada Interface generamos una **Interface** dentro de cada SC que la emplea, indicando cual es el **ImplementationProject** que lo implementa).

```
rule Interface {
	from
		InterfaceInput : EIAM!Interface
	to
		Interface : CAM!Interface (
			name <- InterfaceInput.name,
			project <-
thisModule.resolveTemp(InterfaceInput.servicecontract, 'InteractionProject')
		)
}
```

Extracto de la regla de transformación de Interface

A continuación en la **Tabla 1** mostramos las principales correspondencias que hemos nombrado anteriormente.

ORIGEN	DESTINO
ExtendedIncrementArchitectureModel	DeploymentProjec DynamicConfiguration Participant
Participant	Participant DeploymentProject DynamicConfiguration
ParticipantUse	ImplementationProject FrontEndService DynamicConfiguration ClientObject (En caso del participante que tiene un rol de consumidor del servicio)
ServiceContract	ServiceContract InteractionProject DynamicConfiguration InteractionService
ServiceContractUse	ServiceContractUse
Interface	Interface
RoleBinding	Exposed EndPoint Invoked EndPoint
Message Type	Message Type

Tabla 1: Correspondencia entre EIAM y CAM.

3.2.2. Reglas de transformación entre CAM y XDT

La reconfiguración dinámica de la arquitectura consiste en integrar los servicios de la arquitectura del incremento en la arquitectura de la aplicación y remplazar las dependencias entre servicios. Las dependencias entre servicios se almacenan en los artefactos de configuración de los servicios relacionados; por lo tanto, en este apartado utilizamos la información de dependencias entre servicios, descrita en los elementos de tipo ***ExposedEndPoint*** e ***InvokedEndpoint*** del CAM, y actualizamos los artefactos de configuración por medio de archivos XDT. En este apartado se definen las correspondencias y reglas de transformación para la transformación M2T (Model to Text) que permita generar el archivo XDT que actualizará los artefactos de configuración en base a la información de CAM.

Para seleccionar los *EndPoints* adecuados hemos mirado su atributo **ArtifactImpact** ya que solo tenemos que generar la configuración correspondiente cuando se añada o modifique un *EndPoint*, ya que cuando se elimine el *EndPoint* será porque se ha eliminado un proyecto, así que se eliminará automáticamente el *EndPoint* correspondiente y no tendremos que tenerlo en cuenta. Los EndPoint que habían que filtrar eran aquellos que tenían el estado de *Add* o *Modified* en su atributo **ArtifactImpact**.

Debido a que estamos trabajando con Windows Azure, por cada *DeploymentProject* se modifica su artefacto de configuración de acuerdo a la información de los *EndPoints* correspondientes a los a los *FrontEndService*, *InteractioService* incluidos en los *ImplementationProject* e *InteractionProject* relacionados a cada *DeploymentProject*. La transformación m2t que genera el XDT correspondiente es:

```
<?xml version="1.0" encoding="utf-8"?>

<ServiceConfiguration serviceName="DEPLOYMENTPROYECT_NAME" xmlns =
"http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceConfiguration"
osFamily="4" osVersion="*" xmlns:xdt="http://schemas.microsoft.com/XML-
Document-Transform" >

      <Role name=" DEPLOYMENTPROYECT/INTERACTIONPROJECT_NAME ">

            <ConfigurationSettings xdt:Transform="InsertIfMissing">

<Setting name="ENDPOINT_NAME" value="DIRECCION_ENDPOINT"
xdt:Transform="Insert/ Replace" />

<Setting name ENDPOINT_NAME " value="DIRECCION_ENDPOINT"
xdt:Transform="Insert/Replace" />

            </ConfigurationSettings>

      </Role>

</ServiceConfiguration>
```

[Página intencionalmente en blanco]

4. Caso de estudio

En este trabajo hemos diseñado un caso de estudio que muestra el comportamiento de DIARy cuando tiene que trabajar con elementos externos. El caso de estudio que hemos diseñado es un buscador de hoteles.

Un buscador de hoteles es un servicio que para llevarse a cabo necesita interaccionar con elementos externos a la empresa (las cadenas de hoteles que les suministra la información necesaria para ofrecer el servicio) por lo cual hemos visto que es un caso de estudio muy interesante ya que permite ver cómo trabaja DIARy tanto con elementos internos como externos.

Para empezar con el método necesitamos un modelo de tipo EIAM (Extended Increment Architecture Model), como hemos explicado anteriormente EIAM se define en varias capas. En este caso de estudio solo vamos a mostrar EIAM en dos capas (la *capa de servicios de aplicación* y la *capa de servicios de negocio)*, no incluyendo la *capa de orquestación de servicios* que corresponde con un diagrama de secuencia.

Como puede observarse en la **Figura 11** el primer paso del método DIARy es la **especificación de la integración del incremento** que tiene como entrada dos modelo que satisfacen el meta-modelo EIAM siendo uno la arquitectura actual de la aplicación y el otro la arquitectura del incremento, indicando las modificaciones que van a realizarse.

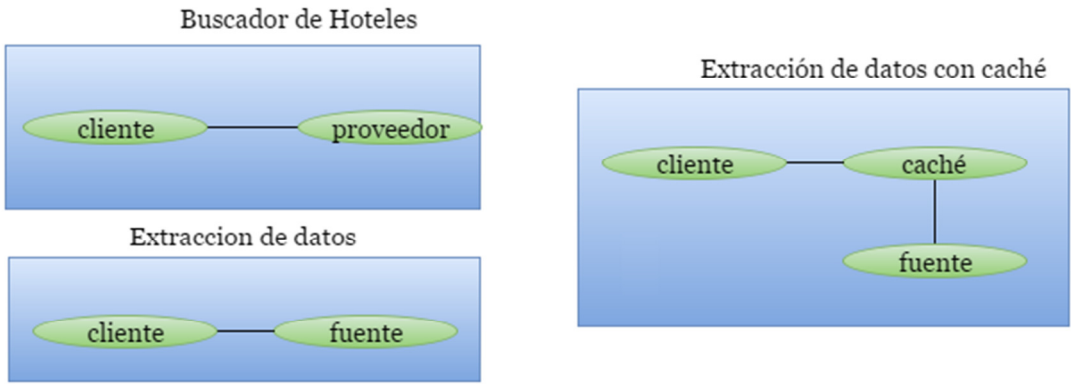

Figura 14: Extracto de la capa de servicios de negocio de EIAM.

En la **Figura 14** podemos ver un extracto de la vista de servicios de negocio de EIAM, en este caso mostramos la arquitectura de los *ServiceContract* indicando con qué roles pueden relacionarse. Con tal de ahorrar espacio, en esta figura representamos las dos capas de negocio de servicios de ambos modelos.

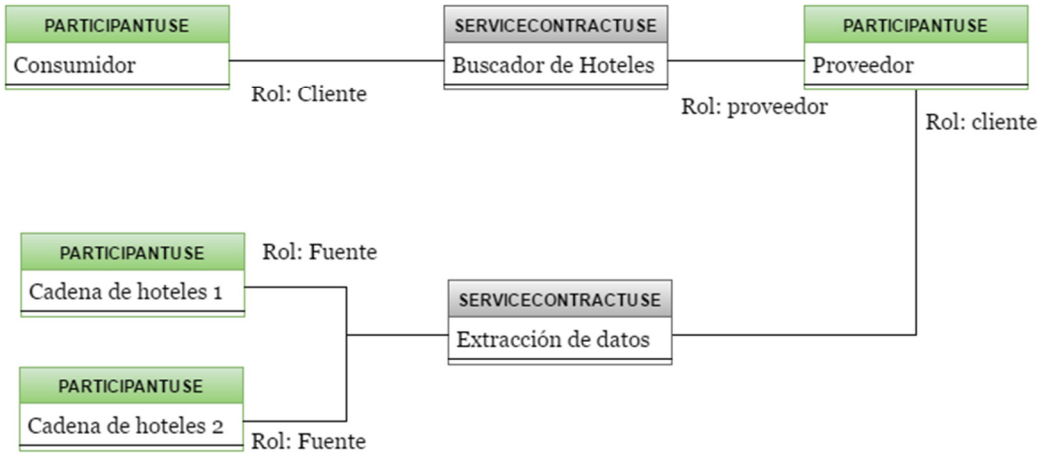

Figura 15: Extracto de la representación de la capa de servicios de aplicación de la arquitectura actual.

En la **Figura 15** podemos visualizar un extracto de la representación de la figura actual, donde podemos ver las instancias de los *ServiceContract*, y *Participant* (en los *ServiceContractUse* y *ParticipantUse*) en un contexto específico, así como las relaciones y roles que implementan cada uno.

Cabe destacar que debido a que los *ParticipantUse* cadena de hoteles 1 y 2 son elementos externos a nuestra arquitectura es necesario crear un **adaptador**

para poder comunicarnos con estos. Esto origina que a la hora de generar la configuración dinámica de la arquitectura tenemos que tener en cuenta donde se pueden localizar estos *ParticipantUse* externos.

En este caso de estudio asumimos que nuestra empresa ha estado ofreciendo el servicio del buscador de hoteles y ha estado ganando importancia por lo que más gente ha comenzado a utilizar nuestro servicio. A causa de este crecimiento, nuestra aplicación está tardando cada vez más en responder las solicitudes que recibe. Ya que en la arquitectura actual utilizamos una interacción directa con nuestras fuentes de información, cada vez que recibimos una solicitud de buscar un hotel lo que hace nuestra aplicación es comunicarse con todas nuestras fuentes, así que con tal de agilizar el proceso se ha decidido incluir un servidor caché a la hora de extraer los datos.

Como ya hemos dicho anteriormente en la **Figura 14** podemos ver la arquitectura de los servicios de este nuevo modelo.

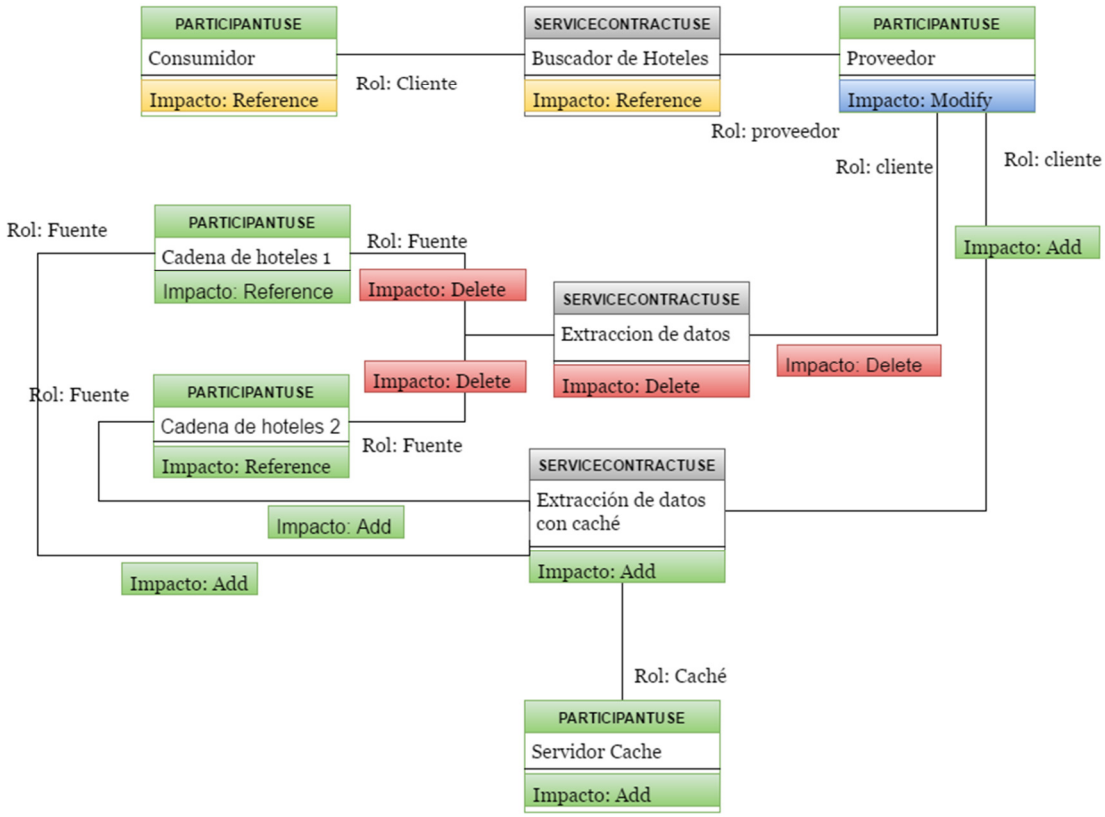

Figura 16: Extracto de la representación de la capa de servicios de aplicación del nuevo incremento de la arquitectura.

Como puede verse en la **Figura 16** en este modelo representamos no solo el incremento de la nueva arquitectura sino que también mostramos cual va a ser el impacto que va a realizar sobre la arquitectura actual y las modificaciones que van a realizarse en esta.

En este caso estamos sustituyendo el *ServiceContract* de **Extracción de datos** por el SC de **Extracción de datos con caché**. Para poder realizar esto necesitamos eliminar las antiguas relaciones que existen entre los distintos elementos que se relacionan con el anterior SC y poner nuevas relaciones con el nuevo SC. Además de esto también necesitamos modificar la implementación que tiene el *ParticipantUse* que implementa el **rol Cliente** en este nuevo SC, ya que antes se relacionaba directamente con las fuentes y ahora debe relacionarse con el servidor caché, y este con las fuentes en el caso de que lo necesite.

Una vez que se tienen ambos modelos y se ha verificado que pueden integrarse correctamente, se procede al segundo paso de DIARy, la **Implementación de incrementos**. En esta segunda fase a partir de los modelos anteriormente citados generamos un nuevo modelo, el ***Modelo de artefactos Cloud del incremento*** de tipo CAM. Utilizando ATL para describir las transformaciones que se realizan entre los metamodelos.

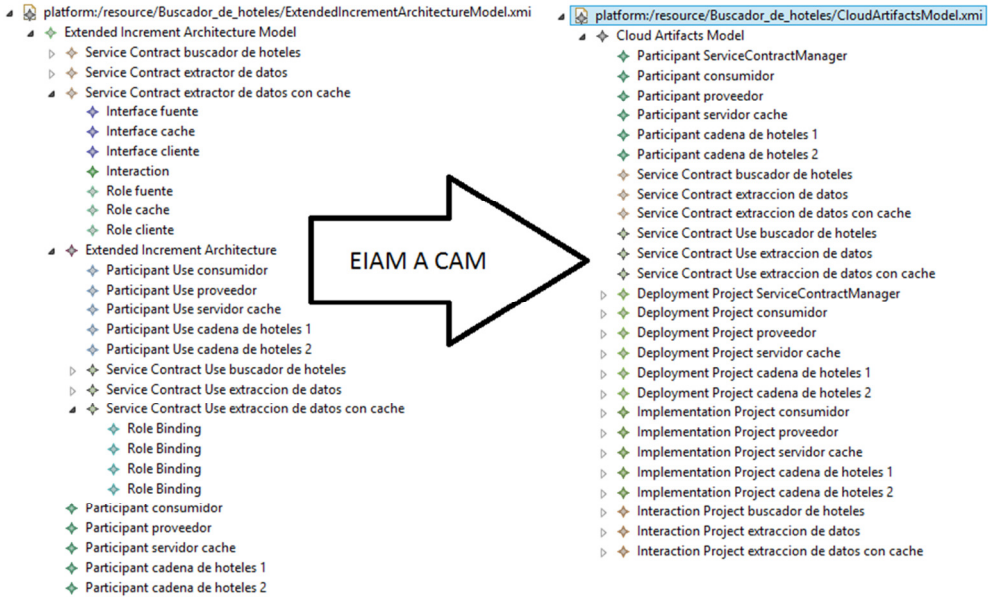

Figura 17: Transformación entre el *modelo del nuevo incremento* y el *modelo de artefactos cloud del incremento*.

A continuación de la generación del ***Modelo de artefactos cloud del incremento*** de tipo CAM se pasa ya a la siguiente fase del método DIARy, el **despliegue y reconfiguración arquitectónica**; en donde, tal como se indicó anteriormente, la reconfiguración arquitectónica se produce al incluir nuevos servicios o al modificar las dependencias entre ellos. En esta fase a partir del modelo generado anteriormente se realiza una transformación M2T generando artefactos que modificaran los archivos configuración incluyendo o actualizando la información de las dependencias entre servicios obtenidas de elementos EndPoint del EIAM. Dado que el caso de estudio ha sido implementado en Microsoft Azure, generamos archivos XDT, los cuales modifican la información de archivos de configuración mientras se produce el despliegue de los servicios incluidos en un incremento de software.

En el lado izquierdo de la **Figura 18** remarcado en rojo puede observarse los archivos XDT que ha generado. En esta transformación primeramente generamos una carpeta cuyo nombre es el nombre del proyecto que modificamos a continuación dentro de esta carpeta generamos el archivo XDT que modificará la configuración. En este caso en el lado derecho de la figura puede observarse el archivo XDT que se ha generado para modificar la configuración del proyecto "*cadena de hoteles 1*".

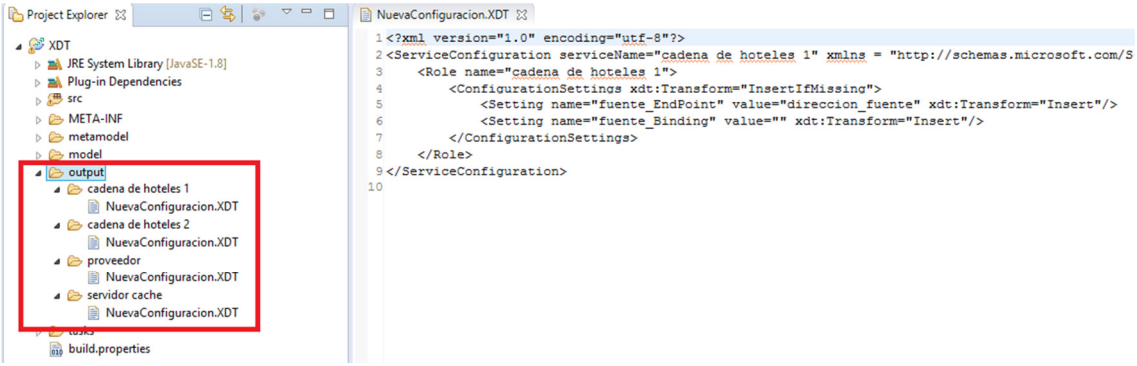

Figura 18: Transformación de modelo a texto.

A continuación mostramos el archivo XDT que ha sido generado para modificar el proyecto proveedor, el que más cambios ha sufrido.

```
<?xml version="1.0" encoding="utf-8"?>

<ServiceConfiguration serviceName="proveedor" xmlns =
"http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceConfiguration"
osFamily="4" osVersion="*" xmlns:xdt="http://schemas.microsoft.com/XML-
Document-Transform" >
       <Role name="proveedor">
              <ConfigurationSettings xdt:Transform="InsertIfMissing">
                     <Setting name="cliente_EndPoint" value="direccion_cliente"
xdt:Transform="Insert"/>
                     <Setting name="cliente_Binding" xdt:Transform="Insert"/>
              </ConfigurationSettings>
       </Role>
       <Role name="proveedor">
              <ConfigurationSettings xdt:Transform="InsertIfMissing">
                     <Setting name="cliente_EndPoint" value="direccion_cliente"
xdt:Transform="Insert"/>
                     <Setting name="cliente_Binding" xdt:Transform="Insert"/>
              </ConfigurationSettings>
       </Role>
</ServiceConfiguration>
```

Archivo XDT de modificación de la configuración de proveedor

5. Conclusiones y trabajo futuro

5.1. Conclusiones

El objetivo principal de este trabajo ha sido la implementación del proceso de **reconfiguración de la arquitectura** del método DIARy por medio de la **automatización**, la generación de la nueva configuración arquitectónica de la aplicación que integra el nuevo incremento de la aplicación desarrollada. Un incremento se ha definido como un conjunto de servicios *cloud* desarrollados en una iteración de un proceso de desarrollo ágil. Para ser exactos, se genera automáticamente la nueva configuración de los *EndPoint* (que son los puntos de acceso a cada servicio y/o participante para poder interactuar con él) de cada uno de los servicios cloud afectados.

Con tal de facilitar al desarrollador el empleo de DIARy se ha empleado la tecnología de MDE (*Model Driven Engineering*) permitiendo así la automatización de las tareas, además al emplear una extensión del *profile de SoaML*, que es un estándar de la OMG para la especificación de servicios, da una mayor sensación de familiaridad.

Según los objetivos planteados en la primera sección del trabajo:

- Una transformación de tipo modelo a modelo (M2M) que utiliza un metamodelo extensión de SoaML y un metamodelo específico de las plataformas *cloud*.

- Una transformación de tipo modelo a texto (M2T) que utiliza el metamodelo específico de plataformas *cloud* y la configuración necesaria para desplegar el proyecto en la plataforma *cloud* Microsoft Azure.

Puede decirse que ambos objetivos han sido abordados y llevados a cabo con éxito.

Con respecto al primer objetivo se refinaron los metamodelos iniciales y se corroboró que fueran correctos, así como también se generó la transformación entre ambos metamodelos empleando el lenguaje de programación ATL. Tras esto se comprobó tras varias pruebas que el resultado era correcto.

En cuanto al segundo objetivo se estudió el metalenguaje de XDT para poder utilizarlo adecuadamente y se implementó una transformación entre el modelo que representa la arquitectura de la aplicación y los archivos XDT que genera que permiten la reconfiguración de la arquitectura.

Como puede observarse los objetivos de este trabajo fueron abarcados en su totalidad y hay que destacar que a nivel académico se ha logrado un claro entendimiento del tema, desarrollo de aplicaciones *cloud*, empleo del paradigma de SOA, y aprendizaje de varios lenguajes.

5.2. Trabajo futuro

Como hemos dicho con anterioridad durante el transcurso de este trabajo de fin de grado únicamente hemos implementado aquellas partes del método DIARy que estaban relacionadas con la reconfiguración dinámica de la arquitectura, para ser exactos solamente la reconfiguración de los *EndPoint* (puntos de acceso de los diversos elementos). Además, esta reconfiguración ha sido orientada únicamente para Microsoft Azure©, debido a que era la plataforma *cloud* que poseíamos.

En un futuro próximo se pretende implementar la reconfiguración para otras plataformas *cloud*, así como extenderla y que no abarque únicamente los *EndPoints* si no también otros elementos como la configuración del tiempo necesario para desplegar un servicio, los retardos que necesita y demás opciones de configuración.

Actualmente se está construyendo una transformación entre los diagramas de secuencia, que son parte del metamodelo EIAM y el protocolo de interacción implementando como un servicio *workflow* en Azure.

A todo esto en este trabajo de fin de grado solo hemos implementado las partes más significativas y necesarias para poder realizar la reconfiguración, así que en un futuro también se quiere ampliar la implementación de otras secciones y no solo la de la reconfiguración. Un ejemplo de las partes que podrían implementarse sería tener en cuenta los Acuerdos de nivel de servicios (SLA), las Guías para la Especificación de la Integración de Incrementos, etc.

5.3. Publicaciones

Durante el transcurso de este trabajo de fin de grado se han realizado dos contribuciones en modo de publicación que fueron sometidas a un proceso de revisión y fueron aceptadas en una conferencia nacional y en otra conferencia internacional.

- Zúñiga M., Abrahão S., Insfran E., **Cano C.**: "Incremental Integration of Microservices in Cloud Applications", 25th International Conference on Information Systems Development. Katowice, Poland (ISD 2016)

  * *Esta conferencia se encuentra catalogada en el Ranking de Conferencias CORE con la categoría A.*

- Sandobalín J., Zúñiga M., Insfran E., Abrahão S., **Cano C.**: "Una aproximación DevOps para el Desarrollo Dirigido por Modelos de Servicios Cloud" 12th Jornadas de Ciencia e Ingeniería de Servicios, Salamanca, España (JCIS 2016)

En el primer artículo se ha presentado una implementación del método DIARy con *Microservices*, mientras que en el segundo artículo se hace una aproximación DIARy con el paradigma de desarrollo *DevOps*. En ambos casos, mi participación directa ha consistido principalmente en escribir partes del artículo relacionadas a explicar las transformaciones de modelos que podrían realizarse además de los cambios de implementación que podrían ser necesarios para el uso de DIARy en estos contextos.

[Página intencionalmente en blanco]

6. Referencias

[1] ATL. ATL - A model transformation language. [Online]. Available: http://www.eclipse.org/atl/. [Accessed: 06-Jan-2014].

[2] Barber S. SOA Testing Challenges, 2006. [Online]. Available: http://www.perftestplus.com/resources/SOA_challenges_ppt.pdf. [Accessed: 06-Feb-2014].

[3] Bezivin J, Gerbe O. Towards a precise definition of the OMG/MDA framework, in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, 273–280, DOI:10.1109/ASE.2001.989813.

[4] Brambilla M, Cabot J, Wimmer M. *Model-Driven Software Engineering in Practice*, 1: 2012, 1–182.

[5] Erl T. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2005, 792.

[6] Erl T. *SOA: principles of service design*. Prentice Hall, 2008, 608.

[7] Bézivin J. On the unification power of models. *Softw. Syst. Model.* 2005; 4: 171–188, DOI:10.1007/s10270-005-0079-0.

[8] Cagle K. SOA is Dead? It's About Time! - O'Reilly Broadcast, 2009. [Online]. Available: http://broadcast.oreilly.com/2009/01/soa-is-dead-its-about-time.html. [Accessed: 06-Oct-2013].

[9] Chou D. Microsoft Cloud Computing Platform, 2010.

[10] CIOL. CIOL: Latest IT News | Latest Enterprise News | Latest SMB News | Hot Tech Topic news, 2008. [Online]. Available: http://www.ciol.com/ec/feature/saas-–-a-revolutionary-approach-for-building-web-applications/23108103050/0/. [Accessed: 08-Oct-2013].

[11] Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer.

[12] Guillén, J., Miranda, J., Murillo, J. M., and Canal, C., 2013, "Developing Migratable Multicloud Applications based on MDE and Adaptation Techniques" Proc. Second Nord. Symp. Cloud Computing Internet Technology - Nord. '13, pp. 30–37.

[13] Haines MN, Rothenberger MA. How a service-oriented architecture may change the software development process. *Commun. ACM* 2010; 53: 135, DOI:10.1145/1787234.1787269.

[14] Homer, A., John, S., Larry, B., Narumoto, M., & Swanson, T. (2014). Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. Microsoft patterns \& practices.
[15] Hull R, Su J. Tools for composite web services. *ACM SIGMOD Rec.* 2005; 34: 86, DOI:10.1145/1083784.1083807.
[16] Hurwitz J, Bloor R, Baroudi C, Kaufman M. *Service oriented architecture for dummies*. 2007.
[17] Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. *Sci. Comput. Program.* 2008; 72: 31–39, DOI:10.1016/j.scico.2007.08.002.
[18] Jouault F, Kurtev I. Transforming models with ATL. *Satell. Events Model. 2005 Conf.* 2006; 128–138,.
[19] Josuttis NM. *SOA in practice*. O'Reilly Vlg. Gmbh & Co., 2007, 342.
[20] Kerherve B, Nguyen KK, Gerbe O, Jaumard B. A Framework for Quality-Driven Delivery in Distributed Multimedia Systems, in Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06), 2006, 195–195, DOI:10.1109/AICT-ICIW.2006.14.
[21] Krafzig D, Banke K, Slama D. Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall, 2004, 408.
[22] Linthicum DS. Cloud computing and SOA convergence in your enterprise: a step-by-step guide. Addison-Wesley, 2010, 264.
[23] Manes AT. Application Platform Strategies Blog: SOA is Dead; Long Live Services, 2009. [Online]. Available: http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html. [Accessed: 06-Oct-2013].
[24] Mellor SJ, Scott K, Uhl A, Weise D. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, 2004, 176.
[25] Microsoft. Web.config Transformation Syntax for Web Project Deployment Using Visual Studio. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd465326(v=vs.110).aspx. [Accessed: 01-Oct-2015].
[26] Microsoft. Windows Azure Service Configuration Schema (.cscfg File). [Online]. Available: http://msdn.microsoft.com/en-us/library/windowsazure/ee758710.aspx. [Accessed: 07-Feb-2014].
[27] Miller J, Mukerji J. MDA Guide Version 1.0. 1. *Object Manag. Gr.* 2003; .
[28] MMT Project. Model to Model Transformation - MMT - Eclipsepedia. [Online]. Available: http://wiki.eclipse.org/MMT. [Accessed: 08-Feb-2014].

[29] OASIS. OASIS Security Services (SAML) TC, 2013. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security. [Accessed: 05-Oct-2013].
[30] Obeo. Model Driven Company. [Online]. Available: http://www.obeo.fr/. [Accessed: 08-Feb-2014].
[31] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. *Final Adopt. Specif. (November 2005)* 2008.
[32] OMG. MOF Model to Text Transformation Language. *2008-01-16]. http//www. omg. org/spec/MOFM2T/1.0/* 2008.
[33] OMG. Object Constrain Languaje (OCL) Specification. 2006.
[34] OMG. OMG Meta Object Facility (MOF) Core Specification. 2013.
[35] OMG. Service oriented architecture Modeling Language (SoaML) Specification. *Object Management Group*. 2012.
[36] Papazoglou M. *Web services: principles and technology*. Prentice Hall, 2007.
[37] Rimal BP, Choi E, Lumb I. A Taxonomy and Survey of Cloud Computing Systems. *2009 Fifth Int. Jt. Conf. INC, IMS IDC* 2009; 44–51, DOI:10.1109/NCM.2009.218.
[38] Sandobalín J., Zúñiga M., Insfran E., Abrahão S., Cano C.: "Una aproximación DevOps para el Desarrollo Dirigido por Modelos de Servicios Cloud" 12th Jornadas de Ciencia e Ingeniería de Servicios, Salamanca, España (JCIS 2016)
[39] Seidewitz E. What models mean. *IEEE Softw.* 2003; 20: 26–32, DOI:10.1109/MS.2003.1231147.
[40] Stahl T, Völter M. *Model-Driven Software Development - Technology, Engineering, Management*. John Wiley and Sons, Ltd., Chichester, England, 2006, 446.
[41] The Eclipse Foundation. Eclipse Open Source Community. [Online]. Available: http://www.eclipse.org/. [Accessed: 06-Jan-2014].
[42] Wettinger W., Andrikopoulos V., and Leymann F., "Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments", Conf. On the *Move to Meaningful Internet Systems (OTM)*, pp. 348–358, 2015.
[43] Wettinger J., Breitenbücher U., Kopp O., and Leymann F., "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel", *Future Generation Computer Systems*, vol.56 pp.317-332, 2016.

[44] Zúñiga, M. A., Abrahão S., Insfrán E.: A Model Driven Approach for the Dynamic Reconfiguration of Cloud Application Architectures. 24th International Conference on Information Systems Development (ISD 2015), August 25 - 27, 2015, Harbin, China.

[45] Zuñiga M. A., Abrahão S., Insfran E, "Perfil UML para el Modelado de la Integración de Servicios Cloud en Procesos de Desarrollo Incremental" *XI Jornadas de Ciencia e Ingeniería de los Servicios (JCIS)*, 2014.

[46] Zuñiga M. A., Abrahão S., Insfran E., Cano C.: "Incremental Integration of Microservices in Cloud Applications", 25th International Conference on Information Systems Development. Katowice, Poland (ISD 2016).