María Julia Blas

Un modelo de simulación para el análisis de arquitecturas de software de aplicaciones web y en la nube

Tesis Doctoral

UTN SANTA FE

CiN REUN
Red de Editoriales de Universidades Nacionales de la Argentina

LUA Libro Universitario Argentino

edUTecNe

UNIVERSIDAD TECNOLÓGICA NACIONAL

FACULTAD REGIONAL SANTA FE

- Doctorado en Ingeniería -

- Mención Ingeniería en Sistemas de Información -

UN MODELO DE SIMULACIÓN PARA EL ANÁLISIS DE ARQUITECTURAS DE SOFTWARE DE APLICACIONES WEB Y EN LA NUBE

- Tesis Doctoral -

María Julia Blas

Director: **Dr. Horacio Leone**

Codirector: **Dr. Silvio Gonnet**

Marzo, 2019

UNIVERSIDAD TECNOLÓGICA NACIONAL

FACULTAD REGIONAL SANTA FE

Se presenta esta tesis en cumplimiento de los requisitos exigidos por la Universidad Tecnológica Nacional para la obtención del grado académico de Doctor en Ingeniería, Mención Ingeniería en Sistemas de Información.

UN MODELO DE SIMULACIÓN PARA EL ANÁLISIS DE ARQUITECTURAS DE SOFTWARE DE APLICACIONES WEB Y EN LA NUBE

por

María Julia Blas

Director: ***Dr. Horacio Leone***

Codirector: ***Dr. Silvio Gonnet***

Jurados

Dr. Jorge Andrés Diaz Pace

Dr. Ernesto Kofman

Dra. Ana Rosa Tymoschuck

Marzo, 2019

a Mariel y Emilia

a Juan

Agradecimientos

"Enthusiasm is one of the most powerful engines of success. When you do a thing, do it with all your might. Put your whole soul into it. Stamp it with your own personality. Be active, be energetic, be enthusiastic and faithful, and you will accomplish your object. Nothing great was ever achieved without enthusiasm."

- Ralph Waldo Emerson

En primer lugar, quiero agradecer a mis directores *Horacio* y *Silvio* por la confianza que depositaron en todas mis "ocurrencias" a lo largo de este trayecto, alentándome siempre a continuar trabajando con nuevos proyectos e ideas, guiándome durante todo el proceso con sus consejos y experiencia.

A mi familia, *Mariel* y *Mili*, por el apoyo de todos estos años para continuar avanzando en mi vida personal y profesional. A mis tíos, *Adriana* y *Julio*, por estar siempre a mi lado. A mi abuela *Eda* por su apoyo constante. También a la familia *Sarli* por haber acompañado y compartido este trayecto.

A *Juan* por su compañía incondicional en todos mis proyectos (tanto personales como laborales), por escucharme y acompañarme siempre, por compartir viajes, congresos, estudio e innumerables momentos, no sólo en el lugar de trabajo sino también en todas las vivencias transitadas los últimos diez años.

A mis compañeros de INGAR con quienes he compartido el día a día de trabajo, investigación, asistencia a eventos, juegos deportivos, vacaciones y demás actividades científicas y no científicas. Especialmente a *Gonzalo* que se ha convertido en un amigo a lo largo de este camino, con quien he compartido muchísimas charlas y revisiones

“científicas”, viajes, compras, juntadas, meriendas “after office” y salidas. A los chicos del “comedor de 1º piso”, *Regi*, *Sergio*, *Álvaro*, *Mari*, *Cori, Danilo* y *Fede*, con quienes he compartido innumerables momentos tanto dentro como fuera del instituto. También a *Guada* que, aunque no pertenece a INGAR, siempre ha estado presente en el grupo de becarios.

A mis amigos de la vida, que están siempre incondicionalmente a mi lado (no importa lo lejos que se encuentren). Especialmente a *Iva*, *Ubaldo*, *Luci*, *Andrés*, *Maxi* y *Ale*. Además, a *Nico* que, aunque no se nos unió al “team INGAR” estuvo siempre presente durante estos cinco años.

A “los DOCS”, *Vale* y *Beto*, con quienes arrancamos este camino de posgrado hace ya un par de años, compartiendo congresos, cursos y salidas post-cursos.

A mis compañeros de cátedra, *Daniel*, *Fer*, *Diego*, *Fede*, *Gas* y *Pato*, quienes me han acompañado durante todo el trayecto. Especialmente a *Marta* que, siempre atenta a mis inquietudes, actuó como guía y apoyo en mi desarrollo como docente-investigadora.

Finalmente, al *Dr. Bernard Zeigler* por su predisposición y ayuda en la comparativa RDEVS-DEVS desarrollada como parte de esta tesis.

Prefacio

Conceptualmente, los entornos de *computación en la nube* (en inglés, cloud computing) se dividen en dos grandes componentes: *infraestructura* y *aplicación*. A nivel de *aplicación*, los arquitectos de software especifican aplicaciones web las cuales, en términos generales, quedan definidas como servicios de software. Usualmente, tales definiciones se basan en *patrones de diseño*.

El *diseño arquitectónico* puede ser considerado una de las primeras especificaciones de cualquier producto de software. Cuando sus componentes son utilizados para formular un *modelo de simulación*, la arquitectura puede ser utilizada como vehículo para estimar el comportamiento del producto final. Aún más, en los últimos años varios autores han aplicado técnicas de simulación sobre diseños arquitectónicos a fin de estimar las *propiedades de calidad de un producto de software*. Recientemente, este tipo de estrategias de análisis han sido utilizadas a nivel de infraestructura en entornos de computación en la nube. Sin embargo, los enfoques desarrollados a lo largo de los últimos años refieren *exclusivamente a un subconjunto reducido de propiedades de calidad, no incorporando el nivel de aplicación a la descripción del entorno bajo análisis*.

La formulación de un modelo de análisis que permita capturar la información de arquitecturas de software de aplicaciones web para evaluar cuantitativamente aspectos de calidad de las mismas en etapas tempranas del proceso de desarrollo, constituye un tema de interés vinculado a la Ingeniería de Software. Es decir, con el objetivo de evaluar atributos de calidad en términos de los *requerimientos no funcionales* definidos para un producto tomando como base el diseño arquitectónico, se requiere de la existencia de *modelos de análisis integrales que permitan estudiar el comportamiento de*

aplicaciones web y servicios de software a fin de evaluarlos desde una perspectiva de calidad. La especificación de modelos de simulación que brinden soporte a los conceptos vinculados a este modelo de análisis, constituye uno de los principales problemas a resolver en este trabajo de tesis.

En términos generales, existen dos aspectos relevantes que deben ser tenidos en cuenta al momento de utilizar técnicas de simulación sobre las arquitecturas de aplicaciones web, a saber: *i) Definición de las propiedades de calidad* (¿qué propiedades de calidad deben ser medidas?¿cómo se relacionan?¿qué indicadores deben ser utilizados?), y *ii) Formulación e implementación de modelos de simulación* (¿qué formalismo de simulación debe ser utilizado?¿cuántos modelos deben diseñarse?¿cómo se vinculan estos modelos?). Ambos aspectos deben ser resueltos conjuntamente en virtud de obtener un modelo de análisis útil y completo.

En esta tesis se propone un modelo de análisis para la evaluación de la calidad de aplicaciones web y servicios de software en la nube, el cual integra una *perspectiva ontológica para la definición y el estudio de las propiedades de calidad junto con una adaptación del formalismo de simulación Discrete Event System Specification (DEVS) para el diseño e implementación de los modelos de simulación requeridos para la representación de la arquitectura de software bajo estudio*. El objetivo final de este trabajo es proveer una perspectiva de Ingeniería de Software útil para la estimación de aspectos de calidad web utilizando el diseño arquitectónico como vehículo para la especificación del modelo de simulación que opera como base del modelo de análisis.

A fin de estructurar la definición de las propiedades de calidad asociadas a los productos de software, se define un nuevo tipo de documento denominado *esquema de calidad*. Este documento se basa en el modelo de calidad de producto de software definido en el *estándar ISO/IEC 25010*, quedando complementado por conceptos propios del dominio de métricas de software junto con la especificación de producto. En este sentido, se presenta una implementación del documento diseñado haciendo uso de una única *ontología* compuesta de tres modelos semánticos, a saber: un modelo semántico de calidad (*quality semantic model*), un modelo semántico de métricas de software (*metric semantic model*) y un modelo semántico de producto de software

(*software semantic model*). Las relaciones entre estos modelos se definen en términos de los conceptos incluidos en los mismos de la siguiente manera: *i)* una métrica de software (concepto definido en *metric semantic model*) es de utilidad para estimar un atributo de calidad (concepto definido en *software semantic model*), y *ii)* un atributo de calidad (definido en *software semantic model*) se encuentra asociado a una subcaracterística de calidad (concepto definido en *quality semantic model*). Cada modelo implementado incluye un *conjunto de reglas* que permiten tanto la *inferencia de nuevo conocimiento* como así también la *verificación de la integridad de las instancias* creadas a partir de la ontología específica. Además, se presenta un *conjunto de consultas* que posibilitan la *obtención de información* relevante en relación a las instancias de esquemas de calidad.

La inclusión de este instrumento como parte de la documentación asociada al desarrollo de un producto de software, incrementa la visibilidad y el entendimiento de las propiedades de calidad entorno al aplicativo específico. Dado que las aplicaciones web constituyen productos de software, *un esquema de calidad específico es propuesto como mecanismo de referencia para el estudio de sus propiedades de calidad*. Aunque distintas aplicaciones tendrán distintos requerimientos no funcionales (es decir, requerimientos de calidad), el esquema de calidad diseñado es susceptible de ser utilizado como base genérica para, posteriormente, ser extendido en casos específicos.

En este contexto, la definición del *esquema de calidad para aplicaciones web* permite identificar la información básica a ser relevada en virtud de evaluar la calidad requerida sobre el producto final. Si se combina esta información con la evaluación de arquitecturas de software por medio de simulación, se puede establecer que estos datos deben ser relevados como *salidas del modelo de simulación* a fin de obtener mediciones que puedan ser utilizadas como base de análisis. Luego, *la información que alimenta el esquema de calidad debería corresponderse con variables de salida del modelo de simulación*.

Sin embargo, *la construcción de los modelos de simulación no debe afectar* el normal desarrollo de las *tareas asociadas a los arquitectos* de software. Aunque la evaluación del ajuste de los diseños arquitectónicos a la calidad esperada es importante a fin de

evaluar la adecuación de la arquitectura propuesta en relación a los requisitos no funcionales, el diseño e implementación de los modelos de simulación requeridos para tal evaluación no puede depender de las habilidades de los desarrolladores. Por este motivo, *es importante proveer mecanismos que faciliten su construcción y desarrollo* en términos familiares a los utilizados por los arquitectos. Si el conjunto de componentes arquitectónicos se encuentra claramente definido, los modelos de simulación pueden ser construidos por medio de la aplicación de reglas de transformación.

Siguiendo este enfoque, se propone un *metamodelo* que identifica un conjunto de componentes arquitectónicos referidos a *patrones de diseño de aplicaciones web*. Complementariamente se define un *proceso de diseño basado en la teoría de sistema-de-sistemas*. Para cada uno de los elementos identificados como parte el metamodelo, se propone un modelo de simulación de eventos discretos formalizado en DEVS a fin de describir su comportamiento en virtud de medir las variables de salida identificadas como parte de la información básica requerida en el esquema de calidad web. Para una *arquitectura específica*, en función del tipo de componente arquitectónico utilizado en su diseño, es posible obtener el *modelo de componentes de simulación equivalente* por medio de la aplicación de *reglas de mapeo*. Sin embargo, dado que las conexiones entre componentes arquitectónicos reflejan dependencias, la obtención de los enlaces entre los modelos no es tan sencilla de derivar.

Por este motivo, se presenta una *adaptación del formalismo DEVS denominada Routed DEVS (RDEVS)* que permite mantener las influencias entre componentes bajo la forma de acoplamientos fijos, pero que regula el flujo de eventos entre modelos en base a una función de ruteo. En este sentido, *los modelos RDEVS utilizan información de ruteo para determinar la aceptación o el rechazo de un determinado evento*. En términos generales un modelo de simulación RDEVS solo aceptará un evento de entrada si y solo si: *i)* el emisor del evento es un modelo autorizado para su envío, y *ii)* el identificador del modelo forma parte de los destinatarios del evento. De la misma manera, sólo generará eventos de salida destinados específicamente a un subconjunto de los modelos vinculados a sus puertos de salida.

Tomando en consideración que el formalismo RDEVS se deriva como una subclase del formalismo DEVS, los modelos de componentes arquitectónicos fueron combinados con modelos RDEVS a fin de lograr una especificación completa de las arquitecturas de software de aplicaciones web en términos de modelos de simulación de eventos discretos. De esta forma, por medio de la aplicación de un conjunto de lineamientos básicos, los modelos de simulación referidos a arquitecturas web son obtenidos directamente de la configuración de componentes arquitectónicos utilizados en el diseño. Luego, el modelo de análisis propuesto utiliza la arquitectura para estructurar el modelo de simulación asociado. La *validación de estos modelos* se realiza tomando como referencia dos *patrones de diseño arquitectónico* que dan origen a las aplicaciones web desarrolladas en entornos de computación en la nube. Para esto, se utiliza el esquema arquitectónico de un subconjunto de patrones de diseño, sobre los cuales se generan múltiples escenarios alternativos con el objetivo de comparar el *nivel de cumplimiento* de las *características propias de cada patrón* en relación al comportamiento esperado de los mismos.

Complementariamente a los modelos de simulación arquitectónica, se describe brevemente un conjunto de modelos de comportamiento asociados a distintos tipos de usuarios que representan patrones temporales para la generación de los eventos de entrada requeridos. Tales modelos se especificaron haciendo uso del formalismo DEVS, por lo que tienen completa compatibilidad con los modelos RDEVS diseñados en base al diseño arquitectónico.

De esta manera, en base a las propiedades de calidad requeridas para estimar el comportamiento de una aplicación web, se propone un conjunto de modelos de simulación de eventos discretos que permite evaluar la calidad futura de una arquitectura específica. Esto posibilita ajustar su diseño a los requerimientos no funcionales en etapas tempranas del proceso de desarrollo.

Como parte de la solución propuesta en esta tesis al problema de evaluación de arquitecturas de software en entornos de computación en la nube, se han llevado a cabo numerosos trabajos previos. Estos trabajos han sido divulgados por medio de

publicaciones en revistas científicas y actas de congresos (nacionales e internacionales) presentados en diferentes jornadas del ámbito científico, a saber:

- *Publicaciones en revistas científicas*:

  - Blas, M.; Gonnet, S.; Leone, H. (2017). "An Ontology to Document a Quality Scheme Specification of a Software Product". *Expert Systems*, 34(5). *https://doi.org/10.1111/exsy.12213*.
  - Blas, M.; Gonnet, S.; Leone, H. (2017). "Modeling User Temporal Behaviors Using Hybrid Simulation Models". *IEEE Latin America Transactions*, 15(2), 341-348.

- *Publicaciones en actas de congresos:*

  - Blas, M. J.; Gonnet S.; Leone H.; Zeigler B. (2018). "A Conceptual Framework to Classify the Extensions of DEVS Formalism as Variants and Subclasses". En *Proceedings of the 2018 Winter Simulation Conference*, editado por M. Rabe, A. Juan, N. Mustafee, A. Skoogh, S. Jain y B. Johansson, 560-571. Piscataway, New Jersey: IEEE.
  - Blas, M. J.; Gonnet S.; Leone H. (2017). "Routing Structure Over Discrete Event System Specification: A DEVS Adaptation To Develop Smart Routing In Simulation Models". En *Proceedings of the 2017 Winter Simulation Conference*, editado por W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer y E. Page, 774-785. Piscataway, New Jersey: IEEE.
  - Blas, M. J. (2017). "An Analysis Model to Evaluate Web Applications Quality Using a Discrete-Event Simulation Approach". En *Proceedings of the 2017 Winter Simulation Conference*, editado por W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer y E. Page, 4648-4649. Piscataway, New Jersey: IEEE.
  - Blas, M.; Gonnet, S.; Leone, H. (2017). "Definición de un Proceso de Diseño basado en un Metamodelo para la Especificación de Arquitecturas de Software

de Entornos Cloud Computing Utilizando el Enfoque de "Sistema-de-Sistemas". En *Actas del 5º Congreso Nacional de Ingeniería Informática / Sistemas de Información (CONAIISI)*, RIISIC, Santa Fe, Argentina.

- Blas, M.; Gonnet, S.; Leone, H. (2016). "Building Simulation Models to Evaluate Web Application Architectures". En *Proceedings of the XLII Latin American Computing Conference (CLEI)*, Valparaíso, Chile. https://doi.org/10.1109/CLEI.2016.7833339.
- Blas, M.; Gonnet, S.; Leone, H. (2016). "Especificación de la Calidad en Software-as-a-Service: Definición de un Esquema de Calidad basado en el Estándar ISO/IEC 25010". En *Proceedings of the 2016 Argentine Symposium on Software Engineering (ASSE)*, 45º JAIIO, Ciudad Autónoma de Buenos Aires, Argentina.
- Blas, M.; Gonnet, S.; Leone, H. (2015). "Un Modelo para la Representación de Arquitecturas Cloud basadas en Capas por medio de la Utilización de Patrones de Diseño: Especificación de la Capa de Aplicación". En *Actas del 3º Congreso Nacional de Ingeniería Informática / Sistemas de Información (CONAIISI)*, RIISIC, Ciudad Autónoma de Buenos Aires, Argentina.
- Blas, M.; Gonnet, S. (2015). "An Ontological Approach to Analyze the Data Required by a System Quality Scheme". En *Proceedings of the 1st Argentine Symposium on Ontologies and their Applications (SAOA)*, 44º JAIIO, Rosario, Argentina.
- Blas, M.; Gonnet, S.; Leone, H. (2014). "Una Taxonomía de Atributos de Calidad para la Evaluación de Arquitecturas de Software por medio de Simulación". En *Actas del 2º Congreso Nacional de Ingeniería Informática / Sistemas de Información (CONAIISI)*, RIISIC, San Luis, Argentina.

Estos trabajos pueden ser considerados resultados parciales del trabajo de investigación realizado.

Organización del Documento

Esta tesis se encuentra estructurada en cinco partes, cada una de las cuales queda conformada por un conjunto de capítulos. Cada una de las partes propuestas abarca un conjunto de elementos básicos del trabajo realizado, dando lugar a una lectura incremental de los conceptos y componentes desarrollados a fin de describir la construcción de los modelos de simulación propuestos. De esta manera, las partes organizan independientemente los distintos componentes de la propuesta, mientras que cada capítulo introduce al lector en un conjunto de elementos relacionados que son aplicados al componente que conforma el modelo resultante.

La Parte I comprende los Capítulos 1 a 3, los cuales describen los conceptos básicos relacionados con la propuesta. El Capítulo 1 introduce al lector en las nociones básicas de calidad en relación al software, diferenciando términos como calidad de producto/calidad de proceso y calidad interna/calidad externa. El Capítulo 2 se centra específicamente en los problemas que existen para estimar la calidad en servicios de software en la nube, dando lugar a los fundamentos de esta tesis. El modelo de análisis diseñado para obtener los modelos de simulación se esquematiza en el Capítulo 3, presentando formalmente el conjunto de elementos que componen la solución propuesta. Tales elementos dan lugar a los apartados subsiguientes.

La Parte II establece los lineamientos necesarios para comprender el dominio de calidad en relación a productos de software en la nube. El Capítulo 4 define formalmente el contexto de calidad para productos de software genéricos, presentando una estrategia para la documentación de los aspectos de calidad a evaluar a lo largo del proceso de desarrollo. Para esto, define un nuevo tipo de documento denominado "esquema de calidad", el cual integra en una única formulación aspectos relacionados a las propiedades de calidad, su medición y los artefactos de software relacionados. En base a esta definición, el Capítulo 5 introduce la noción de ontologías como mecanismo para la definición de esquemas de calidad genéricos. Haciendo uso de este modelo semántico, el Capítulo 6 extiende el esquema de calidad previo incorporando características y

subcaracterísticas de calidad propias del dominio de computación en la nube al modelo de calidad utilizado como base para la definición de la ontología. Además, define un conjunto de métricas de software útiles para el relevamiento de dichas propiedades. Puntualmente, centra su atención en el estudio de la calidad en relación a los servicios de software y las aplicaciones web. Esta definición es, posteriormente, tomada como punto de partida al construir los modelos de simulación.

La Parte III se centra en el estudio de estrategias de representación de arquitecturas de software en entornos de computación en la nube, para lo cual incluye el contenido de los Capítulos 7 y 8. El Capítulo 7 esquematiza, haciendo uso de un estilo arquitectónico basado en capas, la arquitectura de alto nivel utilizada en entornos de computación en la nube. Para esta arquitectura, se presenta un proceso de diseño definido en términos de pasos, pautas y lineamientos en base a la teoría de sistema-de-sistemas. Este proceso abstrae las actividades de diseño arquitectónico en términos de sistemas componentes aplicando una estrategia de co-diseño. Siguiendo esta línea, el Capítulo 8 introduce patrones de diseño de aplicaciones web, los cuales son tomados como base para formular un metamodelo de componentes (elementos y conexiones) frecuentemente utilizados en la construcción de arquitecturas de software de aplicaciones web.

La Parte IV se aboca al diseño e implementación de los modelos de simulación, haciendo hincapié en los formalismos de especificación utilizados para tales fines (Capítulo 9). El Capítulo 10 presenta los modelos diseñados para representar la arquitectura de software especificada en términos de los componentes identificados en el apartado precedente, dando lugar a un conjunto de modelos definidos en base al formalismo Routed DEVS. Adicionalmente, el Capítulo 11 detalla una prueba de conceptos de los modelos desarrollados a fin de validar la propuesta en términos del modelado y simulación de dos patrones de diseño altamente difundidos en la literatura.

La Parte V se encuentra dedicada a las conclusiones y líneas de investigación futuras que se desprenden del trabajo realizado (Capítulo 12).

Contenido

Lista de Figuras

Lista de Tablas

Parte I

Conceptos Fundamentales

Capítulo 1. Introducción

El estudio de la calidad de software es tan antiguo como el estudio del software en sí mismo (Deissenboeck y colab. 2009). En este capítulo se introducen conceptos básicos referidos a la calidad de software, realizando una comparativa entre las diferentes acepciones, su importancia y la forma en la cual puede analizarse cada una de las perspectivas involucradas. Se analizan las dificultades que existen al intentar evaluar la calidad de productos de software en las etapas del proceso de desarrollo, detallándose además la forma en la cual el diseño de la arquitectura de software impacta en la obtención de los distintos atributos de calidad.

1.1 Calidad del Software

La calidad a nivel de software es subjetiva (Barbacci y colab. 1995). La falta de consenso sobre su significado y la ausencia de un formalismo, método o proceso específico para su medición, han dado lugar a numerosos esfuerzos por lograr unanimidad sobre su ámbito y alcance.

En este contexto, diversos autores y organismos han formulado definiciones aplicables al concepto. La IEEE define *calidad de software* como "el grado en el cual un software posee una combinación de atributos deseados" (IEEE 1061 1998). Por su parte, Pressman establece que la *calidad de un sistema de software* es "la concordancia del sistema con los requisitos funcionales y de rendimiento explícitamente establecidos,

con los estándares de desarrollo explícitamente documentados y con las características implícitas que se esperan de todo software desarrollado profesionalmente" (Pressman 2010). Una definición similar se presenta en (Albin 2003), donde se especifica que la *calidad de un software* refiere a "una característica directamente relacionada con la habilidad del sistema para satisfacer sus requerimientos funcionales y no funcionales, tanto implícitos como explícitos".

Aunque existen ciertas diferencias entre las definiciones precedentes, en todas ellas se evidencia la necesidad de *evaluar la calidad del software en función del estudio de los atributos asociados a los requerimientos (funcionales y no funcionales) del sistema*. Esta particularidad guarda relación con el concepto de calidad genérico (aplicable a cualquier tipo de producto), en el cual se enuncia que debe entenderse por *calidad* al "grado en el cual, el conjunto de características inherentes del producto en cuestión, cumple con los requerimientos planteados".

Sin embargo, aunque esta definición genérica brinda un marco de trabajo adecuado para analizar la calidad de los productos de software, al analizar la calidad en el contexto de desarrollo de software se deben contemplar dos niveles de abstracción diferentes, a saber: *i) calidad de diseño* (conjunto de características específicas de un único elemento) y, *ii) calidad de concordancia* (grado de cumplimiento de las especificaciones de diseño durante su construcción) (Pressman 2010). De esta manera, mientras que la calidad de diseño comprende los requisitos, las especificaciones y el diseño del sistema de software; la calidad de concordancia apunta a la implementación del mismo. En conjunto, ambas abstracciones conforman la calidad global del sistema de software.

Luego, es evidente que la calidad es un aspecto que moldea todos los aspectos del proceso de desarrollo de un sistema de software (Pressman 2010). Desde el relevamiento de requerimientos hasta la ejecución del sistema, sin importar la metodología de desarrollo seleccionada, es crítico que el equipo de trabajo comprenda

la importancia de generar software de alta calidad. Para lograr esto, es importante garantizar que:

- El equipo de trabajo conoce claramente las propiedades de calidad genéricas aplicables a cualquier tipo de software (junto con sus relaciones e influencias).
- La definición de los requerimientos de calidad de un producto específico se plantea como una tarea prioritaria a lo largo del proceso de desarrollo, comenzándose a trabajar en etapas tempranas (por ejemplo, desde la elicitación de requerimientos).

Un *atributo de calidad* es una caracterización o propiedad específica de un sistema de software que puede tomar un valor cuantitativo o cualitativo, el cual es medible u observable (Albin 2003). Cuando se especifica una valoración para un atributo de calidad (es decir, un valor deseable para el mismo), se obtiene un *requerimiento de calidad*. Teniendo en cuenta que entre los distintos atributos de calidad existen relaciones e interacciones, no es posible que un sistema de software complejo alcance un único atributo de calidad de forma aislada. Luego, existen distintos esquemas de trabajo que buscan explicitar las vinculaciones entre atributos.

Una taxonomía de atributos de calidad se denomina *modelo de calidad* (Wagner 2013). Estos modelos sirven como marco de trabajo para la especificación, evaluación y testeo de la calidad de un sistema de software, ya que definen de forma jerárquica un conjunto de propiedades de calidad con el objetivo de simplificar la complejidad de su estudio mediante la descomposición (Scalone 2006). De esta manera, un modelo de calidad que agrupa atributos relacionados a artefactos intermedios del proceso de desarrollo, es un *modelo de calidad a nivel de diseño*; mientras que un modelo que trabaja sobre la calidad de la implementación desarrollada en base a los requerimientos de calidad explicitados, es un *modelo de calidad a nivel de concordancia*. Ambos modelos son relevantes a lo largo del proceso de desarrollo para comprender las relaciones que existen entre la calidad de lo desarrollado y los requerimientos

planteados. Luego, este tipo de modelos es de utilidad para detectar un conjunto de propiedades en un software a fin de obtener una indicación de su calidad (Scalone 2006).

En este contexto, a fin de comprender la utilidad de los modelos de calidad existentes, es importante diferenciar las distintas acepciones que se desprenden del estudio de la calidad del software. Luego, al definir *calidad de software* se debe diferenciar entre *calidad del producto de software* y *calidad del proceso de desarrollo*. A su vez, partiendo de la calidad de producto, es posible distinguir *calidad interna*, *calidad externa* y *calidad de uso*.

1.1.1 Calidad de Proceso y Calidad de Producto

Diversos autores han trabajado para establecer una diferenciación entre la calidad del proceso de desarrollo y la calidad del producto de software (Kitchenham y Pfleeger 1996; Harter, Krishnan y Slaughter 2000; Pressman 2010; Muñoz, Velthuis y Moraga de la Rubia 2010). Sin embargo, ambos conceptos se vinculan al *valor técnico* del software ya que refieren a las particularidades propias del mismo que son independientes de su *valor comercial*.

En esencia, la *calidad del proceso de desarrollo* refiere a las actividades que influyen en la calidad del producto final. Es decir, para analizar la calidad de proceso se deben hacer preguntas del tipo:

- ¿Cómo pueden encontrarse los defectos en etapas tempranas del desarrollo?
- ¿Existen actividades adicionales que, en caso de incorporarse, mejoren la calidad del producto?
- ¿Cuáles son las actividades en las que se introducen mayor cantidad de defectos?

Como base para la realización de este tipo de análisis se utilizan modelos basados en proceso como, por ejemplo, *Capability Maturity Model Integration* (CMMI Institute

2018). Ocasionalmente también se utiliza el modelo *Software Process Improvement and Capability Determination* propuesto en (ISO/IEC 15504 2004). En la actualidad, este último modelo se encuentra siendo revisado y reformulado como parte de la familia de estándares ISO/IEC 33000 (ISO/IEC 33000 2015).

Por su parte, la *calidad del producto de software* se vincula con las propiedades requeridas según el punto de vista de los distintos grupos de interés hacia los cuales se encuentra destinado el software. Es, normalmente, el tipo de calidad que se evalúa por medio del uso de *métricas de software*. Tal como se detalla en el siguiente apartado, involucra tres perspectivas de análisis (interna, externa y de uso). Para cada caso, existen modelos de calidad específicos que definen los aspectos no funcionales más relevantes como, por ejemplo, los definidos como parte de la familia de estándares ISO/IEC 25000 (ISO/IEC 25000 2014).

Aunque los conceptos difieren, las técnicas para lograr calidad son aplicables tanto al proceso como al producto. Aún más, usualmente al mejorar la calidad del proceso de desarrollo se logra un impacto positivo sobre la calidad del producto. Sin embargo, la calidad del proceso solo conlleva a mejorar un subconjunto de atributos de calidad de producto. Por ejemplo, si se aplica un proceso de verificación apropiado es posible garantizar un nivel adecuado de *robustez*. Los atributos de calidad de producto vinculados a las funciones del sistema de software que se corresponden con el problema a resolver (como por ejemplo, la *modificabilidad*) quedan exentos de este subconjunto. No obstante, las metas que se establezcan en relación a la calidad del producto determinan las metas requeridas para la calidad del proceso de desarrollo. Esto se debe a que la calidad del producto se define en función de la calidad del proceso. Sin un buen proceso de desarrollo es casi imposible obtener un buen producto.

1.1.2 Calidad Interna, Calidad Externa y Calidad de Uso

Según la familia de estándares ISO/IEC 25000, la *calidad de un producto de software* puede evaluarse de acuerdo a tres perspectivas, a saber: *i)* midiendo atributos internos (medidas estáticas de productos intermedios), *ii)* midiendo atributos externos (estudio del comportamiento del producto en ejecución), y *iii)* midiendo atributos de calidad en uso (evaluación durante la utilización efectiva por parte de los usuarios finales) (ISO/IEC 25000 2014). Tales perspectivas refieren a la *calidad interna*, *calidad externa* y *calidad de uso*, respectivamente.

Se entiende por *calidad interna* a la totalidad de características del software que, generalmente, permanecen constantes a lo largo del ciclo de vida. Para su medición se utilizan métricas internas, las cuales evalúan aspectos estáticos del producto (es decir, no abarcan ni su comportamiento ni su entorno, sino que se derivan del producto en sí mismo). Normalmente se evalúa en las etapas tempranas del ciclo de vida del producto, como ser las fases de análisis, diseño y codificación.

Por su parte, la *calidad externa* refiere a las características del software que se evalúan cuando el producto se encuentra en ejecución por medio de pruebas en un ambiente controlado utilizando datos de prueba reales. Para esto se utilizan métricas externas, las cuales tienen en cuenta el comportamiento del producto en un entorno de ejecución específico. En este caso, la evaluación se realiza sobre el producto de software en etapas finales del ciclo de vida, como ser las fases de implementación, integración, pruebas y mantenimiento.

Finalmente, la *calidad de uso* representa el conjunto de atributos relacionados con la aceptación del producto por parte de los usuarios finales. En este sentido, refiere a la capacidad del producto de software de facilitar a usuarios concretos, en un contexto de uso dado, la obtención de metas específicas con efectividad, productividad, seguridad y satisfacción. Las métricas de calidad en uso calculan los efectos de la utilización del software en un determinado campo, por lo que se relevan en escenarios reales. Tales

métricas evalúan si el producto cumple (o no) con las necesidades requeridas por los usuarios. El objetivo de relevar esta clase de calidad no es necesariamente alcanzar una calidad perfecta, sino la necesaria y suficiente para los contextos de uso de los usuarios.

(CALIDAD DE PRODUCTO DE SOFTWARE) *Grado en el cual un producto de software posee una combinación de atributos requeridos según el punto de vista de los distintos grupos de interés hacia los cuales se encuentra destinado.*

1.2 Dificultades al Evaluar la Calidad en Productos de Software

Es importante destacar que la calidad del software debe ser considerada en todos sus estados de evolución; desde la especificación y el diseño, hasta la implementación y prueba (Pressman 2010). Es decir, evaluar la calidad del producto final una vez construido no es suficiente ya que, en esta instancia del proceso de desarrollo, los defectos y/o errores que lleven a problemas de calidad suelen no tener solución (o su solución es mucho más costosa que una solución semejante aplicada en etapas tempranas del desarrollo).

Luego, el objetivo que se persigue al evaluar la calidad de forma continua en los productos de software es *garantizar que el producto bajo desarrollo tenga el efecto requerido en los contextos de interés*. Tales efectos se traducen en costos directos e indirectos, aplicables al desarrollo y mantenimiento del mismo.

Sin embargo, la naturaleza de los productos de software conlleva a la existencia de un conjunto de problemas asociados a su estudio (Scalone 2006), a saber:

- Constantes incrementos en los costes de desarrollo y en los plazos de desarrollo debido entre otros, a los bajos niveles de productividad.
- Aumento constante del tamaño y complejidad de los programas desarrollados.

- Carácter dinámico e iterativo a lo largo de su ciclo de vida (el software cambia o evoluciona de acuerdo a versiones que buscan mejorar las prestaciones implementadas).
- Imposibilidad de lograr un producto depurado al 100%.
- Falta de procedimientos normalizados para estipular y evaluar la calidad, costes y productividad.

En este contexto, los problemas más comunes que dificultan la especificación, diseño, medición y prueba de los *atributos de calidad* entorno a un producto de software específico incluyen (Albin 2003):

- Entendimiento erróneo sobre la importancia de tales atributos.
- Lenguaje inadecuado para expresar y especificar los requerimientos de calidad.
- Notaciones y métodos de modelado inadecuados para expresar las soluciones aplicables a atributos de calidad específicos.
- Dificultades para generar diseños basados en atributos de calidad.
- Falta de documentación en el diseño y en los patrones arquitectónicos que brindan soporte a las posibles configuraciones.
- Realización del control de calidad como última etapa del proyecto.

1.2.1 Importancia de los Atributos de Calidad

Normalmente, para lograr un estudio adecuado de la calidad, los atributos se agrupan en *categorías*. Independientemente de la denominación y de la cantidad de categorías utilizadas, siempre debe cumplirse que:

- La obtención de un atributo de calidad asociado a una categoría no garantiza nada acerca de: *i)* los atributos de calidad de las categorías restantes, ni de *ii)* los otros atributos de calidad de la misma categoría.
- El logro de un atributo de calidad nunca puede obtenerse de forma aislada.

- El alcanzar un atributo de calidad puede tener un efecto positivo o negativo sobre los atributos de calidad restantes.

En este contexto, debe tenerse en cuenta que las funcionalidades del producto de software (es decir, la habilidad del sistema de hacer el trabajo para el cual fue pensado) son ortogonales con respecto a los atributos de calidad (Bass, Clements y Kazman 2012). Aun así, idealmente, los requerimientos de calidad deben especificarse junto con los requerimientos funcionales de la forma menos ambigua posible. Esto implica que para su evaluación deben utilizarse métricas específicamente diseñadas para su relevamiento.

Todo atributo de calidad tiene asociado un conjunto de métricas (Albin 2003). Una métrica es una medida o escala cuantitativa o cualitativa asociada a un atributo de calidad que incluye un método o técnica aplicable para su obtención. Múltiples métricas son aplicables a un mismo atributo de calidad en distintas etapas del proceso de desarrollo.

1.3 Calidad de Producto y Arquitectura de Software

Como se ha establecido con anterioridad, la calidad de los productos de software es una particularidad que se desprende de la definición de sus requerimientos -tanto funcionales como no funcionales, implícitos y explícitos- (Albin 2003). Sin embargo, debido a que el conjunto de requerimientos se obtiene y perfecciona a medida que se lleva a cabo el proceso de solución, resulta difícil diseñar productos adaptables a todos los posibles cambios que se puedan dar en relación a las solicitudes de los grupos de interés. Aunque se han desarrollado numerosas técnicas y modelos para reducir la ambigüedad al elicitar requerimientos, la determinación del nivel de adecuación de las decisiones tomadas con respecto a un producto específico, resulta a menudo complicada y proclive a errores (aun contando con la guía de metodologías específicas).

En etapas de desarrollo tempranas, alcanzar un nivel de calidad apropiado implica lograr un diseño arquitectónico adecuado a la naturaleza de los atributos de calidad relevantes. Teniendo en cuenta que el diseño arquitectónico se corresponde con: *i)* la primera abstracción del producto de software a ser construido y, *ii)* el modelo fundamental sobre el cual se desarrollan los distintos componentes de software; es importante que dicho diseño integre de forma adecuada los aspectos funcionales y no funcionales (propiedades de calidad) del producto en cuestión.

En este sentido, de acuerdo con (Bass, Clements y Kazman 2012), los diseños de arquitecturas de software:

- Definen restricciones de implementación.
- Dictan la estructura organizacional del producto.
- Impactan sobre los distintos atributos de calidad.
- Permiten predecir cualidades del sistema de software bajo desarrollo.
- Facilitan el estudio del impacto de cambios sobre el producto.
- Ayudan a la realización de prototipos evolutivos.
- Posibilitan la elaboración de estimaciones de costos y planificaciones precisas.

Luego, al construir un diseño arquitectónico es necesario seleccionar los principios de trabajo adecuados a fin de lograr un balance apropiado entre las distintas propiedades requeridas que compiten entre sí. Esta tarea no es sencilla. La formulación de diseños de arquitecturas de software se caracteriza por su complejidad, donde el punto de partida es un conjunto de objetivos que no se encuentra claramente definido, cuya especificación se obtiene a medida que se lleva a cabo el proceso de solución (Jacobson 1999; Clarke y colab. 2003; Gonnet, Henning y Leone 2007; Roldán, Gonnet y Leone 2013; Roldán, Gonnet y Leone 2016; Vegetti y colab. 2016).

1.3.1 Arquitectura de Software

De acuerdo con (Bass, Clements y Kazman 2012), la arquitectura de un producto de software comprende los componentes del software, las propiedades visibles de dichos componentes y las relaciones que existen entre ellos.

Esta definición es indiferente al hecho de que el diseño arquitectónico elaborado sea bueno o malo (es decir, permita o evite que el producto de software cumpla con los requerimientos de calidad identificados). En esencia, el diseño arquitectónico se concentra en vincular los atributos de calidad y las características del sistema de software a ser construido, seleccionando un conjunto de lineamientos fundamentales que ayuden a balancear los distintos aspectos relevados. Para esto, el arquitecto de software debe: *i)* contar con la habilidad requerida para evaluar cuantitativamente los atributos de calidad sobre un diseño funcional dado, y *ii)* en base al resultado de esta evaluación, realizar una compensación de características que le permitan alcanzar el mejor comportamiento posible en el sistema de software.

Por este motivo, las arquitecturas de software deben ser evaluadas a fin de determinar su habilidad para cumplir con los atributos de calidad relevantes del sistema de software bajo desarrollo. En este sentido, el proceso de evaluación de estos diseños no debe centrarse en la aplicación de una única métrica universal (inexistente), sino que debe cuantificar atributos individuales para, luego, realizar una compensación sobre las métricas obtenidas (Barbacci y colab. 1995).

(ARQUITECTURA DE PRODUCTO DE SOFTWARE) *Comprende los componentes del software, las propiedades visibles de dichos componentes y las relaciones que existen entre ellos (Bass, Clements y Kazman 2012).*

1.3.2 Aplicación de Arquitecturas Genéricas y Patrones de Diseño

Frecuentemente, a fin de evitar los conflictos derivados de la falta de pericia al momento de generar un diseño, los arquitectos de software deciden trabajar en base a arquitecturas genéricas. Una *arquitectura genérica* provee una estructura para una familia de problemas, encapsula propiedades inherentes a un dominio específico y guía el proceso de diseño hacia la obtención de la calidad. De esta manera, una "buena" estructura: *i)* reduce la posibilidad de obtener un diseño inflexible, y *ii)* permite a los diseñadores concentrarse en los aspectos importantes del sistema bajo desarrollo. Por el contrario, una "mala" estructura fuerza al diseñador a tomar decisiones que limitan (a futuro) las posibilidades de modificar o alterar el diseño a un costo razonable.

En este contexto los patrones arquitectónicos son los bloques constructivos básicos de una arquitectura de software. Un *patrón de diseño arquitectónico* define una familia de sistemas en base a un esquema de organización estructural (Garlan y Shaw 1994). Es decir, determina un vocabulario de componentes y conectores junto con un grupo de restricciones que indican la forma en la cual estos elementos pueden ser utilizados y combinados. Las arquitecturas de sistemas complejos suelen basarse en más de un patrón (quedando definidos en base a una composición de estos últimos). La identificación de los patrones adecuados es una tarea importante que debe ser llevada a cabo por el arquitecto a cargo de la construcción del diseño. Tal responsabilidad depende de la pericia y experiencia del mismo en el dominio de trabajo.

Independientemente del uso de patrones o arquitecturas genéricas, las arquitecturas resultantes del proceso de diseño deben ser evaluadas a fin de determinar el grado de ajuste de la estructura definida en relación a la calidad requerida.

(PATRÓN DE DISEÑO ARQUITECTÓNICO) *Define una familia de sistemas de software haciendo uso de un esquema de organización estructural (Garlan y Shaw 1994).*

1.3.3 Evaluación de Arquitecturas de Software (Métodos Tradicionales)

La evaluación del diseño arquitectónico de un sistema de software previo a su implementación es una buena práctica ingenieril. En este sentido, toda técnica que permita valorar el ajuste de una arquitectura candidata al sistema de software bajo desarrollo es de gran importancia (Barbacci y colab. 1995).

En este contexto, múltiples autores han definido distintos tipos de estrategias para llevar a cabo dicha evaluación. Desde un punto de vista genérico existen dos clases de estrategias básicas, a saber: *i)* estrategias basadas en escenarios, y *ii)* estrategias basadas en medición cuantitativa o cualitativa.

En el primer caso, los métodos más difundidos para la evaluación de los diseños arquitectónicos son:

- *El método de análisis de arquitecturas de software basado en escenarios (SAAM por sus siglas en inglés, scenario-based architecture analysis method) propuesto en (Kazman y colab. 1996):* Este método plantea la evaluación de arquitecturas de software utilizando escenarios, los cuales corresponden a evaluaciones cualitativas de una arquitectura. Estos escenarios son necesarios pero no suficientes para predecir y controlar múltiples atributos de calidad. Por este motivo, deben ser complementados con otras técnicas de evaluación (por ejemplo, preguntas orientadas a la cuantificación de los indicadores de calidad prioritarios de cada escenario).
- *El método de análisis de compensación de la arquitectura (ATAM por sus siglas en inglés, architecture tradeoff analysis method) propuesto en (Kazman y colab. 1998)*: Este método es planteado como una mejora de SAAM que incorpora categorías y árboles de utilidad en relación a los atributos de calidad a ser estudiados sobre la arquitectura. Corresponde a uno de los métodos de evaluación de arquitecturas de software más difundidos.

- *El método de análisis costo/beneficio (CBAM por sus siglas en inglés, cost benefit analysis method) propuesto en (Kazman, Asundi y Klien 2002)*: Este método se basa en los artefactos producidos por ATAM, modelando los costos y beneficios de las decisiones arquitectónicas utilizadas a fin de obtener una curva definida en términos de su utilidad/respuesta. Las estrategias/tácticas arquitectónicas proveen una utilidad, donde cada una de ellas tiene un costo y un tiempo de implementación asociado.

Por su parte, en el caso de las estrategias de evaluación basadas en medición es importante destacar que, aunque pueden relevarse mediciones de carácter cuantitativo, la mayoría de las características relevadas a nivel de diseño arquitectónico son cualitativas (Albin 2003). En la literatura pueden encontrarse diversas técnicas asociadas a diferentes atributos de calidad específicos (donde cada atributo requiere un tratamiento especial sobre el diseño propuesto).

En este sentido, el trabajo propuesto en (Wang, Wu y Chen 1999) emplea modelos de *Procesos de Markov* para evaluar la "confiabilidad" en base a los estilos arquitectónicos utilizados para construir el diseño. Otros trabajos, como por ejemplo el propuesto en (Spitznagel y Garlan 1998), han optado por la aplicación de la *Teoría de Colas* para medir el "rendimiento" (utilizando el tiempo promedio de arribo de las peticiones a los componentes de software en combinación con el tamaño de la cola). Más recientemente, las *Redes de Petri* han sido aplicadas para medir la "seguridad", "confiabilidad" y el "rendimiento" (Fukuzawa y Saeki 2002); mientras que otras propuestas han utilizado la *Teoría de Grafos* para analizar el atributo de calidad "modificabilidad" (Bachmann y colab. 2004).

Limitaciones de los Métodos Tradicionales

Aunque los métodos detallados en el apartado precedente son útiles para la evaluación de arquitecturas de software, poseen limitaciones vinculadas a:

- La imposibilidad de evaluar múltiples atributos de calidad haciendo uso de una única técnica o método detrás del proceso de evaluación.
- Restricciones para representar un subconjunto de escenarios de desempeño del sistema de software en el mundo real (es decir, se presentan dificultades para construir los modelos requeridos de forma tal que se garantice un adecuado nivel de abstracción de las características presentes en el contexto real).
- Incorporación de las complejidades propias de la técnica de trabajo utilizada para construir el modelo.
- Incremento de los tiempos de desarrollo debido a que se debe incorporar la construcción y validación del modelo abstracto requerido para aplicar la técnica de evaluación sobre el diseño arquitectónico.

En este sentido, por ejemplo, los enfoques basados en *Procesos de Markov* incluyen en la representación de los estados únicamente componentes del modelo de la arquitectura (dejando de lado otros aspectos importantes del dominio arquitectónico) y requieren de la resolución analítica de los modelos (lo que hace tedioso el proceso de evaluación) (Immonen y Niemelä 2008; Singh, Tripathi y Vinod 2011). En el caso de la *Teoría de Colas* se posibilita la realización de una buena evaluación del atributo "rendimiento", pero es complicado representar con este formalismo otros aspectos de la calidad del software (tales como "seguridad" y "confiabilidad"). De forma similar, las *Redes de Petri* son útiles únicamente cuando se las utiliza como base para la evaluación de sistemas simples (ya que en sistemas complejos la tarea de resolución se dificulta). Además, estas redes requieren de la simplificación del problema modelado y, usualmente, este tipo de simplificaciones da lugar a la pérdida de información de las características del sistema de software que está siendo analizado (afectando los resultados finales de la evaluación).

Teniendo en cuenta estas limitaciones, recientemente se han comenzado a desarrollar nuevas estrategias de evaluación centradas en la combinación de técnicas

preexistentes a fin de contribuir no solo a la evaluación de las arquitecturas en términos de distintos atributos de calidad, sino también al dinamismo constante con el cual se producen los cambios en los productos de software.

1.3.4 Evaluación de Arquitecturas de Software (Métodos Complementarios)

A fin de complementar las limitaciones presentes en los métodos de evaluación tradicionales y, en virtud de contribuir al proceso de desarrollo de software con información referida a la calidad del producto bajo desarrollo en las distintas etapas de trabajo, se han realizado numerosos esfuerzos para estimar la adecuación del sistema de software final a requerimientos de calidad individuales a partir del uso de técnicas computacionales aplicadas sobre artefactos intermedios (obtenidos como resultados parciales del proceso de desarrollo). Un producto intermedio es cualquier modelo, especificación, documento o código fuente que es creado y utilizado durante el proceso de construcción del sistema de software (Albin 2003).

La idea detrás de estos métodos complementarios es, por medio de un proceso de evaluación predefinido, proveer una predicción razonable de uno o más atributos de calidad del sistema de software objetivo. En este contexto, múltiples autores han llevado a cabo investigaciones tendientes a mejorar el estudio de la calidad del software. Por ejemplo, en (Dargan y colab. 2014) se utiliza la especificación de requerimientos como base para la predicción del atributo “rendimiento”; mientras que en (Roshandel, Medvidovic y Golubchik 2007) se hace uso de la arquitectura de software para predecir la “confiabilidad” del producto resultante. En este último caso, la aplicación de la técnica propuesta por los autores también se corresponde con una *estrategia de evaluación de arquitectura de software* (ya que permite valorar el ajuste de una arquitectura candidata con los factores de interés asociados a los atributos de calidad requeridos en el sistema de software bajo desarrollo) (Bass, Clements y Kazman 2012).

En la última década, varios autores han utilizado técnicas de simulación como complemento a las metodologías tradicionales a fin de evaluar diseños arquitectónicos en base a una estimación de las propiedades de calidad (Koziolek y colab. 2009; Meiappane, Chithra y Venkataesan 2013; Bogado, Gonnet y Leone 2014). Bajo esta perspectiva, se toma el diseño arquitectónico como base para la construcción de un modelo de simulación específico, por medio del cual se evalúa el comportamiento del sistema diseñado conforme un conjunto de atributos de calidad categorizados de acuerdo a un modelo de calidad específico (Blas, Gonnet y Leone 2014).

(PRODUCTO DE SOFTWARE INTERMEDIO) *Modelo, especificación, documento o código fuente que es creado y utilizado durante el proceso de construcción del sistema de software (Albin 2003).*

Simulación en Ingeniería de Software

Simular es el proceso de diseñar un modelo de un sistema real y realizar experimentos sobre el mismo a fin de entender su comportamiento y/o evaluar estrategias que mejoren su operación (Law y Kelton 1991).

Esta técnica ha sido ampliamente utilizada en ramas como la química, física y eléctrica con el objetivo de estudiar el comportamiento de distintos componentes ante determinadas situaciones y/o de evaluar la capacidad de respuesta de un sistema específico en un momento dado. También se ha empleado con éxito en el dominio militar a fin de estudiar, por ejemplo, la respuesta obtenida como resultado de la aplicación de distintas estrategias de ataque en situaciones de combate específicas. Sin embargo, en el campo de la ingeniería de software la aplicación de este tipo de técnicas se ha visto afectada por las propiedades inherentes de los productos que se desarrollan.

Tal como se ha mencionado con anterioridad, recientemente la ingeniería de software ha comenzado a aplicar este tipo de técnicas sobre los productos intermedios

del proceso de desarrollo a fin de predecir la calidad del sistema resultante. Existen múltiples ventajas de emplear simulación en etapas tempranas de trabajo como, por ejemplo, la reducción del tiempo de desarrollo que se deriva de la posibilidad de verificar las decisiones de diseño en escenarios artificiales. Un mejor ajuste del diseño propuesto en relación a los requerimientos de calidad garantiza una baja presencia de errores y/o costos inesperados en etapas de trabajo futuras. Esto da lugar a sistemas de software robustos y con un alto grado de mantenibilidad. Aún más, los modelos de simulación formulados en la etapa de diseño pueden, luego, ser utilizados como complemento de los componentes de software finales para establecer el mejor curso de acción en tiempo de ejecución con el objetivo de lograr una correcta adaptabilidad. En este sentido, es evidente que el costo de construir un modelo de simulación y realizar corridas a fin de evaluar su comportamiento, es siempre menor a los costos asociados a la obtención de un producto defectuoso. Por este motivo, aunque el producto intermedio no refleje la totalidad del sistema de software, el uso de técnicas de simulación brinda una buena aproximación para analizar la calidad que, eventualmente, se obtendría del producto final.

Conclusiones

La calidad del software se ha convertido en un problema crítico en la ingeniería de software porque afecta los costos de desarrollo, el cumplimiento de tiempos de entrega y la satisfacción del usuario, entre otros. En este contexto, la evaluación de calidad en etapas tempranas de trabajo contribuye a evitar estos inconvenientes, reduciendo riesgos y ayudando al entendimiento del producto requerido.

En este capítulo se han descripto brevemente distintas perspectivas útiles para comprender, estudiar y analizar la calidad en relación a productos de software genéricos. Se han presentado las arquitecturas de software como vehículo para la estimación de la calidad esperada como resultado del proceso de construcción, resaltando la importancia de la evaluación de los diseños arquitectónicos como una forma de verificación. Luego, alcanzar un nivel de calidad apropiado en un producto de software específico implica lograr un diseño arquitectónico adecuado a la naturaleza de los atributos de calidad relevantes.

El siguiente capítulo centra la atención específicamente en el estudio de la calidad de servicios basados en entornos de computación en la nube, haciendo referencia a la aplicabilidad de las estrategias genéricas (descriptas en este capítulo) como base para el trabajo en dicho ámbito.

Capítulo 2. Estudio de la Calidad en Productos de Software como Servicios en la Nube

Los productos de software deben incorporar rápidamente los nuevos requerimientos relacionados con plataformas (es decir, diversidad de dispositivos) y modelos de ejecución (como ser aplicaciones con interacción vía web, arquitecturas basadas en servicios y computación en la nube.). La existencia de un contexto tan exigente y cambiante dentro de este ámbito, conlleva a que la calidad de los productos de software constituya una preocupación dentro de la industria. En el capítulo previo se han introducido nociones básicas referidas a distintas perspectivas de estudio de la calidad en relación al software. Este capítulo centra la atención en la calidad de servicios de software como productos desplegados en entornos de computación en la nube, detallando las diferencias que existen al compararlos con los productos de software tradicionales y describiendo el conjunto de dificultades y problemas que se derivan de la falta de madurez en el área.

2.1 Computación en la Nube

El modelo de *computación en la nube* (CC por sus siglas en inglés, *cloud computing*) refiere a un paradigma de computación distribuida que se centra en ofrecer a un amplio rango de usuarios acceso distribuido y escalable a tecnología de la información

haciendo uso de hardware virtualizado y/o infraestructura de software a través de Internet (Lewis 2010). De acuerdo con el Instituto Nacional de Normas y Tecnología (NIST por sus siglas en inglés, *National Institute of Standards and Technology*), el paradigma de CC es "un modelo que habilita acceso a Internet bajo demanda de forma conveniente y generalizada a fin de compartir un conjunto de recursos de computación configurables que pueden ser asignados y entregados de forma rápida con un esfuerzo marginal reducido o con mínima interacción con el proveedor del servicio" (Mell y Grance 2011).

En este contexto, el concepto base detrás del modelo de CC es bajar el cómputo de los usuarios a un conjunto de proveedores de recursos remotos. Luego, uno de los principales objetivos que persigue este nuevo paradigma es el aumento en la capacidad y productividad en tiempo de ejecución sin invertir en nueva infraestructura, nuevas licencias de software ni en capacitación para empleados (Khan y colab. 2013). En virtud de lograr este objetivo, el modelo de CC propone entregar distintos tipos de tecnología de información bajo el formato de servicio. Es decir, el poder de cómputo, el software, los servicios de almacenamiento y las plataformas de desarrollo se entregan a los usuarios según la demanda de consumidores externos conectados vía Internet (Foster y colab. 2008). Luego, los usuarios acceden al servicio requerido bajo demanda sin necesidad de alterar sus propios modelos de tecnología de la información.

Entre las principales propiedades de este modelo computacional, se destacan las siguientes:

- *Flexibilidad*: Los usuarios pueden rápidamente obtener recursos computacionales, a medida que lo necesiten, sin requerir de interacción humana. Estos recursos (tanto hardware como software) son asignados de acuerdo a las necesidades planteadas por los usuarios, pudiendo escalar automáticamente (tanto hacia arriba como hacia abajo).

- *Escalabilidad*: Una arquitectura de CC puede escalar tanto horizontal como verticalmente, de acuerdo a la demanda. Es decir, en un momento dado pueden agregarse nuevos nodos y/o removerse nodos existentes sin perder poder computacional en la red.
- *Acceso por medio de la red*: Las capacidades computacionales quedan disponibles a los usuarios por medio de mecanismos estándar implementados sobre la red, lo que promueve el uso de plataformas heterogéneas.
- *Independencia de la localización*: El cliente no tiene control ni conocimiento acerca de la localización exacta de los recursos que se asignan en respuesta a sus solicitudes. Sin embargo, en algunos casos, el usuario puede llegar a especificar una localización con un alto nivel de abstracción (por ejemplo, una zona geográfica específica).
- *Confiabilidad*: La capacidad de utilizar múltiples sitios de forma redundante, hace que los entornos de CC sean altamente convenientes para la continuidad y recuperación de los sistemas y aplicaciones de software ante la ocurrencia de desastres naturales.

(COMPUTACIÓN EN LA NUBE - CLOUD COMPUTING -) *Modelo computacional que habilita el acceso a Internet bajo demanda de forma conveniente y generalizada a fin de compartir un conjunto de recursos de computación configurables que pueden ser asignados y entregados de forma rápida con un esfuerzo marginal reducido y/o con interacción mínima con el proveedor del servicio subyacente (Mell y Grance 2011).*

2.1.1 Tipos de Nubes

Existen dos criterios de clasificación que usualmente se utilizan en la literatura para caracterizar los tipos de tecnologías de CC, a saber: *según el tipo de prestación* y *según la forma de acceso* (Reese 2009; Lewis 2010; Dillon, Wu y Chang 2010; Armbrust y colab. 2010; Zhang, Cheng y Boutaba 2010).

Los entornos de CC según su prestación se caracterizan en función del tipo de servicio que provee la nube a los usuarios. En este caso, se identifican tres tipos de nubes, a saber: software como servicio, infraestructura como servicio y plataforma como servicio. En contraposición, de acuerdo al conjunto de usuarios que pueden acceder a los recursos de la nube (es decir, la forma de acceso), los entornos de CC se dividen en nubes públicas y nubes privadas.

La Tabla 2.1 presenta los criterios de clasificación propuestos junto con los tipos de nube identificados en cada caso. Además, para cada tipo de nube, detalla un ejemplo de utilidad para comprender su definición.

COMPUTACIÓN EN LA NUBE	*Según su Prestación*	SOFTWARE COMO SERVICIO	***Ejemplo***: **Gmail -** Aplicativo de software para correo electrónico que se utiliza sobre un cliente delgado (navegador web o aplicativo móvil) que provee la misma funcionalidad que otros servidores de correo sin requerir su administración.
		PLATAFORMA COMO SERVICIO	***Ejemplo*: AWS Elastic Beanstalk -** Servicio de Amazon Web Services (AWS) que permite crear aplicaciones web (desarrolladas con Java, .NET, PHP, Node.js, Python, Ruby, Go y Docker) y desplegarlas sobre un conjunto definido de servicios de AWS.
		INFRAESTRUCTURA COMO SERVICIO	***Ejemplo*: Amazon EC2 (Amazon Elastic Compute Cloud) -** Servicio de infraestructura web (específicamente de poder de cómputo) que proporciona capacidad informática en la nube segura y de tamaño modificable.
	Según su Acceso	PÚBLICA	***Ejemplo*: Microsoft Azure -** Plataforma general para aplicaciones que posee desde servicios que alojan aplicaciones en los centros de procesamiento de Microsoft hasta servicios de comunicación segura y federación entre aplicaciones.
		PRIVADA	***Ejemplo*: Hewlett Packard Enterprise, IBM Mirantis, Rackspace -** Proveedores que brindan un conjunto combinado de hardware y software colocados físicamente en las instalaciones del cliente pero que son gestionados por el proveedor.

Tabla 2.1. Criterios para la clasificación de entornos de computación en la nube.

Tipos de Nubes según su Prestación

Software como Servicio (Software-as-a-Service, SaaS)

Es un modelo de despliegue de software basado en la web que posibilita que un aplicativo de software esté completamente disponible a los usuarios por medio del uso de una interfaz de cliente delgado (como ser, un navegador web) accesible desde diferentes dispositivos. Básicamente es un término que refiere a la instalación de aplicaciones de software en la nube, ya que provee a los usuarios de un conjunto de capacidades específicas que les permiten utilizar aplicaciones del proveedor de software que son ejecutadas sobre un ambiente remoto.

Aunque no todos los sistemas SaaS son sistemas de CC, la mayoría lo son. En estos casos, las organizaciones y desarrolladores utilizan capacidades específicas de negocio desarrolladas por terceros (las cuales se encuentran en la nube) para dar soporte a aplicaciones de software propias.

A un usuario de este tipo de servicios no le interesa conocer los recursos de hardware en los cuales se encuentra alojado el aplicativo de software, el tipo de sistema operativo con el cual interactúa o el lenguaje en el cual ha sido desarrollado. Ni siquiera debe preocuparse por su instalación. Los consumidores SaaS no administran ni controlan la infraestructura subyacente a la aplicación (es decir, las redes, servidores, sistemas operativos y bases de datos de la aplicación). Sin embargo, en aquellos casos en los que el aplicativo de software se encuentre explícitamente diseñado para permitir configuraciones externas, el usuario podrá tener un conjunto limitado de privilegios referidos a decisiones de infraestructura y/o funcionalidades específicas. Un control total de los usuarios sobre la infraestructura y/o el funcionamiento del software en el contexto de SaaS es imposible de lograr.

Entre las principales características que posee el modelo SaaS en relación a los entornos de CC se destacan: disponibilidad vía navegador web, reserva bajo demanda, condiciones de pago basadas en uso, mínimas demandas de tecnología de la

información como soporte a la ejecución en el dispositivo cliente y despliegue multitenancy[1]. En este último caso, una aplicación multitenancy refiere a un software que soporta el despliegue de múltiples clientes sobre una única instancia de software. Esta propiedad permite que los proveedores de SaaS puedan alojar más clientes de software con menos componentes de hardware, brindar mayor rapidez en la instalación de actualizaciones y parches de seguridad y proveer una arquitectura compacta.

Por otro lado, entre los beneficios que obtienen los consumidores SaaS (usuarios) se destacan: inexistencia de costo inicial de adquisición de la aplicación de software, independencia del mantenimiento y de la aplicación de actualizaciones de software y/o hardware, accesibilidad por medio de Internet, alta disponibilidad del servicio y pago en función del consumo (uso) que se realice.

Plataforma como Servicio (Platform-as-a-Service, PaaS)

Las nubes PaaS proveen a los usuarios la capacidad de desplegar aplicaciones propias (creadas por ellos mismos) y/o adquiridas sobre la infraestructura de CC disponible. Tales aplicaciones deben haber sido implementadas haciendo uso de los lenguajes de programación y de las herramientas de desarrollo para las cuales el proveedor del entorno de CC tiene soporte.

En este caso, el usuario no administra ni controla la infraestructura de CC subyacente (incluyendo componentes de red, servidores, sistemas operativos y almacenamiento de la información). Sin embargo, si tiene control sobre la forma en la cual las aplicaciones son desplegadas y, posiblemente, sobre las configuraciones del entorno que aloja sus aplicativos.

[1] Refiere a un principio de arquitectura de software en el cual una única instancia de software se ejecuta sobre un servidor, dando lugar a que un grupo de usuarios comparta una misma vista del software que utilizan de forma remota.

En este sentido, las nubes PaaS permiten a los usuarios aprovechar recursos de organizaciones bien establecidas para crear y almacenar aplicaciones de mayor escala que las que una pequeña empresa sería capaz de manejar.

Infraestructura como Servicio (Infrastructure-as-a-Service, IaaS)

Es un modelo que provee a sus consumidores distintos tipos de recursos computacionales (capacidad de procesamiento, almacenamiento de información y redes de comunicación, entre otros) a modo de servicios. De esta forma, las nubes IaaS permiten que sus usuarios desarrollen y ejecuten software (tanto aplicaciones como sistemas operativos) sobre los recursos disponibles. Es decir, brindan una infraestructura computacional disponible sobre Internet para la ejecución de aplicaciones de software.

A diferencia de las nubes SaaS, en el modelo IaaS el cliente tiene un control total tanto del sistema operativo como así también del almacenamiento de los datos y del despliegue de las aplicaciones de software. Sin embargo, es posible que la nube restringa el control en relación a la selección de los componentes de red sobre los que se ejecutarán sus requerimientos. De esta manera este tipo de nube permite, tanto a los desarrolladores como así también a las organizaciones, extender su infraestructura de tecnología de la información en función a su demanda.

Tipos de Nubes según su Acceso

Nubes Públicas

La infraestructura de la nube se encuentra adaptada para su uso por parte del público en general o para un sector específico de una industria particular, perteneciendo sus recursos a una organización concreta que comercializa entornos de CC. Los proveedores de este tipo de entornos administran tanto la infraestructura subyacente como el conjunto de recursos disponibles de acuerdo a las capacidades requeridas por los usuarios. Tales capacidades quedan indicadas al momento de la

contratación del servicio, por lo que generalmente existen múltiples planes de contratación.

Entre las ventajas de las nubes públicas se destacan:

- *Bajos costos*: No es necesario adquirir hardware ni software. Solo se paga por el servicio que se usa.
- *No requieren mantenimiento*: El proveedor de los servicios de la nube se encarga de realizar el mantenimiento (no el usuario).
- *Escalabilidad casi ilimitada*: El usuario dispone de recursos computacionales conforme su necesidad.

De esta manera, en las nubes públicas los recursos computacionales se ofrecen a los usuarios bajo la forma de servicios, usualmente por medio de una conexión a Internet, a cambio de una tarifa a pagar de acuerdo al consumo realizado. En este caso, los usuarios pueden escalar su uso según la demanda, por lo que no requieren de una inversión en nuevo hardware para mejorar su rendimiento.

Nubes Privadas

Una nube privada queda compuesta por un conjunto de recursos informáticos que son utilizados exclusivamente por una empresa u organización. Tales entornos de CC pueden encontrarse ubicados físicamente en la organización o en un proveedor de servicios externo, pero sus servicios e infraestructura se mantienen en una red privada y, tanto el hardware como el software se dedican exclusivamente a la organización de interés. De esta forma, una organización puede personalizar sus recursos computacionales de forma sencilla a fin de cumplir requisitos específicos de tecnología de la información, manejando la nube y controlando el acceso a tales recursos.

Entre las ventajas de las nubes privadas se destacan:

- *Mayor flexibilidad*: La organización puede personalizar el entorno de la nube para satisfacer necesidades empresariales específicas.

- *Mejor seguridad*: Los recursos no se comparten con otros usuarios, por lo que es posible contar con mayores niveles de control y seguridad.

De esta manera, la infraestructura de la nube es operada por una organización privada, pudiendo ser administrada por esta organización o por terceros y estando ubicada dentro o fuera de sus instalaciones. En general, este tipo de nubes son utilizadas en agencias gubernamentales e instituciones financieras, pudiendo también ser usadas en cualquier organización mediana o grande que realice operaciones esenciales para la empresa y busque aumentar el control sobre su entorno de tecnología de la información.

Otros Tipos de Nubes según su Acceso

Existen dos tipos de nubes que usualmente son identificadas como casos especiales de nubes privadas y públicas, a saber: nubes de comunidad (community clouds) y nubes híbridas (*hybrid clouds*).

Las nubes de comunidad son nubes compartidas por múltiples organizaciones, las cuales se considera que conforman una comunidad. En este sentido, la nube brinda soporte específico a las necesidades y preocupaciones de dicha comunidad, pudiendo ser administrada por estas organizaciones o por terceros. Estos entornos de CC no deben estar necesariamente ubicados dentro de las instalaciones de la comunidad de referencia.

Por su parte, las nubes híbridas refieren a la combinación de dos o más nubes (públicas, privadas y/o de comunidad) las cuales han sido trabajadas como entidades individuales pero, potencialmente, pueden ser combinadas en un entorno de mayor envergadura por medio de tecnología estándar o propietaria, habilitando la portabilidad de datos y aplicaciones.

2.1.2 Adopción de Entornos de Computación en la Nube

Existen múltiples beneficios en la adopción de entornos de CC. Entre las principales motivaciones por las cuales un usuario u organización adoptaría un enfoque de CC se encuentran (Lewis 2010; Gupta, Seetharaman y Raj 2013):

- *Disponibilidad*: Los usuarios tienen la posibilidad de acceder a los recursos en cualquier momento por medio de una conexión a Internet.
- *Colaboración*: Los usuarios ven a la nube como una forma de trabajar simultáneamente con la misma información.
- *Elasticidad*: El proveedor administra dinámicamente la asignación y/o sustracción de los recursos a los usuarios según sus necesidades. Esta tarea se lleva a cabo de forma transparente al usuario.
- *Bajos costos de infraestructura*: El modelo de "pago por uso" permite a las organizaciones pagar únicamente por los recursos que necesitan, invirtiendo muy poco dinero en comparación con el requerido para instalar una infraestructura similar en sus instalaciones. Además, no deben afrontar costos de mantenimiento y/o actualización de tal infraestructura.
- *Movilidad*: Los usuarios tienen la posibilidad de acceder tanto a las aplicaciones como a los datos desde cualquier ubicación.
- *Reducción de riesgos*: Las organizaciones pueden utilizar la nube para probar ideas y conceptos antes de realizar inversiones en tecnología.
- *Escalabilidad*: Los usuarios tienen acceso a una gran cantidad de recursos escalados dinámicamente según sus demandas.
- *Virtualización*: Cada usuario tiene una vista única de los recursos disponibles, independientemente de la forma en la cual tales recursos estén ubicados sobre los dispositivos físicos. Por este motivo, un mismo proveedor de CC puede administrar un gran número de usuarios en base a un conjunto reducido de recursos físicos.

Sin embargo, a pesar de la gran cantidad de beneficios enumerados, también existe un conjunto de barreras que un usuario u organización enfrenta al intentar adoptar un enfoque de CC (Lewis 2010; Low, Chen y Wu 2011). En este sentido, se destacan:

- *Interoperabilidad*: Dado que todavía no se ha definido un conjunto universal de estándares y/o interfaces entre nubes, el usuario corre un alto riesgo asociado al poder del proveedor.
- *Latencia*: Todo el acceso hacia una nube es realizado vía Internet, introduciendo latencia en cada comunicación entre el usuario y el proveedor.
- *Restricciones de lenguaje y/o plataforma*: No todos los proveedores de nubes soportan las mismas tecnologías.
- *Confiabilidad*: Los productos de hardware que forman las infraestructuras subyacentes de los entornos de CC pueden fallar de forma inesperada.
- *Regulaciones*: Existen aspectos de los entornos de CC que aún no se encuentran regulados, como ser: jurisdicción, protección de datos, prácticas de manejo y manipulación de la información y protocolos de transferencia de datos, entre otros.
- *Control de los recursos*: El nivel de control que el usuario tiene sobre el proveedor de la nube y sus recursos varía de acuerdo al proveedor y al tipo de servicio contratado.
- *Seguridad*: En este caso, una de las principales preocupaciones es la privacidad de los datos. En los entornos de CC, los usuarios no tienen control ni conocimiento acerca del lugar en el cual se encuentra almacenada la información.

En este contexto, los usuarios u organizaciones que estén considerando la adopción de entornos de CC para sus negocios, deben valorar tanto los beneficios como las barreras existentes a fin de llevar a cabo una implementación apropiada de este tipo de soluciones.

2.1.3 Relación con la Tecnología de Virtualización

El surgimiento de los entornos de CC fue potenciado por la tecnología de virtualización (Zissis y Lekkas 2012). Esta tecnología nace en el año 1967 pero, durante décadas, sólo estuvo disponible en sistemas mainframe.

En esencia, la virtualización abstrae los detalles del hardware y provee recursos ficticios a aplicaciones de alto nivel (Zhang, Cheng y Boutaba 2010). Es decir, básicamente puede ser interpretada como una tecnología base en la cual una computadora (denominada *host*) ejecuta una aplicación especial (conocida como *hipervisor*) la cual se encarga de crear máquinas virtuales que simulan ser computadoras físicas. Sobre estas simulaciones se puede ejecutar cualquier tipo de software, desde sistemas operativos hasta aplicaciones de usuario (Naone 2009).

Los entornos de CC amplían la definición de virtualización ya que utilizan esta tecnología como mecanismo de soporte para agrupar recursos informáticos de grupos de servidores y asignar o reasignar dinámicamente recursos virtuales a aplicaciones bajo demanda (Zhang, Cheng y Boutaba 2010). Luego, la virtualización es un elemento crítico de las implementaciones de CC que es usado para proveer las características esenciales de este tipo de computación, como ser la independencia de localización, la búsqueda y sondeo de recursos y la elasticidad de componentes de software.

De acuerdo con (Lynch 2008), lo que diferencia a los entornos de CC de la computación distribuida es precisamente la virtualización. El CC, a diferencia de los entornos de computación distribuida, eleva la virtualización a su máximo poder de cómputo logrando separar los aspectos lógicos de los físicos. Mientras que en la computación distribuida se alcanza una alta tasa de utilización por medio de la asignación de múltiples servidores a una única tarea o trabajo, la virtualización propuesta en el CC logra una alta tasa de utilización por medio del cómputo de múltiples tareas de forma concurrente en un único servidor.

Aunque la mayoría de los autores reconocen similitudes entre el paradigma de CC y la computación distribuida, las opiniones parecen agruparse alrededor del hecho de que CC envuelve a la computación distribuida y que la computación distribuida es la base del CC. Luego, su principal diferencia es la forma en la cual tratan la tecnología de virtualización.

2.2 Software como Servicio en la Nube (Software-as-a-Service)

No hay nada nuevo en las tecnologías que forman parte del CC (Cooke 2010). De hecho, han existido desde hace décadas. Sin embargo, el modelo bajo el cual se entregan los recursos constituye una nueva forma de ver los entornos de computación.

Tal como se ha definido con anterioridad, CC puede referir tanto al software como a la infraestructura, siendo comúnmente una aplicación que se accede a través de la web o un servidor que se carga cuando se lo necesita.

De acuerdo con (Reese 2009), una prueba muy sencilla que sirve para determinar si un servicio se encuentra en la nube (o no), es la siguiente: “si se puede ir hasta una librería o café que tenga acceso a Internet, sentarse en una computadora sin preferencia por el sistema operativo de la misma y/o su navegador y aun así acceder al servicio, entonces el servicio se encuentra en la nube”. Claramente, esta es una primera aproximación para la determinación de la naturaleza de un servicio, pero en términos generales existen tres criterios útiles para definir a un servicio como servicio de CC:

- Se accede por medio de un navegador web (no propietario) o por medio de una API.
- No se requiere de un capital inicial para comenzar a utilizarlo.
- Sólo se paga por el uso que se ha realizado.

La utilización del esquema SaaS como modelo de entrega de servicio se ha incrementado en los últimos años. No sólo se encuentra en ascenso la cantidad de

servicios prestados y el número de usuarios que hacen uso de los mismos, sino que también se evidencia un crecimiento en la cantidad de proveedores disponibles para este tipo de plataformas (Fan y colab. 2015). Este tipo de servicios posee múltiples beneficios para los usuarios, entre los que se incluyen (Fonseca 2008):

- *Disponibilidad y movilidad*: Los usuarios tienen la posibilidad de acceder a las funcionalidades del software en cualquier momento, desde cualquier ubicación, por medio de una conexión a Internet.
- *Bajos costos*: Los usuarios evitan el costo o inversión inicial del software, quedando además exentos de las tareas de mantenimiento y/o actualizaciones del mismo.
- *Reducción de riesgos*: Los usuarios pueden utilizar la nube para probar ideas y conceptos vinculados a nuevas funcionalidades con el objetivo de analizar soluciones en una etapa previa a la producción de una mayor inversión.
- *Pago de acuerdo al uso*: Los usuarios pagan una cuota cuyo valor queda determinado de acuerdo al uso que realizan en relación a las distintas funcionalidades del software.

Para poder garantizar los beneficios enunciados con anterioridad, es evidente que un SaaS debe mantener un nivel de calidad relativamente más alto que el de un sistema de software tradicional. Aún más, debido a que todo SaaS es ejecutado sobre una infraestructura de hardware propia del proveedor del servicio, el relevamiento de la calidad de este tipo de software no trata únicamente con aspectos de calidad de aplicación. En este contexto, una primera aproximación a la estructuración arquitectónica para SaaS se presenta en la Figura 2.1. De acuerdo con este diagrama, los *Usuarios* (*Users*) acceden a los SaaS alojados en la *Capa de Software* (*Software Layer*) por medio de un acceso indirecto a aplicaciones de software remotas. En este contexto, las funcionalidades se encuentran alojadas en la capa superior mientras que los recursos (sobre los cuales ejecutan dichas funcionalidades) quedan encapsulados en la *Capa de Infraestructura* (*Infrastructure Layer*).

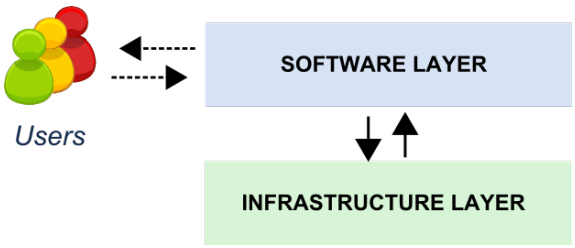


Figura 2.1. Estructuración arquitectónica de Software-as-a-Service.

La mayoría de las arquitecturas de software propuestas en la literatura para analizar especificaciones de CC se basan en modelos de capas (Hu y colab. 2011; Khan y colab. 2013; Fehling y colab. 2014; Blas, Gonnet y Leone 2015). En el Capítulo 7 se analizan los distintos diseños arquitectónicos propuestos a fin de elaborar una representación arquitectónica integral. Sin embargo, independientemente de la cantidad de capas identificadas y su denominación, siempre es posible realizar una abstracción de los conceptos involucrados a fin de lograr un esquema similar al propuesto en la Figura 2.1.

Teniendo en cuenta que el consumidor del SaaS (usuario) experimenta de forma directa las características funcionales y no funcionales del mismo y, tomando en consideración el crecimiento de proveedores que se ha dado en los últimos tiempos, es evidente que el análisis y entendimiento de la calidad de este tipo de software por parte de los desarrolladores puede llevar tanto al éxito como al fracaso. En consecuencia, el estudio y relevamiento de los factores de calidad propios de este tipo de servicios (vistos como productos de software) requiere de una rigurosa evaluación.

2.3 Evaluación de la Calidad en Software-as-a-Service

2.3.1 Desafíos e Inconvenientes

Tal como se ha enunciado en el capítulo precedente, la intrínseca complejidad de los productos de software dificulta la determinación de la calidad asociada (Pressman

2010). Este panorama empeora al intentar aplicar los conceptos y técnicas existentes sobre plataformas más novedosas basadas en modelos de ejecución modernos; como ser, por ejemplo, el esquema de CC.

De acuerdo con (Silva y colab. 2015), el CC obliga a cambios en las normas y estándares, las capacidades de infraestructura (hardware) y la especificación de las arquitecturas de software. En el caso de SaaS, estos aspectos son fundamentales para evaluar el despliegue del servicio, ya que reducen la brecha entre la aplicación y la infraestructura (proporcionando los medios necesarios para que tanto los proveedores como los consumidores puedan evaluar la calidad de los servicios).

En los siguientes aparatos se describen brevemente los principales desafíos detectados al momento de evaluar la calidad en servicios de software basados en esquemas de CC. Estos problemas sientan las bases fundamentales sobre las cuales se desarrolla el trabajo de tesis descripto en los capítulos subsiguientes.

Nuevas Propiedades de Calidad

La determinación de cuál es el conjunto de elementos y propiedades que definen un nivel adecuado de calidad para un sistema de software específico es una pregunta altamente dependiente del contexto (Kitchenham y Pfleeger 1996).

En este escenario, el estudio, entendimiento y análisis de la calidad en SaaS se ve afectado por la naturaleza del entorno de CC. Esto conlleva a que:

- Se generen distintas inquietudes con respecto a la calidad del servicio resultante de acuerdo a los distintos grupos de interés involucrados en la estructuración del servicio (como ser, por ejemplo, los usuarios finales y los proveedores de infraestructura)
- Se visibilice el surgimiento de nuevas propiedades de calidad propias de este nuevo tipo de tecnologías (no contempladas en los enfoques tradicionales).

- Se requiera de nuevas técnicas de estudio de la calidad que se ajusten al dinamismo con el cual surgen (y por lo tanto, con el cual deben ser evaluados) estos nuevos servicios de software.

Luego, el entendimiento de los aspectos de calidad que impactan en este nuevo tipo de productos y la forma en la cual tales aspectos deben ser interpretados, representa un desafío a ser analizado a fin de dar respuesta a preguntas como: ¿Cuáles son las principales propiedades de calidad que influyen en el desempeño de los productos SaaS? ¿Cómo se vinculan estas nuevas propiedades con las definidas para productos de software tradicionales? ¿Cómo deben medirse? ¿Qué indicadores deben tomarse como referencia?

En este contexto es probable que un subconjunto de las propiedades de calidad definidas para productos de software tradicionales continúe siendo válido para el caso de SaaS (ya que, en los aspectos básicos, ambos productos corresponden a desarrollos de software). Luego, la ampliación de los modelos de calidad existentes (por medio de la incorporación de propiedades complementarias) constituye un área de trabajo susceptible de ser abordada a fin de lograr una solución adecuada a las necesidades de evaluación de la calidad en este nuevo tipo de productos.

Representación del Diseño Arquitectónico

El diseño arquitectónico es un proceso creativo en el que se intenta establecer una organización de los componentes del sistema de software que satisfaga tanto los requerimientos funcionales como los no funcionales (Sommerville 2005). Por este motivo, la tarea de diseñar arquitecturas de software se caracteriza por su complejidad.

Varios autores plantean que las decisiones arquitectónicas son el núcleo de las arquitecturas de software (Kruchten 2004; Jansen y Bosch 2005; Tyree y Akerman 2005; Boer y colab. 2009). Una *decisión arquitectónica* es una descripción del conjunto de agregados, eliminaciones y modificaciones realizadas sobre una arquitectura de

software, el motivo de que estos cambios tengan lugar y las reglas de diseño junto con las respectivas restricciones y especificaciones adicionales que (posiblemente de forma parcial) logran cubrir uno o más de los requerimientos de la arquitectura (Jansen y Bosch 2005).

En términos generales, una decisión arquitectónica es la salida que se genera en el proceso de diseño durante la construcción inicial o la evolución de un sistema de software. Las decisiones de diseño refieren al dominio de aplicación del sistema, los patrones arquitectónicos utilizados, los componentes definidos, la infraestructura seleccionada y todos los aspectos adicionales necesarios para satisfacer los requerimientos relevados. En este contexto, las decisiones estructurales comprenden la determinación del conjunto de patrones de diseño aplicables al sistema de software y la definición de los principales componentes arquitectónicos.

Aunque en la actualidad existe una amplia variedad de patrones arquitectónicos útiles para el diseño de productos de software tradicionales (Garlan y Shaw 1994; Monroe y colab. 1997; Myllymäki, Koskimies y Mikkonen 2002; Pahl, Giesecke y Hasselbring 2009), al intentar trasladar estos patrones a productos SaaS se presentan dos problemas derivados de la falta de madurez del área (Blas, Gonnet y Leone 2015):

- *Los patrones orientados a entornos de CC aún no están totalmente establecidos*: Dada la variedad de enfoques que pueden utilizarse en esta rama, la mayoría de los patrones existentes apuntan a la resolución de problemas asociados a la infraestructura (subyacente al diseño funcional de la aplicación).
- *No existe una técnica de representación que facilite la comprensión de las arquitecturas de software basadas en CC*: Frecuentemente las estrategias de representación arquitectónica mezclan elementos alojados en la infraestructura con componentes de software de aplicación, generando confusión y ambigüedad al lector.

A estas dificultades se le suma la necesidad de evaluar los diseños arquitectónicos resultantes a fin de establecer su nivel de adecuación de acuerdo a la calidad requerida (tal como se ha detallado en el apartado *"Calidad de Producto y Arquitectura de Software"*).

Luego, en relación a la representación de arquitecturas SaaS surgen múltiples interrogantes, entre los cuales se destacan: ¿Cuáles son los patrones de diseño básicos aplicables a servicios basados en CC? ¿Cómo impacta el diseño arquitectónico de la capa de infraestructura subyacente en la arquitectura de la capa de software? ¿Existe alguna técnica de representación que permita independizar el diseño de la capa de software de la organización de la capa de infraestructura? ¿Qué metodologías disponen los arquitectos para evaluar sus diseños? ¿Existen herramientas de software que los ayuden a automatizar el proceso de evaluación?

Con el objetivo de brindar a los arquitectos de software un mecanismo de soporte adecuado que los ayude a aumentar su productividad en el contexto de servicios basados en CC, se tiene que: *i)* la identificación de patrones de diseño útiles para la definición de arquitecturas SaaS, junto con *ii)* la implementación de herramientas de software que faciliten su aplicación; constituye un área de trabajo relevante.

(DECISIÓN ARQUITECTÓNICA) *Descripción del conjunto de agregados, eliminaciones y modificaciones realizadas sobre una arquitectura de software, el motivo de que estos cambios tengan lugar y las reglas de diseño junto con las respectivas restricciones y especificaciones adicionales que (posiblemente de forma parcial) logran cubrir uno o más de los requerimientos de la arquitectura (Jansen y Bosch 2005).*

Dependencia de la Infraestructura Subyacente

Tomando como referencia el esquema presentado en la Figura 2.1, es evidente que la calidad percibida por el usuario final no sólo depende de la calidad de la capa de software sino también de la calidad de la capa de infraestructura que le brinda soporte.

Luego, la calidad final del producto SaaS se ve condicionada por el contexto de ejecución definido para su infraestructura. En la mayoría de los casos, este contexto es contratado a un proveedor externo.

La dependencia de la infraestructura imposibilita lograr una visión objetiva del comportamiento del servicio ya que su calidad siempre se verá afectada por las características propias de la capa subyacente. Desafortunadamente, la tarea de determinar si las suposiciones hechas sobre el diseño arquitectónico de una infraestructura son correctas resulta a menudo complicada y tediosa. Frecuentemente el arquitecto carece de información cualitativa y/o cuantitativa de utilidad para el soporte de la evaluación y selección de alternativas de diseño parciales (Kazman, Klein y Clements 2000; Tang y Han 2005; Kazman, Gagliardi y Wood 2012). Esta realidad, sumada a la desconocimiento propio de la "caja negra" de la infraestructura, dificulta el estudio de la calidad del producto.

Aunque usualmente los proveedores de infraestructura brindan estadísticas referidas a un subconjunto de propiedades de calidad vinculadas al tipo de servicio que entregan, no siempre esta información está disponible y/o es de utilidad para evaluar el impacto de la misma en relación a los aspectos de calidad requeridos en el SaaS. En este contexto surgen múltiples inquietudes, como ser: ¿Es posible aislar la calidad de la infraestructura a fin de independizar su estudio en lo referido a la calidad de los servicios? ¿Cuáles son los aspectos de calidad fundamentales que deben ser relevados para estudiar la infraestructura? ¿Estos aspectos se condicen con los requeridos a nivel de software? ¿En qué grado impactan sobre el servicio dependiente? ¿Cómo deben ser relevados?

Luego, el diseño de estrategias de evaluación que independicen el estudio de la calidad de las capas de software e infraestructura pero que, al mismo tiempo, posibiliten a futuro su integración en base a indicadores de calidad adecuados, se convierte en un área de estudio y trabajo fundamental para los entornos de CC.

2.3.2 Arquitectura como Modelo para la Estimación de la Calidad

Tal como se ha explicado en la sección *"Calidad de Producto y Arquitectura de Software"*, el diseño arquitectónico de un producto de software puede ser utilizado como modelo base para estimar la calidad final con el objetivo de comparar el nivel de calidad logrado con respecto al nivel de calidad deseado.

Sin embargo, los problemas detallados en el apartado previo visibilizan la dificultad que existe cuando se intenta evaluar la calidad en entornos de CC (específicamente para el caso de SaaS). Este escenario no sólo se ve complejizado por las exigencias actuales de desarrollo de las aplicaciones web y en la nube (las cuales incorporan nuevas características funcionales que afectan a múltiples atributos de calidad como ser, por ejemplo, la seguridad y la disponibilidad), sino que también se dificulta a causa de la falta de mecanismos de representación apropiados para la construcción de diseños arquitectónicos que faciliten la aplicación de las estrategias de evaluación existentes (Mather, Kumaraswamy y Latif 2009; Al-Azzani y Bahsoon 2012; Gonzalez y colab. 2012; Zalewski y Kijas 2013). Además, la imposibilidad de independizar el diseño de software de la infraestructura obstaculiza el trabajo de los arquitectos ya que no disponen de los elementos necesarios para evaluar el diseño arquitectónico de forma aislada de la infraestructura subyacente. Este último punto se convierte en un arma de doble filo ya que el desempeño de una arquitectura de software puede evidenciar pequeñas variaciones cuando se la sitúa sobre distintos proveedores de entornos de CC. Por este motivo, aunque los diseños arquitectónicos de SaaS deben plantearse de forma aislada a la infraestructura a fin de garantizar el cumplimiento de sus objetivos (funcionales y no funcionales), los arquitectos de software deben tomar decisiones relativas a la infraestructura subyacente en virtud de contribuir a los intereses de calidad del servicio. Esta tarea añade tiempo de trabajo que ralentiza el desarrollo del producto.

Es evidente que, en este contexto, resulta necesario brindar toda la ayuda posible a los arquitectos de servicios basados en CC a fin de aumentar su productividad, contribuyendo a la evaluación de sus diseños en una etapa previa a la implementación. Este tipo de soporte debería auxiliarlos en el análisis de los atributos de calidad asociados a la arquitectura de software en etapas tempranas de desarrollo, permitiéndoles obtener información que sirva tanto para la toma de decisiones como para prevenir defectos en el servicio resultante. En relación a este último punto, se podrían reducir costos derivados de la presencia de errores detectados en etapas tardías del desarrollo (contribuyendo al ajuste del producto con respecto a las exigencias actuales del mercado).

Conclusiones

El paradigma de computación en la nube se ha convertido rápidamente en una de las estrategias de solución tecnológica más populares e influyentes del mundo actual (Hu y colab. 2011). De acuerdo con este esquema, los usuarios tienen la posibilidad de acceder a recursos bajo demanda por medio de Internet, dando lugar a la configuración de distintos tipos de servicios.

En este capítulo se ha presentado el entorno de computación en la nube, poniendo el foco en las aplicaciones de software que se despliegan como servicios sobre este modelo computacional. Específicamente, se abstrae el modelo en dos grandes componentes (software e infraestructura) a fin de presentar las relaciones que existen entre ellos. Para el caso de los servicios de software, se han detallado los problemas detectados en relación a la evaluación de la calidad; destacándose las dificultades evidenciadas en el uso de arquitecturas de software como modelos base para este análisis.

El siguiente capítulo aborda esta problemática, presentando la estructura general de la solución propuesta en esta tesis con el objetivo de evaluar las arquitecturas de servicios de software basados en la nube por medio del uso de modelos de simulación de eventos discretos.

Capítulo 3. Modelo para la Estimación de la Calidad utilizando Arquitecturas Web

El proceso de diseño de software requiere de un conjunto de herramientas computacionales que brinden soporte al análisis de la calidad de un producto de software en etapas tempranas de desarrollo. Teniendo en cuenta lo enunciado en el capítulo previo, en este capítulo se presenta el esquema de trabajo llevado a cabo en esta tesis a fin de dar respuesta al problema de la evaluación de la calidad en servicios de software y aplicaciones web basadas en entornos de computación en la nube.

3.1 Objetivo

3.1.1 Objetivo General

A pesar de los recientes avances, las complejidades cognitivas y técnicas del proceso de diseño de arquitecturas en los entornos de CC son evidentes. Tal proceso aún no se ha comprendido en forma acabada y, junto con el estudio de las propiedades de calidad, existe una gran necesidad de herramientas que soporten la evaluación de las alternativas de diseño obtenidas de manera eficiente y flexible.

En este contexto, el objetivo general del trabajo de investigación desarrollado en esta tesis consiste en *la formulación de un modelo de evaluación integral basado en simulación junto con el desarrollo de herramientas computacionales que permitan estimar*

un conjunto de atributos de calidad previamente definidos sobre la arquitectura de software de una aplicación web basada en un entorno de CC.

3.1.2 Objetivos Específicos

En relación al objetivo general, se consignan los siguientes objetivos específicos para la investigación:

- Especificar un modelo conceptual que facilite la comprensión de las propiedades de calidad y sus relaciones sobre productos de software genéricos el cual, luego, pueda ser extendido para el ámbito de CC.
- Establecer el conjunto de atributos de calidad medibles en tiempo de ejecución y la información mínima requerida para su evaluación a partir del diseño arquitectónico a fin de especificar las salidas requeridas como resultado de los modelos de simulación.
- Formular un modelo que permita capturar la información relacionada con la capa de software de una arquitectura de CC a fin de facilitar la representación de los diseños arquitectónicos de aplicaciones web en términos de patrones básicos.
- Especificar modelos de simulación que brinden soporte a los conceptos involucrados en la representación arquitectónica definida a fin de evaluar los aspectos de calidad seleccionados.
- Verificar los modelos de simulación tomando como referencia una comparativa de desempeño entre los resultados obtenidos en los modelos de simulación y el comportamiento esperado de los patrones de diseño.

3.2 Estructura General del Modelo

La Figura 3.1 esquematiza la estructura del enfoque integral desarrollado para estimar un conjunto de atributos de calidad predefinidos sobre un producto de

software web por medio de la combinación de técnicas de simulación de eventos discretos aplicadas sobre el diseño arquitectónico. El principal objetivo es *transformar el diseño de una arquitectura web en un modelo de simulación ejecutable que permita obtener mediciones de calidad basadas en métricas predefinidas, las cuales han sido especificadas de forma transversal al proceso de desarrollo*.

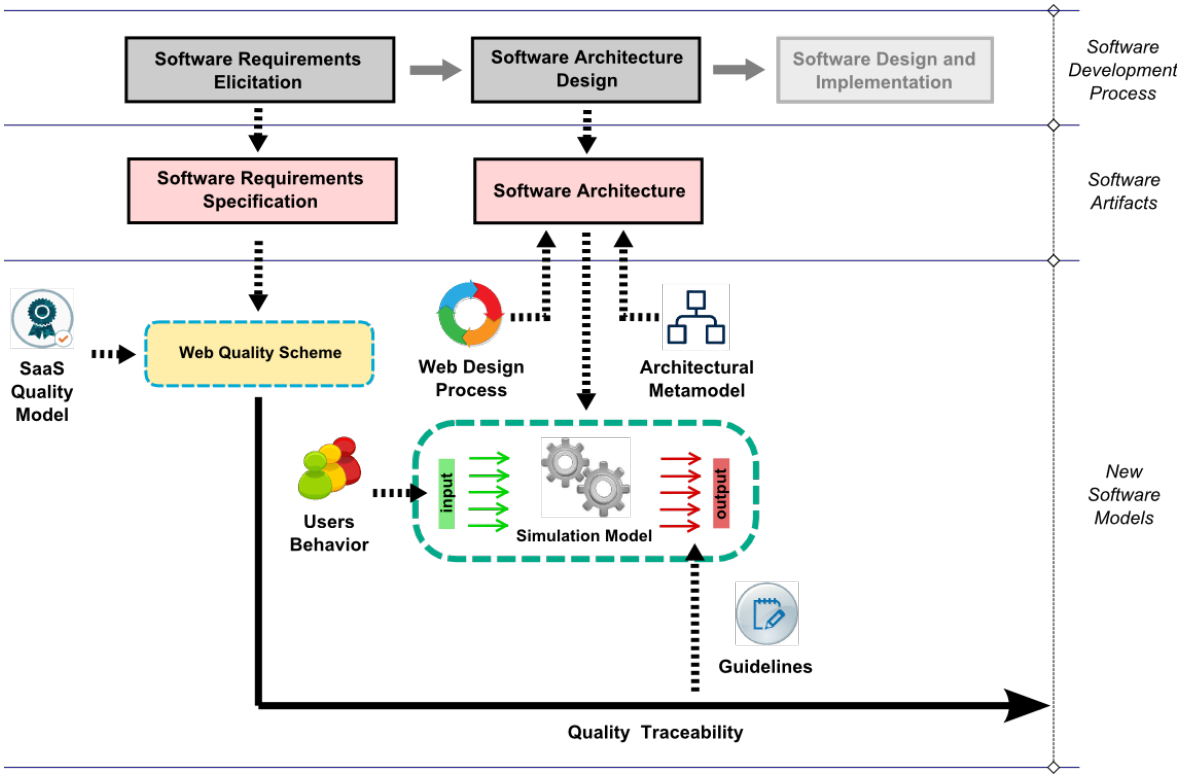


Figura 3.1. Esquema general de la propuesta.

En este contexto, el esquema propuesto vincula tres niveles de trabajo. Tales niveles han sido definidos como *Proceso de Desarrollo de Software* (*Software Development Process*), *Artefactos de Software* (*Software Artifacts*) y *Nuevos Modelos de Software* (*New Software Models*).

A fin de brindar un mecanismo de soporte para el análisis temprano de la calidad y la toma de decisiones referidas a la estructuración del producto bajo desarrollo en etapas temprana de trabajo, en el nivel superior (*Software Development Process*) se

diagraman (de forma secuencial y con un alto nivel de abstracción) las primeras etapas de un proceso de desarrollo de software genérico[1], a saber: *Elicitación de Requerimientos de Software* (*Software Requirements Elicitation*), *Diseño de la Arquitectura de Software* (*Software Architecture Design*) y *Diseño e Implementación del Software* (*Software Design and Implementation*).

En base a este proceso, sobre el nivel intermedio (*Software Artefacts*) se identifican dos de los principales artefactos creados como resultado de la ejecución de las etapas iniciales: *Especificación de Requerimientos de Software* (*Software Requirements Specification*) y *Arquitectura de Software* (*Software Architecture*). Tomando como referencia estos artefactos, el último nivel (*New Software Models*) actúa como mecanismo complementario a los enfoques de ingeniería de software tradicionales ya que incorpora un nuevo conjunto de modelos (derivados de los existentes) que facilita la evaluación de la calidad en CC. Aunque el esquema propuesto se encuentra planteado específicamente para el caso de aplicaciones web, puede ser replicado en otros contextos.

La *Especificación de Requerimientos de Software* es utilizada como elemento de entrada para la generación de un nuevo documento denominado *Esquema de Calidad Web* (*Web Quality Scheme*). Este documento encapsula las relaciones existentes en el conjunto de atributos de calidad identificados como parte de los requerimientos. Para esto, utiliza como referencia un *Modelo de Calidad SaaS* (*SaaS Quality Model*), el cual identifica la jerarquía de conceptos de calidad relevantes en el dominio del producto de software bajo desarrollo (en este caso, el dominio de servicios de software). Además, incorpora la especificación del conjunto de métricas requeridas para su estudio. Luego la definición de un *Esquema de Calidad* sobre una *Especificación de Requerimientos* particular contribuye a identificar los aspectos de calidad requeridos y la forma en la cual tales aspectos deben ser relevados.

[1] Independiente de la metodología de desarrollo utilizada, estas etapas corresponden a actividades que, en un punto u otro, de forma explícita o implícita, deben llevarse a cabo.

Teniendo en cuenta que todos los involucrados en un proceso de ingeniería de software son responsables de la calidad (Pressman 2010) y que el cumplimiento de las propiedades de calidad debe ser considerado a través del diseño, implementación y desarrollo (Bass, Clements y Kazman 2012); la definición de un *Esquema de Calidad* sienta las bases necesarias para todo tipo de evaluación de calidad que deba realizarse a lo largo del proceso de desarrollo de la aplicación. Esto se debe a que como parte del documento se identifican características y métricas específicas sobre los distintos artefactos, lo que permite mantener la trazabilidad de las propiedades de calidad analizadas (*quality traceability*). En el apartado *"Esquema de Calidad (Quality Scheme)"* se presenta una breve definición de este modelo.

En base a la aplicación de un *conjunto de criterios* (*guidelines*) que permitan analizar la información de calidad requerida de la aplicación web a fin de estimar la adecuación de un producto intermedio en una etapa de desarrollo específica, es posible establecer el conjunto de datos mínimos a ser obtenidos como parte del proceso de evaluación. Luego, los elementos definidos como parte de un *Esquema de Calidad* pueden ser utilizados como base para construir el proceso de evaluación de arquitecturas web. En un contexto de simulación, tales elementos definen las *variables de salida* (*output values*). Estas variables corresponden al conjunto de valores requeridos como resultado de la ejecución del modelo (influenciando los objetivos que se persiguen durante su construcción).

La construcción del modelo de simulación que representa la arquitectura de software de una aplicación web no forma parte de las tareas a ser desarrolladas por el arquitecto. Esto se debe, principalmente, a dos motivos:

- Usualmente la construcción de modelos de simulación no forma parte del área de conocimiento ni de las habilidades requeridas para el puesto.

- Aun cuando el arquitecto posea la experiencia requerida, debe gastar tiempo y esfuerzo en el desarrollo del modelo, desvinculándolo de sus objetivos iniciales y sobrecargando sus actividades.

Además, si se piensa en un proceso de desarrollo ágil, la construcción de múltiples modelos de simulación que posibiliten la evaluación de distintas arquitecturas sólo contribuye a un retraso en los tiempos de entrega pactados (dando como consecuencia la ralentización del proceso de desarrollo). Este punto en particular se aleja de los objetivos planteados en términos de una estimación de calidad temprana con retorno rápido (a fin de corregir posibles desviaciones en los diseños).

Estas dificultades, sumadas a la falta de una especificación uniforme para la definición de arquitecturas web de forma independiente a su infraestructura, quedan ocultas en el modelo propuesto por medio del uso de un *metamodelo arquitectónico* (*architectural metamodel*). Este metamodelo permite crear instancias de patrones de diseño arquitectónicos como parte de una estrategia de representación web. Teniendo en cuenta que un ambiente de CC puede diseñarse en base a diferentes esquemas y, dado que la descomposición modular es una de las estrategias más difundidas para el diseño de arquitecturas de software, el modelo propuesto se basa en la especificación de componentes y conexiones.

La forma en la cual este metamodelo debe ser utilizado como mecanismo de soporte a la tarea de diseño arquitectónico, queda prescrita en un *proceso de diseño de arquitecturas web* (*web design process*) basado en una descomposición de sistema-de-sistemas. Esta descomposición es posible gracias a la selección de un estilo arquitectónico basado en capas. De esta manera, los componentes del metamodelo son utilizados como elementos fundamentales de este proceso, brindando una referencia para la construcción de arquitecturas de software para aplicaciones web de forma estandarizada. En el apartado *"Diseño de Arquitecturas Web"* se describe brevemente la forma en la cual se generan las arquitecturas web junto con el formato

de representación diseñado, mientras que en la sección *"Metamodelo UML-OCL para el Diseño de Arquitecturas Web"* se detallan las principales características del modelo propuesto.

En este contexto, cada componente arquitectónico identificado en el metamodelo es susceptible de ser transformado en un modelo de simulación específico. Dado que las composiciones y relaciones se encuentran explícitamente definidas en los elementos del metamodelo, los vínculos entre distintos modelos también pueden ser derivados (es decir, se parte del diseño arquitectónico original y se refleja el mismo conjunto de composiciones y conexiones en los modelos de simulación equivalentes). Bajo este esquema de transformaciones, la generación del modelo de simulación para la estimación de la calidad en base a una arquitectura específica, se convierte en una tarea transparente para el arquitecto (ya que este modelo puede derivarse automáticamente del conjunto de elementos web utilizado en su diseño). En el aparatado *"Modelos de Simulación para Elementos Arquitectónicos"* se describen brevemente los lineamientos definidos como base para el establecimiento de las reglas que permiten generar los modelos de simulación.

Sin embargo, la definición de modelos de simulación no es la única actividad requerida para la evaluación de diseños arquitectónicos con el objetivo de estimar la calidad. Los arquitectos de software también deben disponer de mecanismos que les permitan representar adecuadamente el comportamiento de los potenciales usuarios de la aplicación, a fin de generar un conjunto de valores de entrada útiles para alimentar el modelo. Por este motivo, además de los modelos de simulación para las arquitecturas web, se especifica un conjunto de modelos de simulación que representan *comportamientos de usuario* (*users behavior*) en base a diferentes clases de cargas de trabajo. En el apartado *"Modelos de Simulación para Comportamientos de Usuario"* se detallan los fundamentos básicos que guiaron el proceso de diseño y construcción de estos modelos.

3.2.1 Alcance

El modelo general propuesto en esta tesis abarca el conjunto de elementos requeridos para la definición de la calidad, la representación de la arquitectura, la obtención de un modelo de simulación equivalente y la generación de modelos de usuario asociados al dominio de aplicaciones web. No incluye ni el modelado ni la simulación de la infraestructura subyacente (es decir, aísla la definición de los elementos incluidos en la capa de aplicación).

Desde este punto de vista, contribuye a la resolución de los problemas descriptos en los apartados *"Nuevas Propiedades de Calidad"* y *"Representación del Diseño Arquitectónico"*. Además contribuye, parcialmente, a resolución del problema de *"Dependencia de la Infraestructura Subyacente"* ya que desvincula los componentes de las capas aplicación/infraestructura pero no plantea un mecanismo complementario que posibilite, a futuro, incorporar nuevamente esta dependencia con el objetivo de evaluar la calidad desde una perspectiva integral.

3.2.2 Nuevos Modelos de Software (Componentes de la Propuesta)

Esquema de Calidad (Quality Scheme)

Dado que la calidad está relacionada con todas las actividades del proceso de desarrollo, la especificación de los requerimientos de calidad debe elaborarse como parte de la primera etapa de trabajo (previo a que se construyan diseños asociados al producto). De esta manera, esta especificación puede ser utilizada tanto por los arquitectos de software como por los desarrolladores a fin de evaluar el ajuste de sus diseños e implementaciones con respecto a la calidad esperada del producto. La aplicación efectiva de estas especificaciones, especialmente en la evaluación de las decisiones arquitectónicas, genera un control de calidad temprano en el proceso de desarrollo.

En este contexto, es posible aislar un conjunto de propiedades de calidad iniciales tomando como punto de partida la *Especificación de Requerimientos de Software* en combinación con la aplicación de *Modelos de Calidad de Producto*. Teniendo en cuenta que la tarea de especificar atributos de calidad sin ambigüedad es difícil y tediosa, por medio de la aplicación de *Modelos de Calidad* es posible dirigir este proceso a fin de construir un documento de valor que posibilite la formalización de los aspectos de calidad relevantes en un producto dado. En el esquema de trabajo propuesto, este documento se denomina *Esquema de Calidad*.

Un *Esquema de Calidad* queda definido como un nuevo documento/artefacto de software que reúne tres dominios relevantes, a saber:

i) *Modelo de calidad*: Define los atributos de calidad (junto con sus relaciones de jerarquía) de forma precisa, mejorando la comunicación entre analistas, arquitectos y desarrolladores.
ii) *Producto de software*: Describe una descomposición de los principales elementos que forman parte de la definición del producto.
iii) *Métricas*: Especifica la forma de cálculo asociada a un determinado par *{componente de software, atributo de calidad}*.

Existen múltiples ventajas de la utilización de *Modelos de Calidad* como parte del proceso de desarrollo, entre las que se destacan:

- Validar la completitud de la definición de requerimientos.
- Identificar objetivos del diseño del producto de software.
- Identificar criterios de aceptación del usuario en relación a la completitud del producto de software.

Sin embargo, la aplicación de este tipo de modelos se ve limitada ya que, por si solos, no tienen la capacidad de comunicar la importancia de los distintos atributos de calidad. Luego, convencer a los clientes o a la gestión de un proyecto de software de la

importancia que tiene la validación de los atributos de calidad por encima de las funcionalidades requeridas, no es tarea sencilla.

En este sentido, el uso de los *Modelos de Calidad* como parte de *Esquemas de Calidad* visibiliza la importancia de los distintos atributos de calidad por medio del establecimiento de relaciones entre la forma de medición y los componentes del producto. De esta manera, los distintos grupos de interés evidencian la forma en la cual impacta cada aspecto de calidad sobre el producto final, dando lugar a una perspectiva de calidad aplicable a lo largo de todo el proceso de desarrollo. Además, la aplicación de *Modelos de Calidad* como componente de un documento específico contribuye a garantizar un entendimiento uniforme de los conceptos de calidad por parte del equipo de trabajo.

En el Capítulo 4 se presenta el *Modelo de Calidad de Producto de Software* propuesto en el estándar ISO/IEC 25010 (ISO/IEC 25010 2011) junto con la definición formal (genérica) de los *Esquemas de Calidad*. Teniendo en cuenta que para el caso específico de entornos de CC y servicios de software en la nube no existen *Modelos de Calidad* establecidos, en el Capítulo 6 se detalla la forma en la cual se ha extendido el modelo propuesto en ISO/IEC 25010 a fin de incorporar propiedades específicas de este dominio. A partir de este modelo, se construye un *Esquema de Calidad SaaS* aplicable al dominio bajo análisis.

Diseño de Arquitecturas Web

Tal como se ha enunciado con anterioridad, el diseño de arquitecturas SaaS es una de las actividades del proceso de desarrollo de software que requiere mayor nivel de pericia en lo referido a técnicas y herramientas aplicables a distintas situaciones. Teniendo en cuenta la rapidez con la que han evolucionado los productos basados en CC, es difícil establecer un conjunto específico de elementos que posibilite modelar cualquier tipo de solución arquitectónica requerida en el contexto de aplicaciones web.

Aun cuando pudiese delimitarse el conjunto de componentes aplicables, es necesario precisar la forma en la cual estos componentes deben ser estructurados.

De acuerdo con la literatura, el estilo arquitectónico asociado al nivel de diseño más alto de los entornos de CC se corresponde con un modelo de capas. En este tipo de modelos, los componentes residen sobre capas funcionales separadas y el acceso es permitido únicamente entre elementos de la misma capa y/o con su inmediata inferior (no permitiéndose el acceso entre capas no consecutivas). En este contexto, cada capa puede ser vista como un sistema independiente que forma parte de un sistema mayor (el CC). Siguiendo esta perspectiva, es posible definir un proceso basado en actividades que facilite la tarea de especificación de diseños arquitectónicos web por medio de una separación de los intereses asociados a cada capa.

En el Capítulo 7 se describe el proceso diseñado a fin de estructurar el diseño de aplicaciones web en base a los lineamientos de la teoría de *sistema-de-sistemas*. De esta manera, se aísla la definición de la aplicación web de la especificación de la infraestructura subyacente. Posteriormente, en el Capítulo 8, se define un metamodelo de componentes y conectores basado en el estudio de patrones de diseño arquitectónicos referidos a entornos de CC. Los conceptos generales identificados en este metamodelo pueden ser utilizados como mecanismos de soporte al proceso de diseño previamente detallado.

Modelos de Simulación para Elementos Arquitectónicos

Existen múltiples ventajas de emplear simulación en la etapa de diseño arquitectónico, entre las que se destacan: reducción del tiempo de desarrollo y verificación de las decisiones arquitectónicas en escenarios artificiales.

Tomando como punto de partida los elementos arquitectónicos identificados en la representación de diseños de arquitecturas de software web (descripta en la sección previa), se propone un conjunto de modelos de simulación basados en eventos

discretos que corresponden, individualmente, a los distintos tipos de componentes arquitectónicos definidos. Para esto, cada modelo de simulación propuesto ha sido diseñado utilizando la descripción de comportamiento asociada a sus responsabilidades como parte de los patrones de diseño que lo utilizan.

En base a esta definición individual, se plantea un mapeo directo entre la estructura del diseño arquitectónico y la estructura del modelo de simulación. Es decir:

Sea A una arquitectura de software web y sea M_A el modelo de simulación de eventos discreto equivalente al diseño A. Si C es un componente arquitectónico que pertenece a A y M_C es el modelo de simulación que representa el comportamiento genérico de C como parte de una arquitectura web cualquiera, entonces M_C se encuentra incluido en M_A de la misma forma en la que C se encuentra incluido en A. Luego M_A posee la misma estructura que A pero en términos de modelos de simulación.

De esta manera, el modelo de simulación resultante es semejante a la arquitectura. Esta particularidad tiene dos ventajas en relación a los arquitectos de software, a saber:

- Son capaces de comprender el contenido aun cuando no tengan conocimientos específicos del área de simulación.
- Pueden utilizar el modelo para la estimación de la calidad sin percibirlo como una barrera u obstáculo que ralentiza el proceso de desarrollo.

Siguiendo estos lineamientos, en el Capítulo 10 se presentan los modelos de simulación diseñados para los componentes arquitectónicos definidos y la forma en la cual se mantiene la estructura del diseño como resultado del proceso de transformación de la arquitectura al modelo de simulación.

Modelos de Simulación para Comportamientos de Usuario

Dado que el modelo de simulación de arquitectura se utiliza como una representación del software real, sus entradas deben obtenerse por medio del estudio de los posibles comportamientos de usuario. De esta manera, para completar el enfoque de simulación arquitectónica es necesario contar con algún mecanismo que actúe como modelo de entrada. Los conceptos de metamodelo están restringidos a componentes arquitectónicos y, por lo tanto, nada dicen sobre los patrones de comportamiento en relación a los cuales se generan las solicitudes de usuario.

Según Fehling, la *carga de trabajo* (*workload*) es la utilización de los recursos de tecnología de la información en los que se aloja una aplicación (Fehling y colab. 2014). Esto quiere decir que, en un contexto web, la carga de trabajo se produce como consecuencia de que los usuarios accedan a la aplicación y generen solicitudes que se traduzcan en trabajos que deben manejarse automáticamente. Por este motivo, usualmente, la carga de trabajo toma diferentes formas según el tipo de recurso de tecnología de la información sobre el cual se está midiendo.

Recientemente, algunos trabajos de investigación han propuesto soluciones basadas en el estudio de este tipo de problemas sobre recursos de hardware asociados a la infraestructura de un entorno de CC (Magalhães y colab. 2015; Zehe y colab. 2015). En este contexto, la carga de trabajo es vista como el conjunto de peticiones de recursos o de solicitudes de ejecución de tareas específicas que se repite de acuerdo con un patrón temporal.

Luego, el resultado de la carga de trabajo puede verse como tipos de patrones temporales que se evidencian de acuerdo a la forma en la cual arriban las solicitudes de los usuarios a la aplicación web. Este enfoque puede utilizarse como complemento de la simulación arquitectónica dado que pueden identificarse diferentes tipos de usuarios por medio de patrones temporales (por ejemplo, usuario periódico).

En el Capítulo 11 se presentan los modelos de simulación de eventos discretos diseñados (e implementados) a fin de generar solicitudes de usuarios según diferentes clases de patrones temporales. De esta forma, se proporciona un mecanismo útil para la generación de las solicitudes de usuario requeridas como entradas en el modelo de simulación arquitectónica.

3.2.3 Validación: Prueba de Conceptos

Teniendo en cuenta que el modelo propuesto combina múltiples soluciones para cada una de las áreas de interés, se propone una prueba de conceptos basada en la instanciación de un *Esquema de Calidad Web* sobre el cual se aplican los *criterios de evaluación arquitectónica* definidos a fin de determinar un subconjunto de variables de calidad a ser obtenidas como salida de los modelos de simulación. En base a este subconjunto se definen los modelos de simulación genéricos para los componentes arquitectónicos.

Luego, se formulan casos de análisis basados en un conjunto de patrones de diseño de aplicaciones web. En relación a estos escenarios, se construyen los modelos de simulación arquitectónica equivalentes a fin de obtener los valores asociados a las métricas especificadas. Tales modelos pueden ser combinados con los modelos de simulación de comportamientos de usuario a fin de garantizar un entorno experimental real (en términos de las condiciones de entrada).

Finalmente, dado que el estudio de patrones de diseño arquitectónico brinda un contexto de trabajo beneficioso para un subconjunto de propiedades de calidad específicas ante un problema de diseño puntual, se comparan los resultados obtenidos de la ejecución de los modelos de simulación con los rendimientos esperados de cada uno de los patrones analizados.

3.3 Construcción de los Modelos: Estrategias de Solución

3.3.1 Ontología: Modelo Semántico de Calidad

Una *ontología* es una especificación explícita de una conceptualización, es decir, una visión abstracta y simplificada del mundo que incluye los objetos, conceptos y las relaciones entre ellos en un dominio de interés (Gruber 1993). Su definición da lugar a una especificación inequívoca de la estructura del conocimiento de un dominio, permitiendo compartir y reutilizar el conocimiento y, en consecuencia, brindando la posibilidad de generar razonamiento automatizado (Orgun y Meyer 2008).

Este tipo de especificaciones ha sido utilizado como mecanismo de soporte al modelado en distintas áreas de la ingeniería de software, habiendo aumentado su difusión en los últimos años. Gran parte de las investigaciones se relacionan con el proceso de elicitación de requerimientos (Jwo y Cheng 2010; Pires y colab. 2011; Balushi, Sampaio y Loucopoulos 2013; Couto, Ribeiro y Campos 2014) y el estudio de productos de software intermedios (Graaf y colab. 2014; Abebe y Tonella 2015); mientras que otras líneas de trabajo abordan el desarrollo de herramientas de software basadas en ontologías a fin de respaldar el proceso de desarrollo (Henderson-Sellers 2011; García-Peñalvo y colab. 2012; Reinhartz-Berger, Sturm y Wand 2013).

En este contexto, las ontologías constituyen un mecanismo útil para especificar el conjunto de conceptos y relaciones establecidas en el dominio de calidad de producto de software. Teniendo en cuenta que el estudio de este dominio como parte del proceso de desarrollo se formaliza por medio de la definición de *Esquemas de Calidad*, se diseña una única ontología que refleja la composición de los tres dominios involucrados en su definición (modelo de calidad, métricas de software y producto de software). Cada dominio es modelado de forma independiente a fin de:

- *Facilitar la tarea de modelado por medio de la reutilización de modelos existentes*: Se pueden usar o adaptar ontologías de dominio existentes. Este es el caso del

dominio de métricas de software en el cual, a lo largo de los años, se han propuesto múltiples modelos semánticos (Kitchenham, Hughes y Linkman 2001; Olsina y Martín 2003; Bertoa, Vallecillo y García 2006).

- *Independizar la conceptualización de los dominios*: La estrategia de diseño modular independiente posibilita, en el caso de dominios para los cuales no existen modelos previos, el desarrollo de una definición aislada del contexto remanente. Esto permite formular nuevos modelos semánticos que representen de forma apropiada el dominio bajo estudio. Este es el caso del dominio de calidad basado en el estándar ISO/IEC 25010. Debido a que el modelo de calidad de producto propuesto en dicho estándar ha sido estructurado recientemente, los modelos semánticos de este dominio aún no se encuentran establecidos. Luego, es necesario formular una nueva ontología de dominio que brinde soporte a sus conceptos y relaciones.
- *Promover la reutilización de las ontologías diseñadas en nuevo contextos*: Las ontologías desarrolladas pueden usarse en nuevos problemas en los que se requiera representar el dominio modelado. Por ejemplo: *i)* el modelo de producto de software puede combinarse con modelos que representen el resto de los estándares propuestos en la serie ISO/IEC 25000 (ISO/IEC 25000 2014), *ii)* el modelo de métricas de software puede integrarse con ontologías de control de calidad, y *iii)* el modelo de producto de software puede utilizarse para componer un modelo de trazabilidad de productos intermedios.

En el Capítulo 5 se detalla la forma en la cual se han diseñado los modelos semánticos que brindan soporte a la definición de *Esquemas de Calidad*. Además, se describe el conjunto de reglas y consultas complementarias que permiten la aplicación de este tipo de modelos como estrategia de estudio de las propiedades de calidad requeridas en un producto de software dado. En base al diseño final, en el Capítulo 6 se genera una instancia de la ontología resultante como parte de la definición del *Esquema de Calidad Web*.

(ONTOLOGÍA) *Visión abstracta y simplificada del mundo que incluye los objetos, conceptos y las relaciones entre ellos en un dominio de interés (Gruber 1993).*

Tipos de Ontologías: Definición y Propiedades

En la actualidad existen muchas dimensiones aplicables para la clasificación de ontologías (Roussey y colab. 2011). De acuerdo a la clasificación basada en el *nivel de formalidad* propuesta por (Hadzic y colab. 2009), existen cuatro tipos de ontologías:

i) *Altamente informal*: Una ontología es altamente informal si se expresa en lenguaje natural. Por lo general, las definiciones de términos utilizadas en este tipo de ontologías son ambiguas debido a la ambigüedad propia del lenguaje natural.

ii) *Semi-informal*: Las ontologías semi-informales intentan resolver el problema de la ambigüedad restringiendo el lenguaje natural. Este tipo de ontologías se expresa haciendo uso de una estructuración del lenguaje natural a fin de mejorar su entendimiento.

iii) *Semi-formal*: Las ontologías semi-formales se expresan en un lenguaje artificial (no necesariamente con definición formal).

iv) *Rigurosamente formal*: Las ontologías rigurosamente formales proporcionan un conjunto de términos predefinidos que poseen una semántica formal. Además definen, complementariamente, un conjunto de teoremas y pruebas de propiedades (tales como la solidez y la integridad).

Por otra parte, de acuerdo con (Gómez-Pérez, Fernandez-Lopez y Corcho 2010), una ontología debe ser (idealmente):

- *Estable*: Una ontología- es estable, si y solo si, no permite deducir conclusiones inválidas.

- *Completa*: Una ontología es completa, si y solo si, permite deducir todas las conclusiones válidas posibles a partir de su vocabulario por medio de la aplicación de las reglas de deducción definidas.

La prueba de estas propiedades (es decir, de la integridad del modelo propuesto) solo puede llevarse a cabo en ontologías formales. La falta de formalidad que poseen los otros tipos de ontologías imposibilita su demostración. Sin embargo, si es posible demostrar la "falta de completitud" de una definición individual.

Dada una ontología, es posible deducir la "no-completitud" de la misma por medio de la ausencia de (por lo menos) una de las definiciones del conjunto de elementos requeridos en el marco de referencia establecido (Gómez-Pérez, Fernandez-Lopez y Corcho 2010). Este marco de referencia puede ser cualquier artefacto que defina la información mínima a ser incluida en el modelo semántico (por ejemplo, una especificación de requisitos o una descripción del mundo real). De acuerdo con este enfoque, si todas las entidades requeridas están representadas en la ontología (ya sea explícitamente o por medio de la aplicación de inferencias entre representaciones y axiomas), la ontología es completa.

En este contexto, la ontología diseñada para dar soporte a la definición de *Esquemas de Calidad* fue implementada en OWL Web Ontology Language. En consecuencia, los modelos semánticos propuestos para cada uno de los dominios bajo estudio se corresponden con ontologías semi-formales. Por este motivo, la evaluación de la integridad de los modelos individuales fue realizada verificando la existencia de todas las entidades requeridas (dada por su descripción de dominio) sobre la definición de la ontología.

Lenguajes para la Especificación de Reglas y Consultas

Semantic Web Rule Language (SWRL)

El lenguaje SWRL se utiliza para definir reglas de implicancia lógica entre un antecedente y un consecuente (W3C 2004). Cada uno de los modelos semánticos propuestos es complementado con un conjunto de reglas SWRL que ayudan a enriquecer su definición y verificar su contenido.

En este contexto, la incorporación de este tipo de reglas como complemento a la definición de las ontologías diseñadas permite verificar la integridad de las instancias creadas (a partir de los modelos propuestos) y predecir nuevo conocimiento (a partir de las instancias existentes).

En el caso de las ontologías propuestas, las reglas SWRL son utilizadas para mantener la consistencia requerida en las instancias de los diferentes dominios.

SPARQL Protocol and RDF Query Language

SPARQL es un lenguaje de consulta semántico para bases de datos capaz de recuperar y manipular información almacenada en formato RDF (W3C 2013). El resultado de este tipo de consultas es, normalmente, definido como un conjunto o grafo RDF (según el contenido de la consulta).

La ontología final se complementa con un conjunto de preguntas de competencia definidas en SPARQL que ayudan a explorar el contenido de las instancias creadas a partir de los conceptos y relaciones que la componen. De esta manera, los esquemas de calidad instanciados pueden ser utilizados para conocer los principales artefactos, métricas y atributos de calidad utilizados en un contexto específico.

3.3.2 Metamodelo UML-OCL para el Diseño de Arquitecturas Web

Un metamodelo puede definirse como “un modelo de un modelo”. Esto quiere decir que las instancias de este tipo especial de modelos corresponden a un nuevo tipo de modelo que, a su vez, puede ser instanciado (Atkinson y Kuhne 2003).

En la ingeniería de software, múltiples trabajos han utilizado este tipo de modelos con el objetivo de representar un dominio específico (Nissen y colab. 1996; Albin-Amiot y Guéhéneuc 2001; Tolvanen y Rossi 2003; Franch y colab. 2010). En este caso, dada la falta de mecanismos de soporte para la definición de arquitecturas web (expuesta en el apartado *"Representación del Diseño Arquitectónico"*), se define un metamodelo que posibilita la instanciación de diseños arquitectónicos en base a un conjunto de componentes y conectores.

Este modelo utiliza como fundamento un subconjunto de los patrones de diseño de arquitecturas basadas en CC propuestos en (Fehling y colab. 2014). Los componentes identificados a partir del estudio de estos patrones son clasificados de acuerdo a tres categorías, a saber:

- *Componentes de administración*: Componentes que se utilizan para gestionar automáticamente el comportamiento de la aplicación web por medio del monitoreo de los elementos que la componen.
- *Componentes de aplicación*: Componentes que se utilizan para detallar el comportamiento de la aplicación web. Se dividen en: *i) Componentes de aplicación definidos*: Elementos básicos que el arquitecto puede utilizar para construir su diseño. *ii) Componentes de aplicación no definidos*: Elementos específicos del dominio de la aplicación web que deben ser modelados por el arquitecto utilizando una composición de componentes funcionales.
- *Componentes funcionales*: Componentes que representan responsabilidades asociadas a funciones específicas. Estos componentes deben ser utilizados

para modelar el comportamiento de los componentes de aplicación no definidos.

En este contexto, el modelo final incluye los elementos asociados a cada una de las categorías junto con las relaciones identificadas entre los mismos (las cuales fueron obtenidas a partir del estudio de la descripción de los patrones).

A fin de especificar los distintos elementos identificados en términos de clases, relaciones y atributos, el metamodelo final fue especificado en UML (Unified Modeling Language). Este lenguaje de modelado gráfico permite especificar, construir y documentar un sistema, pudiendo aplicarse también para la representación de modelos de dominio (como es el caso del diseño de arquitecturas de software web). Además, brinda la posibilidad de complementar la descripción gráfica con expresiones formales definidas en OCL (Object Constraint Language). Estas descripciones permiten especificar restricciones semánticas que no pueden ser expresadas a partir de una notación gráfica. Luego, con el objetivo de garantizar la consistencia de los modelos instanciados, se especificaron restricciones OCL como complemento del metamodelo UML. Estas restricciones corresponden a invariantes que permiten validar las representaciones generadas según los patrones analizados.

En el Capítulo 8 se presentan los patrones de diseño utilizados como base para la definición del metamodelo junto con la especificación del modelo resultante.

3.3.3 Discrete Event System Specification (DEVS): Modelos de Simulación

Si bien todas las propuestas mencionadas en el apartado *"Evaluación de Arquitecturas de Software (Métodos Complementarios)"* pueden ser aplicadas para la evaluación de arquitecturas de software en situaciones específicas, resulta necesario desarrollar modelos formales que presenten un mayor grado de flexibilidad y reutilización. En lo referido a técnicas de simulación, uno de los enfoques más difundidos para el modelado de sistemas es el formalismo DEVS.

DEVS es un formalismo para el modelado y análisis de sistemas de eventos discretos generales, el cual ha sido diseñado en base a una especificación modular y jerárquica con el objetivo de simular sistemas dinámicos complejos utilizando una abstracción de eventos discretos (Zeigler, Praehofer y Kim 2000).

Para esto, el formalismo DEVS propone dos niveles de descripción aplicables al modelado de un sistema, a saber: *descripción de comportamiento* (modelo atómico) y *descripción de estructura* (modelo acoplado). Un *modelo atómico* describe el comportamiento autónomo de un sistema de eventos discretos como una secuencia de transiciones deterministas entre estados secuenciales, así como también la forma en la cual el sistema: *i)* reacciona ante eventos de entrada externos, y *ii)* genera eventos de salida. Por su parte, un *modelo acoplado* describe la estructura del sistema como una red de componentes interconectados que pueden corresponder tanto a modelos atómicos como a modelos acoplados.

En este contexto, este formalismo de modelado y simulación ha sido utilizado con el objetivo de especificar un subconjunto de los modelos de simulación requeridos para la evaluación de las arquitecturas web propuestas.

En el Capítulo 9 se presenta el formalismo DEVS, detallándose su especificación formal en términos de los distintos tipos de modelos que lo componen y sus características. Además, en el Capítulo 11 se detallan los modelos de simulación DEVS diseñados como parte de la propuesta para especificar el comportamiento de los distintos tipos de usuarios de las aplicaciones web.

(DISCRETE EVENT SYSTEM SPECIFICATION - DEVS -) *Formalismo para el modelado y simulación de sistemas de eventos discretos diseñado en base a una especificación modular y jerárquica por el Prof. Bernard Zeigler.*

Routed DEVS (RDEVS)

RDEVS es una extensión del formalismo DEVS que ha sido diseñada con el objetivo de encapsular el envío de mensajes entre componentes por medio de la definición de funciones de ruteo (Blas, Gonnet y Leone 2017a). Su estructura facilita la tarea de modelado y simulación en el contexto de problemas en los cuales existen múltiples componentes (a veces replicados) que deben conectarse de acuerdo a información que es recibida por medio de eventos externos.

En el Capítulo 9 se presentan las dificultades evidenciadas cuando se intenta aplicar DEVS en este tipo de problemas de ruteo, detallándose la definición formal de RDEVS como extensión del formalismo original junto con la prueba de clausura bajo acoplamiento. Complementariamente, en el Capítulo 10, se describe la forma en la cual se ha utilizado esta subclase de DEVS para generar los modelos de simulación requeridos a fin de facilitar la estructuración de las arquitecturas web analizadas.

Conclusiones

La calidad del software se ha convertido en un problema crítico de la ingeniería de software ya que afecta los costos de desarrollo de los productos, los tiempos de entrega y la satisfacción del usuario. En este contexto, los artefactos de software combinados con los modelos de documentación y simulación propuestos en esta tesis actúan como nexo para la estimación de la calidad en las etapas iniciales del desarrollo de aplicaciones web.

En este capítulo se ha desarrollado la estructura general de la propuesta diseñada a fin de presentar los elementos que la componen y establecer sus relaciones. La calidad queda definida en un nuevo tipo de documento (basado en la aplicación de Modelos de Calidad de Productos de Software), que se denomina Esquema de Calidad. El conjunto de modelos de simulación se deriva de un metamodelo de componentes arquitectónicos (elementos básicos) que ayudan a componer diseños de arquitecturas web a partir de los cuales es posible estimar su comportamiento a fin de evaluar la calidad. Se incluye además un conjunto de modelos de simulación complementarios que ayudan a representar distintos tipos de comportamiento asociados a las solicitudes de usuarios. Estos últimos permiten generar las entradas necesarias para la estimación de la calidad.

En los siguientes apartados (Parte II, III y IV) se abordan cada una de las partes que componen la estructura general propuesta, describiendo tanto los dominios de interés como así también los modelos desarrollados para dar soporte a las mismas.

Parte II

Especificación de la Calidad en Servicios de Software

Capítulo 4. Calidad en Productos de Software

De acuerdo con la IEEE, la calidad de software puede definirse como "el grado en el cual un software posee una combinación de atributos deseados" (IEEE 1061 1998). La delimitación del conjunto de elementos que define un nivel adecuado de calidad para un sistema o producto de software específico, es un problema altamente dependiente del contexto en el cual se esté trabajando (Kitchenham y Pfleeger 1996). En este sentido, aún no se ha establecido la forma en la cual debe interpretarse y evaluarse la calidad deseada en un producto específico a lo largo del proceso de desarrollo. En este capítulo se presenta un mecanismo de soporte para la documentación de la calidad de productos de software, el cual ha sido diseñado en base al modelo de calidad de producto detallado en el estándar ISO/IEC 25010 (ISO/IEC 25010 2011). El documento resultante se denomina Esquema de Calidad (Quality Scheme) y, su definición, permite que el equipo de desarrollo elabore una única especificación del conjunto de propiedades de calidad relevantes (junto con las métricas aplicables para su evaluación) sobre un producto de software dado. De esta manera, se formalizan las relaciones entre las propiedades de calidad y las entidades del producto de software.

4.1 ISO/IEC 25000: "Systems and Software Quality Requirements and Evaluation"

El objetivo de la serie ISO/IEC 25000 denominada "Requerimientos y Evaluación de la Calidad de Software y Sistemas" (Systems and Software Quality Requirements and Evaluation, SQuaRE) es la creación de un marco de trabajo común que facilite la evaluación de la calidad de los productos de software. En este sentido, esta serie organiza, amplia y unifica el conjunto de estándares que cubren los procesos de calidad del software asociados tanto a la especificación de requerimientos, como así también a la evaluación de la calidad resultante. Para tales fines, reemplaza a las normas ISO/IEC 9126: "Calidad de Productos de Software" (ISO/IEC 9126-1 2001; ISO/IEC TR 9126-2 2003; ISO/IEC TR 9126-3 2003) e ISO/IEC 14598: "Evaluación de Productos de Software" (ISO/IEC 14598 1999), las cuales representan la primera generación de estándares de calidad asociados a productos de software. Mientras que ISO/IEC 9126 describía el conjunto de características de un modelo de calidad de producto software, la norma ISO/IEC 14598 fijaba el proceso de evaluación de tales productos. En este sentido, la nueva serie de estándares conforma una segunda generación de normas de calidad asociadas a los productos de software.

4.1.1 Divisiones

Los estándares de la serie ISO/IEC 25000 se organizan en seis divisiones, a saber: *división de gestión de la calidad*, *división de modelos de calidad*, *división de medición de la calidad*, *división de requisitos de calidad*, *división de evaluación de calidad* y *división de extensión*. Cada división abarca un conjunto de normas de ámbito específico (Figura 4.1).

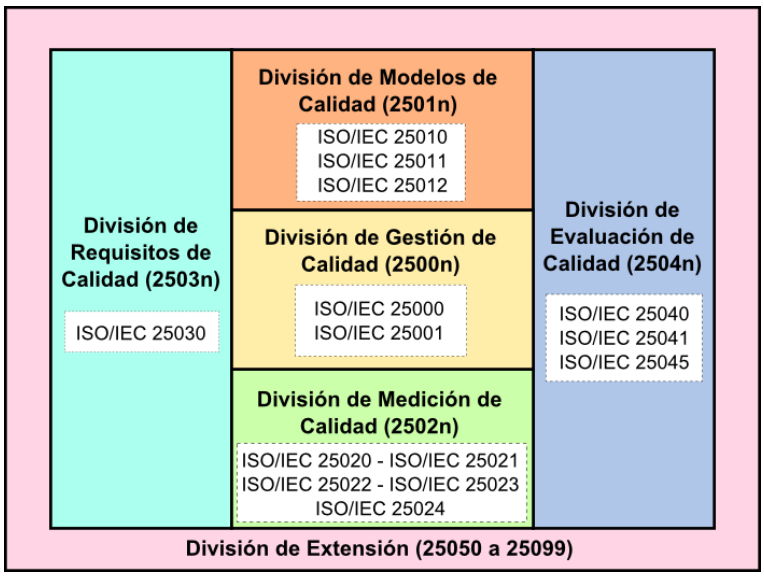


Figura 4.1. Divisiones que conforman la serie ISO/IEC 25000[1].

División de Gestión de la Calidad (ISO/IEC 2500n)

El conjunto de estándares incluidos en esta división especifica los modelos, términos y definiciones comunes a toda la serie. Incluye las normas ISO/IEC 25000 (ISO/IEC 25000 2014) e ISO/IEC 25001 (ISO/IEC 25001 2014).

La norma ISO/IEC 25000 se denomina "Guía para SQuaRE". Contiene el modelo de la arquitectura propuesta por la serie, la terminología, un resumen de las partes de cada uno de los estándares componentes, los usuarios previstos, un detalle de las secciones de interés para cada uno de ellos y los modelos de referencia. Por su parte, el estándar ISO/IEC 25001 ha sido designado como estándar de "Planeamiento y Gestión". En este estándar se establecen los requisitos y las orientaciones a ser utilizadas para gestionar la evaluación y especificación de los requisitos del software. La última versión de ambos estándares es del año 2014.

[1] Adaptado de (ISO/IEC 25000 2014).

División de Modelos de Calidad (ISO/IEC 2501n)

El objetivo del conjunto de normas incluidas en esta división es definir modelos de calidad útiles en ámbitos específicos. Incluye tres estándares, a saber: ISO/IEC 25010 (ISO/IEC 25010 2011), ISO/IEC 25011 (ISO/IEC TS 25011 2017) y ISO/IEC 25012 (ISO/IEC 25012 2008).

El estándar ISO/IEC 25010 denominado "Modelos de Calidad de Sistemas y de Software" detalla dos modelos de calidad diferentes aplicables a los sistemas y al software, a saber: un modelo de calidad para productos de software y un modelo de calidad referido a la calidad de uso. La versión vigente de este estándar corresponde al año 2011.

Por su parte, el estándar ISO/IEC 25011 denominado "Modelos de Calidad de Servicio" es aplicable a aquellos servicios de tecnología de la información que brindan soporte a un único usuario o negocio. Define un modelo de calidad aplicable al diseño, transición, entrega y mejora de los servicios de tecnología de la información. Además, provee una guía que establece como utilizar el modelo de calidad de uso de ISO/IEC 25010 para especificar la calidad de uso de este tipo de servicios. La versión actual corresponde al año 2017.

Finalmente, el estándar ISO/IEC 25012 denominado "Modelo de Calidad de Datos" detalla un modelo general para la calidad de los datos. Este modelo es aplicable sólo a aquellos datos que se encuentren almacenados de forma estructurada dentro de un sistema de información. La última versión publicada de este estándar es del año 2008.

División de Medición de Calidad (ISO/IEC 2502n)

Esta división de estándares contiene un conjunto de normas que presentan modelos de referencia para la medición de la calidad del producto de software, definiciones de medidas de calidad y guías prácticas para su aplicación. Dentro de su ámbito se encuentran incluidos cinco estándares, a saber: ISO/IEC 25020 (ISO/IEC 25020

2007), ISO/IEC 25021 (ISO/IEC 25021 2012), ISO/IEC 25022 (ISO/IEC 25022 2016), ISO/IEC 25023 (ISO/IEC 25023 2016) e ISO/IEC 25024 (ISO/IEC 25024 2015).

El estándar ISO/IEC 25020 se denomina "Guía y Modelo de Referencia para la Medición". Define un modelo de referencia común a los elementos de medición de la calidad. Además, proporciona una guía para que los usuarios seleccionen, desarrollen y/o apliquen las medidas propuestas por las normas ISO. La versión vigente de este estándar corresponde al año 2007.

Por su parte, el estándar ISO/IEC 25021 denominado "Elementos de Medición de la Calidad" especifica un conjunto recomendado de métricas que pueden utilizarse a lo largo de todo el ciclo de vida del proceso de desarrollo de un software. La última versión publicada de este estándar es del año 2012.

Finalmente, los estándares ISO/IEC 25022, 25023 y 25024 (denominados "Medición de la Calidad de Uso", "Medición de la Calidad del Sistema/Software" y "Medición de la Calidad de Datos", respectivamente) definen un conjunto de métricas asociadas a la medición de la calidad, haciendo referencia a distintos aspectos del software. Específicamente:

- El estándar ISO/IEC 25022 utiliza el conjunto de propiedades detalladas en el modelo de calidad de uso del estándar ISO/IEC 25010, para definir un conjunto de métricas asociadas a la calidad de uso.
- El estándar ISO/IEC 25023 utiliza las propiedades detalladas en el modelo de calidad de producto propuesto en el estándar ISO/IEC 25010, para definir métricas que permitan evaluar cuantitativamente la calidad de sistemas/software.
- El estándar ISO/IEC 25024 utiliza el conjunto de propiedades detalladas en el modelo de calidad de datos del estándar ISO/IEC 25012, para definir métricas de calidad de datos que permitan evaluar cuantitativamente la calidad de los datos asociados a un determinado esquema de información.

Las versiones vigentes de cada uno de estos estándares corresponden a los años 2016 (para ISO/IEC 25022 e ISO/IEC 25023) y 2015 (para ISO/IEC 25024).

División de Requisitos de Calidad (ISO/IEC 2503n)

Esta división especifica los requisitos de calidad que pueden ser utilizados durante el proceso de elicitación de requerimientos o como entrada del proceso de evaluación. Incluye únicamente al estándar ISO/IEC 25030 (ISO/IEC 25030 2007), denominado "Requerimientos de Calidad"; el cual provee un conjunto de recomendaciones para la especificación de los requisitos de calidad del software. La versión vigente es del año 2007.

División de Evaluación de Calidad (ISO/IEC 2504n)

Los estándares incluidos en esta división proporcionan requisitos, recomendaciones y guías para llevar a cabo el proceso de evaluación de un producto software. El conjunto de estándares incluidos comprende tres normas, a saber: ISO/IEC 25040 (ISO/IEC 25040 2011), ISO/IEC 25041 (ISO/IEC 25041 2012) e ISO/IEC 25045 (ISO/IEC 25045 2010).

El estándar ISO/IEC 25040, denominado "Guía y Modelo de Referencia para la Evaluación", propone un modelo general para la evaluación del software en el cual considera las entradas, las restricciones y los recursos requeridos para generar las salidas deseadas. La última versión publicada de dicho estándar corresponde al año 2011. Como complemento, el estándar ISO/IEC 25041 denominado "Guía de Evaluación para Desarrolladores, Adquisidores y Evaluadores Independientes", describe los requisitos y recomendaciones para la implementación práctica de la evaluación del software desde el punto de vista de diferentes actores. La versión vigente de este estándar es del año 2012. Finalmente, el estándar ISO/IEC 25045 denominado "Módulo de Evaluación para Recuperabilidad", especifica un módulo de evaluación (junto con su

documentación y estructura) a ser utilizado para evaluar el atributo de calidad recuperabilidad sobre un producto de software dado.

División de Extensión (ISO/IEC 25050 a 25099)

Esta división se reserva para normas o informes técnicos que traten dominios de aplicación específicos o que puedan ser utilizados para complementar otras normas de la serie. Actualmente incluye las normas ISO/IEC 25051 (ISO/IEC 25051 2014), 25062 (ISO/IEC 25062 2006), 25063 (ISO/IEC 25063 2014), 25064 (ISO/IEC 25064 2013) y 25066 (ISO/IEC 25066 2016) junto con el reporte técnico ISO/IEC TR 25060 (ISO/IEC TR 25060 2010).

El estándar ISO/IEC 25051 denominado "Requerimientos de Calidad de Productos de Software 'Ready-to-Use' e Instrucciones para su Testeo", establece requerimientos de calidad junto con requerimientos asociados a la documentación de pruebas, instrucciones para la evaluación de la conformidad y otros aspectos relacionados con productos de software "ready-to-use". La versión vigente es del año 2014.

Por su parte, los estándares numerados a partir del ISO/IEC 25060 (es decir, los numerados como 25060, 25062, 25063, 25064 y 25066) refieren a distintos aspectos asociados a una nueva familia de estándares internacionales denominada "Formato de Industria Común" (Common Industry Format, CIF). Específicamente se tiene que:

- El reporte técnico ISO/IEC TR 25060 describe una potencial familia de estándares internacionales (llamados CIF) que documentan la especificación y evaluación de la usabilidad de sistemas interactivos. La versión vigente corresponde al año 2010.
- El estándar ISO/IEC 25062 provee un método estandarizado para reportar los resultados de las pruebas de usabilidad en base al CIF. La versión vigente corresponde al año 2006.

- El estándar ISO/IEC 25063 detalla el CIF para las descripciones de contexto de uso de sistemas existentes, previstos, implementados o desplegados; especificando el contenido requerido tanto para descripciones de alto nivel como así también para especificaciones detalladas. La versión vigente corresponde al año 2014.
- El estándar ISO/IEC 25064 describe el CIF para los reportes de necesidades de usuarios, proveyendo una especificación para su contenido y formato. La versión vigente corresponde al año 2013.
- El estándar 25066 describe el CIF para el reporte de evaluaciones de usabilidad, proveyendo una clasificación de distintos enfoques de evaluación y la especificación del contenido a ser incluido en el mismo. La versión vigente corresponde al año 2016.

4.1.2 Relación entre Estándares e Importancia de los Modelos de Calidad

La serie de estándares SQuaRE define un marco teórico que contextualiza múltiples aspectos relevantes a ser considerados en relación a la calidad del software. Estos aspectos no son independientes, por lo que existe un conjunto de relaciones que permite vincular los estándares asociados a las distintas divisiones.

Los estándares de la "División de Gestión de la Calidad" definen un marco de soporte genérico para las divisiones restantes. Por su parte, las relaciones que especifican los vínculos entre las divisiones "Modelos de Calidad", "Medición de la Calidad", "Requisitos de Calidad" y "Evaluación de Calidad", quedan definidas por modelos de calidad. Un *modelo de calidad* es un conjunto de factores de calidad (junto con sus relaciones) que proporciona una base tanto para la especificación de requerimientos de calidad como así también para la evaluación de la calidad de los componentes de un producto de software (Carvallo y colab. 2004). Generalmente, se encuentra estructurado como una jerarquía de propiedades genéricas que se descomponen en factores específicos a distintos niveles de trabajo.

Ya sea para el trabajo con sistemas de software o en relación a los datos almacenados en tales sistemas, los estándares de la "División de Modelos de Calidad" definen múltiples modelos útiles para estudiar la calidad en los diferentes contextos. En relación con estos modelos de calidad, se evidencia que:

- Los estándares incluidos en la "División de Medición de la Calidad" detallan un conjunto de métricas aplicables para la valoración de los factores propuestos.
- Los estándares incluidos en la "División de Requerimientos de Calidad" especifican recomendaciones a ser tenidas en cuenta al momento de definir requerimientos de calidad, a fin de categorizar tales requerimientos haciendo uso de los factores propuestos.

Complementariamente, los estándares incluidos en la "División de Evaluación de la Calidad" refieren al proceso de evaluación de la calidad, en el cual es fundamental una buena definición de requerimientos de calidad basada en un conjunto de factores claramente identificados. Luego, los factores que componen los modelos de calidad son los elementos que transcienden las distintas divisiones. De esta manera, el conjunto de elementos definidos como parte de un modelo de calidad proporciona un marco de trabajo de utilidad para analizar la calidad del software en múltiples contextos.

(MODELO DE CALIDAD) *Conjunto de factores de calidad, junto con sus relaciones, que proporciona una base tanto para la especificación de requerimientos de calidad como así también para la evaluación de la calidad de los componentes de un producto de software (Carvallo y colab. 2004).*

4.1.3 Tipos de Modelos de Calidad

En las últimas décadas, los *modelos de calidad de software* se han convertido en poderosos mecanismos de soporte para la gestión de la calidad (Deissenboeck y colab. 2009). Muchos autores han centrado su atención en el estudio y desarrollo de modelos de calidad con el objetivo de fomentar su uso en diferentes contextos, ya que estos

modelos presentan una taxonomía de atributos de calidad junto con las relaciones que los vinculan, los cuales pueden ser utilizados como marco de referencia durante la elaboración de la especificación inicial, la evaluación del diseño y la prueba de un sistema de software (Albin 2003). En (Musa y Alkhateeb 2013), los autores presentan un análisis de los modelos clásicos junto con una comparativa de los mismos en relación a su aplicabilidad sobre los productos de software comerciales; mientras que, en (Dromey 1995; Lee, Lee y Kim 2009; Kläs, Lampasona y Münch 2013) se exponen distintos modelos de calidad desarrollados para especificar propiedades referidas a entornos específicos.

En términos generales, todo modelo de calidad de software tiene las siguientes propiedades estructurales:

1. *Número de capas*: Nivel de detalle con el que describe el dominio de calidad en relación al tipo de software para el cual ha sido construido.
2. *Tipos de elementos de modelado*: Generalmente incluye dos tipos de elementos, a saber: *i) elementos de alto nivel* (son elementos utilizados con propósitos de clasificación) y, *ii) elementos de bajo nivel* (corresponden a elementos utilizados con propósitos de descripción detallada y evaluación de características observables de los componentes).
3. *Propósito*: Refiere a su aplicabilidad en relación a las dimensiones específico/general y reutilizable/desechable. En general puede decirse que la reusabilidad de un modelo de calidad se reduce al hacerlo más específico y, en contraposición, se incrementa al hacerlo más general.
4. *Separación entre los elementos internos y externos*: Los factores externos son aquellos que pueden ser directamente percibidos por los usuarios, mientras que los factores internos hacen referencia a las características constructivas de los componentes.
5. *Relaciones entre factores de calidad*: Además de encontrarse vinculados por relaciones jerárquicas, los factores que componen el modelo se relacionan

por: *i) solapamiento* (un factor participa en la descomposición jerárquica de varios factores de niveles superiores, pudiendo evaluarse con métricas diferentes para cada factor al que descompone), *ii) transversalidad* (relación de solapamiento donde no sólo cambia la métrica, sino también la definición), y *iii) dependencia* (un factor se relaciona con otros factores, generalmente del mismo nivel).

De acuerdo con su propósito, existen dos formas de clasificar a los modelos de calidad: *modelos de calidad generales* y *modelos de calidad específicos* (Suman y Rohtak 2014). Mientras que los primeros son desarrollados con el objetivo de ser utilizados en cualquier tipo de software (es decir, sus atributos de calidad se eligen para ser aplicables sobre cualquier tipo de producto), los modelos específicos son diseñados para ser utilizados sobre una clase de software particular (por lo que sus atributos son elegidos para cubrir características de calidad propias de la clase elegida).

A su vez, los modelos de calidad pueden ser clasificados como *fijos*, *a medida* o *mixtos*, según la forma en la cual se definen los factores de calidad para su aplicación en proyectos particulares. La Tabla 4.1 resume esta clasificación. Como puede observarse, los *modelos de calidad fijos* suponen que contienen todos los factores de calidad posibles, por lo que un subconjunto de dichos factores es susceptible de ser aplicado a un proyecto concreto. En contraposición, los *modelos de calidad a medida* son diseñados a partir de la identificación de los objetivos a alcanzar, los cuales son utilizados como factores abstractos que se descomponen en factores concretos. En este último caso, los modelos son creados desde cero para cada proyecto. Los *modelos de calidad mixtos* plantean un enfoque intermedio en relación a los factores definidos y a la posibilidad de refinarlos para casos puntuales.

TIPO	DESCRIPCIÓN
FIJO	Existe un catálogo de factores de calidad que se usa como base para la evaluación.
A MEDIDA	Los factores deben ser identificados para cada proyecto.
MIXTO	Existe un conjunto de factores de calidad abstractos que son total o parcialmente reutilizados en todos los proyectos pero que, al mismo tiempo, pueden refinarse para un proyecto particular.

Tabla 4.1. Tipos de modelos de calidad de software según su uso en proyectos.

La selección del modelo de calidad a ser utilizado sobre un producto de software particular es un verdadero reto (Suman y Rohtak 2014). No sólo involucra la identificación de los factores de calidad apropiados para el producto bajo análisis, sino que también debe garantizar la flexibilidad requerida para ajustar su contenido a nuevas propiedades. Para poder realizar una verdadera comparación en un ámbito específico, es importante que el experto a cargo posea la destreza requerida para identificar el modelo de calidad apropiado conforme los requerimientos establecidos (Al-Badareen y colab. 2011).

En este contexto, el modelo de calidad de producto de software del estándar ISO/IEC 25010 (el cual se describe en el siguiente apartado) identifica un conjunto de factores de calidad abstractos que pueden ser utilizados en diferentes productos de software (modelo general) pero que, al mismo tiempo, pueden refinarse para su aplicación en un producto o entorno de software particular (modelo mixto) (Muñoz, Velthuis y Moraga de la Rubia 2010).

4.2 Modelo de Calidad de Producto del Estándar ISO/IEC 25010

El modelo de calidad de producto definido en ISO/IEC 25010 clasifica la calidad de los productos de software en base a un conjunto estructurado de características,

subcaracterísticas y atributos. Una *característica* (*characteristic*) representa una cualidad externa (es decir, una propiedad que puede ser experimentada por el usuario), la cual se logra por medio de un balance de las subcaracterísticas que la componen. Una *subcaracterística* (*subcharacteristic*) se manifiesta cuando el software es utilizado como parte de un sistema, pudiendo medirse de forma externa o interna en relación a un conjunto de atributos. Un atributo (*atribute*) es una entidad que puede ser verificada o medida sobre un producto de software.

En este contexto, una característica de calidad puede estar asociada a una o más subcaracterísticas de calidad y, a su vez, cada subcaracterística de calidad puede vincularse a uno o más atributos de calidad (Figura 4.2). Esta jerarquía de conceptos debe interpretarse entendiendo que la calidad global del producto estará dada por la evaluación resultante del conjunto de características de calidad propuestas. Tales características, a su vez, quedarán determinadas por la evaluación de las subcaracterísticas asociadas, las cuales resultan de la evaluación de los atributos que tenga asociados. En este sentido, el estándar no especifica la forma en la cual debe interpretarse la medición de la calidad a través de los distintos niveles de la jerarquía de propiedades. Esto implica que, quien haga uso del estándar, deberá proponer la forma en la cual deben agruparse las distintas mediciones a fin de lograr un único valor (o conjuntos de valores si es el caso) representativo de la característica de calidad asociada a los mismos.

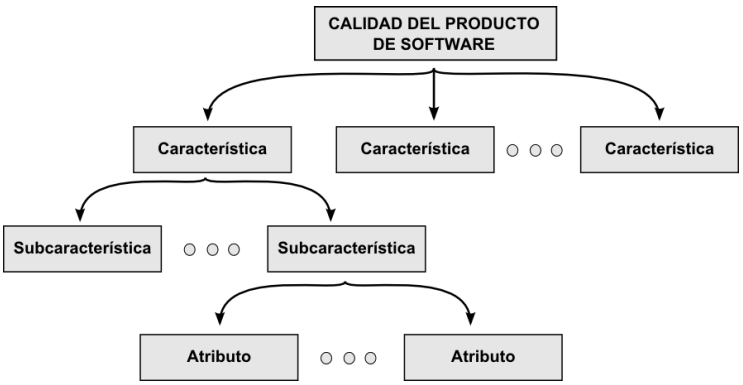


Figura 4.2. Jerarquía del modelo de calidad de producto del estándar ISO/IEC 25010.

El modelo de calidad propuesto en el estándar ISO/IEC 25010 modifica los elementos componentes del modelo de calidad definido en del estándar ISO 9126-1 (ISO/IEC 9126-1 2001). Esta actualización incorpora dos nuevas características y redefine el conjunto de subcaracterísticas asociadas a cada una de las características resultantes.

En este sentido, el nuevo modelo identifica ocho características que cubren propiedades tanto estáticas como dinámicas, a saber: *Funcionalidad (Functional Suitability)*, *Rendimiento (Performance Efficiency)*, *Compatibilidad (Compatibility)*, *Usabilidad (Usability)*, *Confiabilidad (Reliability)*, *Seguridad (Security)*, *Mantenibilidad (Maintainability)* y *Portabilidad (Portability)*. Para cada una de estas características, identifica un conjunto de subcaracterísticas asociadas. Por ejemplo, el modelo identifica tres subcaracterísticas asociadas a la característica "Rendimiento", a saber: *Capacidad (Capacity)*, *Comportamiento Temporal (Time Behavior)* y *Utilización de Recursos (Resource Utilization)*. La Figura 4.3 presenta la estructura basada en *Características* (*Characteristics*) y *Subcaracterísticas* (*Subcharacteristics*) definida en el *Modelo de Calidad* (*Quality Model*) del estándar.

Contrariamente a lo expuesto para las características y subcaracterísticas de calidad, la definición de atributos de calidad (nivel inferior de la jerarquía) no forma parte del modelo propuesto en el estándar. Esto se debe a que los atributos varían en función del tipo de producto de software a analizar, por lo que su incorporación debe realizarse por fuera del modelo.

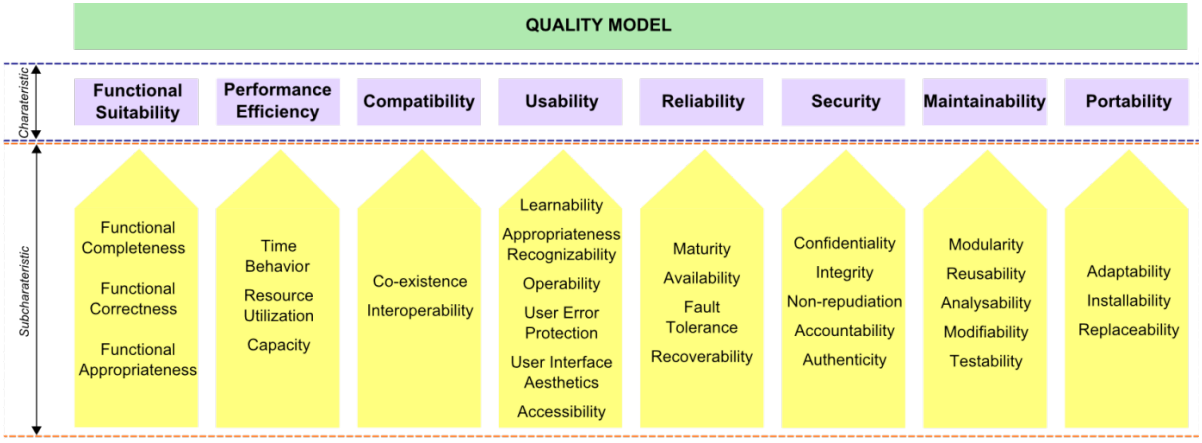


Figura 4.3. Modelo de calidad de producto del estándar ISO/IEC 25010.

De esta manera, el estándar provee un marco de trabajo para que los usuarios interesados en analizar la calidad de productos de software (organizaciones, entidades, desarrolladores, entre otros), definan un modelo de calidad específico basándose en los lineamientos (características y subcaracterísticas) del modelo propuesto. Es decir, la jerarquía propuesta de características y subcaracterísticas de calidad, provee una terminología consistente para la especificación, medición y evaluación de la calidad de un producto o sistema de software. La descripción de un modelo específico aplicable a un producto en particular, implica la identificación de los atributos de calidad de interés asociados a dicho producto, junto con su incorporación a la jerarquía propuesta. Esta definición queda a cargo de quien desee hacer uso del modelo propuesto en el estándar.

(CARACTERÍSTICA DE CALIDAD) *Cualidad externa de un producto de software que se logra por medio de un balance de las subcaracterísticas de calidad que la componen.*

(SUBCARACTERÍSTICA DE CALIDAD) *Se manifiesta cuando el software es usado como parte de un sistema, pudiendo medirse de forma externa o interna en base a un conjunto de atributos de calidad.*

4.2.1 Definición de las Características y Subcaracterísticas de Calidad

Funcionalidad (Functional Suitability)

Esta característica de calidad refiere a la capacidad del producto de software para proporcionar el conjunto de funciones que satisfacen las necesidades (tanto declaradas como implícitas) de los usuarios.

Queda conformada por tres subcaracterísticas de calidad, a saber: *Completitud* (*Functional Completeness*), *Correctitud* (*Functional Correctness*) y *Adecuación* (*Functional Appropriateness*). La subcaracterística "Completitud" refiere el grado en el cual el conjunto de funcionalidades del producto de software cubre el conjunto de tareas y objetivos especificados por el usuario. La subcaracterística "Correctitud" se relaciona con la capacidad del producto o sistema de software para proveer resultados correctos con un nivel de precisión requerido. Finalmente, la subcaracterística "Adecuación" refiere a la habilidad del producto de software para proporcionar un conjunto apropiado de funciones a fin de cumplir tareas y objetivos específicos.

Rendimiento (Performance Efficiency)

Esta característica de calidad representa el desempeño del producto de software en lo referente a la cantidad de recursos utilizados bajo condiciones previamente definidas.

Al igual que la característica "Funcionalidad", queda formada por tres subcaracterísticas de calidad, a saber: *Comportamiento Temporal* (*Time Behavior*), *Utilización de Recursos* (*Resource Utilization*) y *Capacidad* (*Capacity*). La subcaracterística "Comportamiento Temporal" refiere a la evaluación de tiempos de respuesta, procesamiento y tasas de "throughput" en condiciones de ejecución particulares (por ejemplo, ejecución normal, bajo estrés, entre otros); mientras que la subcaracterística "Utilización de Recursos" se vincula con la cantidad y tipos de recursos de hardware utilizados cuando el software es ejecutado en un contexto específico. Finalmente, la

subcaracterística "Capacidad" refiere al grado en el cual los límites máximos de un producto o sistema de software cumplen los requerimientos planteados.

Compatibilidad (Compatibility)

Esta característica de calidad se encuentra asociada a la capacidad de dos o más sistemas (o componentes de software), de intercambiar información y/o llevar a cabo sus funciones cuando comparten el mismo entorno de hardware o software.

Se divide en dos subcaracterísticas de calidad, a saber: *Coexistencia* (*Co-existence*) e *Interoperabilidad* (*Interoperability)*. La subcaracterística "Coexistencia" refiere a la capacidad del producto de software para residir en un entorno común en conjunto con otro (u otros) software, de forma tal que al compartir recursos no se generen problemas y/o inconvenientes en relación a sus funciones. Por su parte, la subcaracterística "Interoperabilidad" apunta a la habilidad de dos o más sistemas (o componentes de software), para intercambiar información y, luego, utilizarla como vehículo para cumplir con sus funciones.

Usabilidad (Usability)

Esta característica de calidad se vincula con la capacidad del producto de software para ser entendido, aprendido, usado y atractivo para con el usuario final.

Incluye seis subcaracterísticas de calidad, a saber: *Adecuación* (*Appropriateness Recognizability*), *Aprendizaje* (*Learnability*), *Operabilidad* (*Operability*), *Protección contra Errores de Usuario* (*User Error Protection*), *Estética de Interfaz de Usuario* (*User Interface Aesthetics*) y *Accesibilidad* (*Accessibility*). La subcaracterística "Adecuación" refiere a la habilidad del producto de software para dar a conocer sus capacidades y funciones, de forma tal que el usuario pueda comprender si el software se ajusta o no a sus necesidades. La subcaracterística "Aprendizaje" se relaciona con el grado en el cual un producto o sistema de software puede ser utilizado por un conjunto de usuarios, en un contexto específico; logrando cumplir sus objetivos de aprendizaje de forma tal que el

producto o sistema sea utilizado de forma eficiente, libre de riesgo y satisfactoria. La subcaracterística "Operabilidad" trabaja sobre el grado en el cual el producto de software permite que los usuarios lo operen y/o controlen, mientras que la subcaracterística "Protección contra Errores" se vincula con la capacidad del sistema para resguardar a los usuarios de cometer errores durante su uso. La subcaracterística "Estética de la Interfaz de Usuario" apunta a la habilidad del formato de presentación del producto de software para agradar y satisfacer la interacción con los usuarios. Finalmente, la subcaracterística "Accesibilidad" refiere a la capacidad del producto de software de ser utilizado por usuarios con determinadas discapacidades.

Confiabilidad (Reliability)

Esta característica de calidad representa la capacidad de un sistema o componente de software de llevar a cabo sus funciones cuando se lo utiliza bajo condiciones y periodos de tiempo determinados.

Queda dividida en cuatro subcaracterísticas de calidad, a saber: *Madurez* (*Maturity*), *Disponibilidad* (*Availability*), *Tolerancia a Fallos* (*Fault Tolerance*) y *Capacidad de Recuperación* (*Recoverability*). La subcaracterística "Madurez" refiere a la capacidad del sistema para satisfacer las necesidades de confiabilidad en condiciones normales de ejecución. Por su parte, la subcaracterística "Disponibilidad" trata la capacidad del sistema (o componente de software) de estar operativo y accesible para su uso en el instante en que se lo requiere. La subcaracterística "Tolerancia a Fallos" apunta a la habilidad del software de operar según lo previsto en presencia de fallos (tanto en relación al hardware como así también al software), mientras que la subcaracterística "Capacidad de Recuperación" se asocia con la destreza del componente de software para recuperar datos directamente afectados por un fallo y/o interrupción a fin de restablecer el estado deseado del sistema.

Seguridad (Security)

Esta característica de calidad se vincula con la capacidad del producto de software para mantener protegido tanto su información como sus datos, de forma tal que personas y/o sistemas no autorizados no tengan acceso a los mismos (tanto para leer y/o modificar).

Incluye cinco subcaracterísticas de calidad, a saber: *Confidencialidad* (*Confidentiality*), *Integridad* (*Integrity*), *No Repudio* (*Non-repudiation*), *Autenticidad* (*Authenticity*) y *Responsabilidad* (*Accountability*). La subcaracterística "Confidencialidad" refiere a la capacidad del producto de software para protegerse de accesos no autorizados a datos e información (ya sean accidentales o deliberados). La subcaracterística "Integridad" se vincula con la habilidad del sistema de software para prevenir accesos y/o modificaciones no autorizados a datos o programas del equipo sobre el que está siendo ejecutado. La subcaracterística "No Repudio" representa la capacidad del producto de software para demostrar las acciones o eventos que han tenido lugar, de forma tal que estas acciones o eventos no puedan ser repudiados posteriormente. La subcaracterística "Autenticidad" se asocia a la habilidad del producto para demostrar la identidad de un sujeto o un recurso, mientras que la subcaracterística "Responsabilidad" se vincula con la capacidad del producto para rastrear de forma inequívoca las acciones de una entidad (ya sea una persona u otro sistema de software).

Mantenibilidad (Maintainability)

Esta característica representa la capacidad del producto de software para ser modificado de forma efectiva y eficiente ya sea por necesidades evolutivas, correctivas o perfectivas.

Al igual que la característica "Seguridad", queda conformada por cinco subcaracterísticas de calidad, a saber: *Modularidad* (*Modularity*), *Reusabilidad* (*Reusability*), *Analizabilidad* (*Analysability*), *Modificabilidad* (*Modifiability*) y *Capacidad de*

Testeo (*Testability*). La subcaracterística "Modularidad" trata la capacidad de un sistema o producto de software (formado por componentes discretos) de generar un mínimo impacto sobre los componentes cuando se realiza una modificación sobre alguno de ellos. La subcaracterística "Reusabilidad" refiere a la habilidad de un componente de software para ser utilizado en más de un sistema (o en la construcción de otros componentes). La subcaracterística "Analizabilidad" se asocia a la facilidad con la cual se puede evaluar el impacto de un cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos y/o identificar las partes a modificar ante una eventual reformulación de componentes. La subcaracterística "Modificabilidad" referencia la habilidad del producto de software para ser modificado de forma efectiva y eficiente, sin introducir defectos o degradar su desempeño actual. Finalmente, la subcaracterística "Capacidad de Testeo" refiere la facilidad con la cual es posible establecer criterios de prueba (tanto para un sistema como para un componente) junto con la existencia de un nivel adecuado de simplicidad para llevar a cabo tales pruebas (a fin de determinar si se cumplen o no los criterios establecidos).

Portabilidad (Portability)

Esta característica de calidad se asocia con la capacidad del producto o componente de software de ser transferido, de forma efectiva y eficiente, de un entorno de ejecución a otro diferente (tanto en lo referente al hardware como así también al software).

Se divide en tres subcaracterísticas, a saber: *Adaptabilidad* (*Adaptability*), *Facilidad de Instalación* (*Installability*) y *Capacidad de Reemplazo* (*Replaceability*). La subcaracterística "Adaptabilidad" se asocia a la capacidad del producto de software para adecuarse de forma efectiva y eficiente a entornos de hardware y software predefinidos. La subcaracterística "Facilidad de Instalación" representa el grado en el cual es posible instalar/desinstalar el producto exitosamente en un entorno determinado. Por último, la subcaracterística "Capacidad de Reemplazo" se vincula con

la habilidad del producto de software para ser utilizado en lugar de otro software, con el mismo propósito, en el mismo entorno.

4.3 Especificación de la Calidad en el Proceso de Desarrollo

Todos los actores involucrados en el proceso de desarrollo de un producto o sistema de software son responsables de la calidad (Pressman 2010). El cumplimiento de los atributos de calidad requeridos debe ser considerado durante el diseño, la implementación y el despliegue de cualquier producto (Bass, Clements y Kazman 2012). Sin embargo, mantener la trazabilidad de estos atributos a lo largo de todas las etapas del proceso de desarrollo no es una tarea sencilla. Frecuentemente, los atributos se pierden al pasar de una etapa a la siguiente debido a que no existen mecanismos que brinden un soporte apropiado para conservar las decisiones de calidad que han sido tomadas en etapas precedentes. Sumado a esto, es usual que el equipo de desarrollo no conozca la forma en la cual se relacionan los distintos atributos de calidad ni el impacto que tales atributos tienen en la calidad global del producto en desarrollo. Esta falta de conocimiento da como consecuencia que las necesidades de calidad se evidencien en etapas tardías del proceso de desarrollo, como ser la fase de operación y mantenimiento (Khoshgoftaar, Liu y Seliya 2004). En esto casos, el costo de solucionar problemas asociados a la calidad, es demasiado alto.

En este contexto, el estudio de la calidad del software en relación a los equipos de desarrollo continúa siendo una de las áreas más relevantes de la ingeniería de software (Ampatzoglou, Frantzeskou y Stamelos 2012). La especificación de la calidad de los productos de software como elemento adicional a la especificación funcional, constituye un anexo crítico deseable, dado que permite distinguir propiedades del producto tales como facilidad de aprendizaje y disponibilidad (Zeist y Hendriks 1996).

4.3.1 Esquema de Calidad

Definición Formal

En un proceso de desarrollo tradicional, la Especificación de Requerimientos de Software es el artefacto que establece el conjunto de requerimientos asociados a un producto de software. En base al contenido de esta especificación, el equipo de trabajo puede aislar el conjunto de propiedades de calidad relevantes, haciendo uso de un modelo de calidad previamente definido. Posteriormente, puede utilizar estas propiedades como base para la especificación un nuevo tipo de artefacto, en el cual se indique no sólo las propiedades a analizar, sino también la forma en la cual tales propiedades deben ser evaluadas sobre el producto de software deseado. Esta nueva especificación se denomina Esquema de Calidad (Quality Scheme) y ha sido presentada en (Blas, Gonnet y Leone 2017b).

Un *esquema de calidad* es un conjunto de elementos asociado a la definición de un producto de software, en el cual cada elemento queda compuesto por:

i) Un *atributo de software*: Este componente identifica un módulo (elemento componente) asociado a una entidad específica del producto de software, determinando el elemento sobre el cual se evaluará la calidad.
ii) Una *métrica*: Este componente indica la forma de medición a ser utilizada para relevar la calidad del atributo.
iii) Una *subcaracterística de calidad*: Este componente vincula la evaluación de la calidad con un contexto de propiedades de calidad específico.

De esta manera, un esquema de calidad vincula propiedades de calidad con métricas específicas, las cuales deben ser aplicadas sobre un conjunto definido de componentes del producto de software bajo análisis. Las propiedades de calidad identificadas deben estar asociadas a algún modelo de calidad existente. Esto garantiza un correcto entendimiento de las relaciones entre propiedades de calidad.

Para analizar esta definición, tómese como ejemplo el sitio web de una juguetería online (producto de software bajo análisis). Considérese además que se utilizará el modelo de calidad de producto detallado en el estándar ISO/IEC 25010 como contexto de referencia para las propiedades de calidad. De acuerdo con la definición precedente, la especificación de un esquema de calidad para este producto específico debe partir de la identificación de los atributos de software asociados a las entidades relevantes. En este caso, se considera relevante a la entidad "información". Esto se debe a que el producto manipulará datos asociados a las compras de los usuarios, los cuales deben protegerse de accesos malintencionados. Sobre esta entidad relevante, pueden identificarse dos atributos a analizar: "seguridad de acceso" y "seguridad de almacenamiento". Las métricas "cantidad de accesos no autorizados" y "cantidad de accesos" pueden ser utilizadas para evaluar las subcaracterísticas "Confidencialidad" (*Confidentiality)* e "Integridad" (*Integrity)*, respectivamente (ambas asociadas a la característica "Seguridad" -*Security*-). Luego, estas métricas y subcaracterísticas pueden relacionarse con los atributos de software previamente identificados. Haciendo uso de este conjunto de elementos, la Tabla 4.2 presenta una abstracción del esquema de calidad resultante asociado al producto de software bajo análisis.

Para mayores detalles sobre este ejemplo, se sugiere remitirse a (Blas, Gonnet y Leone 2017b).

CALIDAD		**MEDICIÓN**	**SOFTWARE**		
Característica	*Subcaracterística*	*Métrica*	*Atributo*	*Entidad*	*Producto*
Seguridad	Confidencialidad	Cantidad de accesos no autorizados	Seguridad de acceso	Información	Juguetería Online
	Integridad	Cantidad de accesos	Seguridad de almacenamiento		

Tabla 4.2. Abstracción del esquema de calidad de una juguetería online.

Como puede observarse en el ejemplo planteado, el esquema de calidad resultante detalla una directa relación entre la calidad y los componentes del producto, por medio de la definición de métricas; dando visibilidad a los requerimientos de calidad asociados. Al vincular las propiedades de calidad con un contexto de referencia se provee al equipo de trabajo una definición única de los conceptos de calidad, no dando lugar a ambigüedades.

Trazabilidad

La ausencia de un mecanismo de soporte para la documentación de la calidad afecta al proceso de desarrollo desde dos puntos de vista: *i)* se presenta una falta de seguimiento de los requerimientos de calidad (es decir, de los requerimientos no funcionales) a lo largo de las distintas etapas de trabajo y, *ii)* se imposibilita una correcta comprensión (por parte del equipo de desarrollo) del impacto de los aspectos de calidad sobre las entidades de software.

Al generar un esquema de calidad en las primeras etapas del proceso de desarrollo (es decir, luego de la especificación de la ERS), la información asociada a las distintas partes del documento queda disponible para todos los grupos de interés (*stakeholders*) del producto. En este contexto, no sólo se gana visibilidad acerca de la importancia de los aspectos de calidad sino que, además, se provee una descripción uniforme para su tratamiento. Bajo esta perspectiva, el uso de esquemas de calidad como técnica de documentación mejora tanto el entendimiento como así también el relevamiento y la medición de los requerimientos no funcionales.

Así mismo, si el contenido del documento se actualiza conforme avanzan las distintas etapas de desarrollo, es posible mantener la trazabilidad de la calidad de manera uniforme en un único documento a lo largo de todo el proceso. Es decir, un esquema de calidad definido en etapas tempranas, puede ser utilizado, mejorado y mantenido mientras el proceso de desarrollo se lleva a cabo. En este sentido, la especificación de un esquema de calidad contribuye de la siguiente manera:

i) *En la etapa de diseño arquitectónico*: Brinda una clara definición de los requerimientos de calidad que deben marcar los lineamientos de la arquitectura propuesta para el producto de software bajo desarrollo, dando un marco de trabajo útil para evaluar la adecuación de distintas propuestas.

ii) *En la etapa de codificación*: Define inequívocamente la forma de medición aplicable para relevar el cumplimiento de los requerimientos de calidad a medida que se avanza con el desarrollo, permitiendo determinar la adecuación del mismo en una etapa previa a las pruebas.

iii) *En la etapa de pruebas*: Contribuye a determinar fehacientemente tanto la satisfacción de los requerimientos de calidad, como así también un conjunto de valores asociados a las distintas propiedades de calidad de alto nivel (definidas en términos de subcaracterísticas y características de calidad).

iv) *En la etapa de operación y mantenimiento*: Brinda soporte para analizar el impacto de la aplicación de cambios sobre entidades de software vinculadas a los aspectos de calidad definidos, permitiendo comparar distintos escenarios conforme se incorporen las modificaciones.

De esta manera, los beneficios de usar esquemas de calidad como mecanismos de documentación durante el proceso de desarrollo son múltiples, contribuyendo a mejorar de forma integral las tareas involucradas como parte de la construcción de un producto de software.

(ESQUEMA DE CALIDAD) *Conjunto de elementos asociado a la definición de un producto de software, en el cual cada elemento queda compuesto por un atributo de software, una métrica y una subcaracterística de calidad.*

4.3.2 Componentes de un Esquema de Calidad

Tal como se ha definido con anterioridad, un esquema de calidad queda formado por: *i)* un producto de software, *ii)* un conjunto de métricas aplicables a las entidades

del producto y, *iii)* un contexto de referencia de atributos de calidad utilizado para clasificar los requerimientos no funcionales del producto. En base a estos componentes, el esquema de calidad define un conjunto de relaciones que vinculan la información a fin de documentar la calidad del producto a desarrollar de forma coherente y concisa.

En este contexto, es importante definir el alcance de cada uno de estos componentes a fin de comprender la información requerida para desarrollar esquemas de calidad útiles como mecanismo de documentación.

Producto de Software

De acuerdo con (ISO/IEC 25001 2014), un producto de software puede ser definido como un conjunto de programas de computadoras junto con sus procedimientos y toda la información y/o documentación existente sobre los mismos, que ha sido diseñado para cumplir los requerimientos de un usuario específico. Incluye siempre el desarrollo de al menos un programa de computadora.

Usualmente, a lo largo de las distintas etapas del proceso de desarrollo que tiene como objetivo la construcción de los programas asociados a un producto de software, se crean múltiples artefactos. En este sentido, un artefacto es un producto que resulta del desarrollo de un software, el cual contiene información asociada una o más partes del mismo. Mientras que algunos artefactos ayudan a describir la arquitectura y el diseño del producto de software, otros son útiles para gestionar los distintos aspectos propios del proceso de desarrollo. Todos los artefactos pueden dividirse en un conjunto de entidades. Una entidad es un objeto que puede ser caracterizado por medio de la medición de alguno de sus atributos. Por lo tanto, un atributo es una propiedad, física o abstracta, asociada a una entidad que puede ser medida de forma directa o indirecta. Es decir, un atributo es una característica inherente a una entidad de software que puede relevarse cuantitativa o cualitativamente por medios físicos o automatizados.

Métricas de Software

En los últimos años se han realizado numerosos trabajos de investigación con el objetivo de definir estrategias para la medición de la calidad de los productos de software. Como parte de la serie ISO/IEC 25000, se propone un modelo de referencia junto con una guía para la medición de las características definidas en los distintos modelos de calidad detallados en la serie. Sin embargo, estos estándares solo proveen lineamientos generales. Es decir, definen un marco de trabajo para la medición, pero no especifican la forma en la cual este marco debe ser aplicado sobre un caso concreto. Sin embargo, debido a que especifican una terminología común, es posible estudiar los principales conceptos asociados a la medición de la calidad (junto con sus relaciones) aplicando esta perspectiva. Otras perspectivas similares en relación al dominio de métricas de software se presentan en (Pressman 2010; Fenton y Bieman 2014).

Las métricas de software pueden ser clasificadas de acuerdo a tres categorías, a saber: *métricas de producto*, *métricas de proceso* y *métricas de proyecto* (Kan 2003). Una *métrica de producto* describe las características asociadas al producto de software, mientras que una *métrica de proceso* puede ser utilizada para mejorar el desarrollo y mantenimiento del mismo. Por su parte, una *métrica de proyecto* describe características relacionadas con el desarrollo y ejecución del proyecto.

De acuerdo con (ISO/IEC 25023 2016), toda métrica queda definida en base a nueve elementos, a saber: *i)* nombre, *ii)* propósito, *iii)* método de aplicación, *iv)* fórmula de medición, *v)* interpretación, *vi)* tipo de escala, *vii)* tipo de medición, *viii)* entradas del proceso de medición y, *ix)* grupo de interés. La identificación de una métrica está dada por su *nombre*, motivo por el cual debe ser único para cada métrica especificada sobre un mismo producto. Además, a fin de facilitar su interpretación, debe ser representativo de la información obtenida como resultado de la aplicación de la métrica. El *propósito* es usualmente expresado como una pregunta a ser respondida luego de que la métrica ha sido aplicada sobre el producto. Por su parte, el *método de*

aplicación provee el contexto en el cual debe aplicarse la métrica, mientras que la *fórmula de medición* indica la expresión matemática a ser utilizada para el cálculo (explicando además el significado de cada uno de los elementos intervinientes en el mismo). La *interpretación* del valor medido provee el rango de valores en el cual dicho valor es de utilidad, pudiendo indicar también uno o más valores de preferencia. El *tipo de escala* define la dimensión de la métrica, siendo normalmente nominal, ordinal, de intervalo, de proporción o absoluta. El *tipo de medición* especifica la forma en la cual la métrica es relevada, quedando usualmente definido como dimensión, tiempo o enumeración. Las *entradas del proceso de medición* refieren a la fuente de información a ser utilizada durante la medición. Finalmente, el *grupo de interés* identifica a los usuarios que harán uso de los resultandos de la medición.

Contexto de Referencia de Calidad

Como se ha mencionado previamente, el objetivo de este componente como parte de un esquema de calidad es la clasificación de los requerimientos no funcionales del producto (es decir, los requerimientos de calidad). En este contexto, el uso de modelos de calidad es una buena propuesta, ya que proveen los lineamientos necesarios para describir, evaluar y/o estimar la calidad (Deissenboeck y colab. 2009). Al establecer una taxonomía estándar de atributos de calidad, sirven como marco de trabajo para la especificación tanto del sistema de software como así también para la ejecución de las pruebas relacionadas con la calidad (Albin 2003). Por lo tanto, los modelos de calidad son mecanismos de soporte ampliamente aceptados para gestionar la calidad de sistemas y/o productos de software (Deissenboeck y colab. 2009).

Al igual que en el caso de las métricas de software, los modelos de calidad de software se clasifican de acuerdo a su enfoque de evaluación (Callejas-Cuervo, Alarcón-Aldana y Álvarez-Carreño 2017). Esta clasificación trae como consecuencia la tipificación de tres clases de modelos de calidad: *modelos de calidad de producto*, *modelos de calidad de uso* y *modelos de calidad de proceso*. Los esquemas de calidad se vinculan

específicamente con productos de software, por lo que el contexto de referencia de calidad a utilizar como elemento componente de tales esquemas, debe corresponder a un modelo de calidad de producto.

En este sentido, múltiples modelos de calidad de producto de software han sido desarrollados a lo largo de los años. Entre los más conocidos se encuentran el modelo de calidad de McCall (McCall, Richards y Walters 1977), el de Boehm (Boehm y colab. 1978) y el propuesto en (ISO/IEC 9126-1 2001). Todos estos modelos presentan una definición jerárquica de conceptos (en algunos casos similares y, en otros, no tanto), que refieren a distintas características del producto de software a evaluar por medio del uso de métricas e indicadores. Diversos autores han realizado estudios comparativos sobre estos modelos (Scalone 2006; Al-Badareen y colab. 2011; Callejas-Cuervo, Alarcón-Aldana y Álvarez-Carreño 2017), detallando las ventajas y desventajas que encuentran al utilizar cada uno de ellos. Partiendo de estas comparaciones, se evidencia que los modelos de calidad clásicos han sentado las bases requeridas para la formulación de los modelos de calidad más recientes, permitiendo que los modelos de calidad actuales se consoliden como los más completos en relación a la evolución que ha sufrido la industria del software en los últimos años. En este contexto, en el año 2011, un nuevo modelo de calidad ha sido presentado como evolución del modelo propuesto en ISO/IEC 9126. Este nuevo modelo corresponde al modelo de calidad de producto del estándar ISO/IEC 25010 (ver apartado *"Modelo de Calidad de Producto del Estándar ISO/IEC 25010"*). En (Suman y Rohtak 2014) se presenta una comparación de este nuevo modelo con los precedentes.

4.3.3 Utilidad Práctica de los Esquemas de Calidad

De acuerdo con (Pressman 2010), el uso de métricas de software como mecanismos disparadores de estrategias que busquen mejorar la calidad del producto final, es una buena práctica. Bajo esta perspectiva, la definición de un esquema de calidad en un producto de software presenta un mecanismo de soporte para gestionar

la calidad. Su aplicación en las distintas etapas de desarrollo posibilita analizar (a lo largo del proceso) los resultados de las mediciones de calidad de forma integral. Tales mediciones pueden ejecutarse como parte de las tareas que se llevan a cabo en cada etapa (en la medida que su definición lo permita), a fin de garantizar que los resultados parciales (obtenidos de los productos intermedios) persiguen los objetivos de calidad definidos para el producto final.

Sin embargo, previo a su aplicación, debe recolectarse toda la información necesaria para la medición. En este sentido, la definición del esquema de calidad debe permitir que el equipo de desarrollo determine cuál es la mínima información requerida para lograr la evaluación de las métricas propuestas. En el caso de que esta información no se encuentre disponible, el esquema debe proveer los mecanismos necesarios para que el equipo de desarrollo identifique el conjunto de métricas que pueden ser evaluadas haciendo uso de los datos existentes. De esta forma, se tiene la posibilidad de contemplar las métricas no calculables por falta de información como parte de la evaluación, generando una evaluación parcial (de la calidad del producto final) que indique el subconjunto de propiedades no relevadas.

Luego, los esquemas de calidad son útiles desde múltiples puntos de vista. Además de definir una conceptualización práctica de las propiedades de calidad requeridas para un producto de software específico, permiten obtener resultados certeros sobre su evaluación, admitiendo la posibilidad de analizar la información necesaria para obtener tales valoraciones. En el caso de que no sea posible analizar la totalidad de los requerimientos de calidad especificados en el documento, el esquema de calidad posibilita la determinación del subconjunto de propiedades de calidad que pueden ser estudiadas haciendo uso de los datos disponibles, facilitando la interpretación de los mismos como parte de la calidad integral requerida para el producto.

Conclusiones

La tarea de documentación (tanto del proceso como del producto) constituye una actividad integral a lo largo del desarrollo, ya que contiene toda la información necesaria para construir, utilizar y mantener un sistema de software. Sin embargo, en la práctica, la creación de documentos apropiados para tales fines es frecuentemente desatendida (Bayer y Muthig 2006). Esta situación, sumada a la falta de claridad acerca de cómo debe interpretarse la calidad a lo largo del proceso de desarrollo de los productos de software, dificulta la formulación de mecanismos alternativos que permitan acompañar el estudio de la calidad en las distintas etapas de trabajo.

En este capítulo se ha presentado una alternativa para la documentación de las métricas a utilizar para evaluar la calidad requerida en un producto de software genérico, tomando como base la definición de un conjunto de propiedades de calidad específicas. Para esto, se ha definido el concepto de esquema de calidad desde un punto de vista formal, dando como resultado la identificación de tres dominios de trabajo vinculados.

En el siguiente capítulo se modelan estos tres dominios haciendo uso de ontologías, a fin de permitir una instanciación adecuada de esquemas de calidad válidos como parte de la especificación de un producto de software. Dados los beneficios del uso de ontologías como estrategia de modelado, se implementa su definición y se habilita la posibilidad de inferir nuevo conocimiento en base a los términos y relaciones establecidas.

Capítulo 5. Ontología para la Especificación de Esquemas de Calidad

De acuerdo al capítulo previo, un esquema de calidad puede usarse como documento de soporte para gestionar los requerimientos no funcionales a lo largo del proceso de desarrollo. Sin embargo, su especificación no puede realizarse de forma arbitraria ya que debe respetar el conjunto de componentes propuestos y las relaciones requeridas entre ellos. Esto se debe a que los elementos que conforman un esquema de calidad presentan propiedades y relaciones semánticas que deben garantizarse a fin de obtener un documento de valor. En este capítulo se propone usar ontologías como mecanismo base para la definición de un marco de trabajo conceptual que facilite la especificación de instancias de esquemas de calidad aplicables a productos de software genéricos. Se definen tres elementos: i) una ontología de esquema de calidad (quality scheme ontology, QSO) que establece el marco de referencia sobre el cual se generan los esquemas, ii) un conjunto de reglas SWRL (W3C 2004) y de consultas SPARQL (W3C 2013) que permiten analizar el grado de cobertura que posee un subconjunto de datos en relación al total requerido para la estimación de la calidad, y iii) una actividad que indica cómo utilizar los otros dos elementos. De esta manera, los esquemas creados en base a la ontología, permiten derivar conocimiento relacionado a la información de calidad requerida.

5.1 Ontología QSO: "Quality Scheme Ontology"

La ontología QSO ha sido definida como modelo base para la especificación de esquemas de calidad. Corresponde a una ontología de dominio, debido a que ha sido formulada como un modelo aplicable a un ámbito definido en base a un punto de vista específico (Roussey y colab. 2011). Su diseño se compone de tres modelos semánticos, a saber: un *modelo semántico de calidad de producto de software*, un *modelo semántico de métricas de producto de software* y un *modelo semántico de producto de software*. Cada uno de estos modelos corresponde a un dominio identificado previamente como componente de un esquema de calidad (ver apartado *"Componentes de un Esquema de Calidad"*).

Aunque muchos autores han propuesto ontologías relacionadas con el dominio de calidad, los modelos resultantes no reflejan los avances que han tenido lugar como resultado de la formulación de las normas de la serie SQuaRE. Por este motivo, se diseñaron modelos semánticos específicos para cada elemento requerido. En todos los casos, la metodología de desarrollo utilizada fue una adaptación del Método 101 (Noy y McGuinness 2001) en combinación con lo propuesto en (Grüninger y Fox 1995), quedando definida básicamente en 5 etapas, a saber:

1. *Determinar el dominio*: Se establecieron los dominios de los modelos a desarrollar junto con su aplicación. El objetivo de la ontología final es la definición de un marco de referencia para la construcción de esquemas de calidad. Por este motivo (tal como se ha mencionado con anterioridad), los modelos semánticos que forman la ontología QSO fueron desarrollados en base a las descripciones de los dominios asociados a la composición de los esquemas de calidad.
2. *Determinar el alcance*: A fin de obtener información acerca de la composición de un esquema de calidad, se definieron *preguntas de competencia*. Una *pregunta de competencia* es una pregunta que la ontología debe ser capaz de

responder (Grüninger y Fox 1995). Estas preguntas sirven como prueba de control de calidad, ya que permiten determinar si la ontología resultante contiene suficiente información como para responder a todas las preguntas planteadas de la forma esperada. La Tabla 5.1 resume las preguntas de competencia formuladas para la ontología QSO. Como puede observarse, las preguntas han sido clasificadas de acuerdo al tipo de respuesta esperada. Las categorías utilizadas en esta clasificación corresponden a *métrica*, *característica de calidad / subcaracterística de calidad / esquema de calidad* y *artefacto / programa de computadora*.

CATEGORÍA	**ID**	**PREGUNTA DE COMPETENCIA**
Métrica *(metric)*	Q1	¿Qué métricas son útiles para evaluar la característica de calidad 'X'? *(Which metrics are useful to evaluate the 'X' quality characteristic?)*
	Q2	¿Qué métricas son útiles para evaluar la subcaracterística de calidad 'Y'? *(Which metrics are useful to evaluate the 'Y' quality subcharacteristic?)*
	Q3	¿Cuántas métricas distintas se asocian a la característica de calidad 'X'? *(How many different metrics are related to the 'X' quality characteristic?)*
	Q4	¿Cuántas métricas distintas se asocian a la subcaracterística de calidad 'Y'? *(How many different metrics are related to the 'Y' quality subcharacteristic?)*
Característica / Subcaracterística / Esquema *(characteristic / subcharacteristic / quality scheme)*	Q5	¿A cuál característica de calidad se encuentra asociada la métrica 'Z'? *(To which quality characteristic is associated the 'Z' metric?)*
	Q6	¿Cuántas veces se asocia la métrica 'Z' con la característica de calidad 'X'? *(How many times are related the 'Z' metric with the 'X' quality characteristic?)*

	Q7	¿A qué característica de calidad es usualmente asociada la métrica 'Z'? *(To which quality characteristic is more often associated the 'Z' metric?)*
	Q8	¿A cuál subcaracterística de calidad se encuentra asociada la métrica 'Z'? *(To which quality subcharacteristic is associated the 'Z' metric?)*
	Q9	¿Cuántas veces se asocia la métrica 'Z' con la subcaracterística de calidad 'Y'? *(How many times are related the 'Z' metric with the 'Y' quality subcharacteristic?)*
	Q10	¿A qué subcaracterística de calidad es usualmente asociada la métrica 'Z'? *(To which quality subcharacteristic is more often associated the 'Z' metric?)*
	Q11	¿A qué esquema de calidad se asocia el artefacto 'V' del producto de software 'W'? *(Which is the quality scheme related to the 'V' artifact of the 'W' software product?)*
	Q12	¿Cuál es el esquema de calidad asociado al producto de software 'W'? *(Which is the quality scheme related to the 'W' software product?)*
Artefacto / Programa *(artifact / computer program)*	Q13	¿A qué artefacto se asocia la métrica 'Z'? *(To which artifact is associated the 'Z' metric?)*
	Q14	¿A qué programa de computadora se asocia la métrica 'Z'? *(To which computer program has been associated the 'Z' metric?)*
	Q15	¿Cuántos artefactos se asocian a la métrica 'Z'? *(How many artifacts have been associated the 'Z' metric?)*
	Q16	¿Cuántos programas de computadora se asocian a la métrica 'Z'? *(How many computer programs have been associated the 'Z' metric?)*

Tabla 5.1. Preguntas de competencia formuladas para la ontología QSO.

3. *Enumeración de términos importantes y definición de conceptos*: En base a las descripciones de los dominios asociados a cada componente, se realizaron transformaciones directas entre cada uno de los conceptos identificados como parte de la terminología y los conceptos del modelo semántico propuesto.

4. *Definición de propiedades de conceptos*: Se establecieron atributos y relaciones asociadas a cada uno de los conceptos identificados en la etapa previa. Las dependencias entre conceptos se incorporaron a los modelos bajo la forma de relaciones, utilizando un nombre descriptivo a fin de indicar la forma en la cual se vinculan los elementos (por ejemplo, “es-dividido-en”).
5. *Definición de aspectos de propiedades*: Se detallaron para cada una de las propiedades identificadas en la etapa previa, las características o aspectos a considerar en su definición (como ser tipo de dato, dominio y rango, entre otras).

Cada uno de los modelos detallados incorpora un conjunto de reglas SWRL (W3C 2004) a fin de contribuir a su correcta instanciación. Tal como se ha detallado en el apartado *"Semantic Web Rule Language (SWRL)"*, este lenguaje es utilizado para definir reglas bajo el formato de implicancia lógica (permitiendo derivar nueva información a partir de un conjunto de instancias ya definidas). En el caso del diseño de la ontología QSO, este lenguaje es utilizado para formular reglas que permiten mantener la consistencia en relación a los vínculos requeridos como parte de la definición de los conceptos.

Es importante destacar que los modelos semánticos propuestos fueron formulados en inglés a fin de mantener una directa compatibilidad con el material base usando como fundamento de la ontología. Sin embargo, con el objetivo de ampliar la utilidad de la ontología resultante, se asociaron etiquetas en español a cada uno de los elementos.

En cuanto a la representación gráfica utilizada para describir los elementos intervinientes en cada uno de los modelos, el formato elegido es similar al propuesto en (Gómez-Pérez, Fernandez-Lopez y Corcho 2010); en el cual:

- Los nodos grises representan conceptos.
- Los nodos blancos representan relaciones entre conceptos.

- Las cajas grises representan tipos de datos asociados a los atributos.
- El sentido de las flechas indica la dirección de las relaciones.
- La flecha con punta triangular sin relleno indica subsunción, disjunta y completa.

5.1.1 Diseño de los Modelos Semánticos

Modelo Semántico 1: Modelo de Calidad de Producto de Software

Considerando que: *i)* se ha establecido con anterioridad la utilidad de los modelos de calidad como mecanismo de soporte para la definición del contexto de referencia asociado a los esquemas de calidad y, *ii)* el modelo de calidad de producto de software definido en el estándar ISO/IEC 25010 representa los principales aspectos de calidad esperados en los productos de software (siendo el modelo más completo y actual que se encuentra en la literatura); el modelo semántico que define las propiedades de calidad como parte de la ontología QSO se basa en los conceptos definidos en dicho estándar. En este sentido, el modelo desarrollado enriquece la definición original, ya que evidencia la forma en la cual se relacionan las distintas características y subcaracterísticas posibilitando la detección de relaciones implícitas entre conceptos.

La Tabla 5.2 resume el conjunto de transformaciones realizadas sobre los elementos del dominio (es decir, los conceptos definidos en la Figura 4.3) a fin de obtener los componentes del modelo semántico. Cada una de las características y subcaracterísticas identificadas como parte del modelo de calidad fue transformada en un concepto de la ontología (T1 y T2, respectivamente). Teniendo en cuenta que estos conceptos representan una jerarquía de elementos, las relaciones entre ellos se modelaron haciendo uso del tipo de relación "es-descompuesto-en" (*is-decomposed-in*). Tómese como ejemplo la definición de la característica *Functional Suitability*. En este caso, el modelo de calidad de producto define que esta característica se relaciona con las subcaracterísticas *Functional Appropriateness*, *Functional Completeness* y *Functional Correctness*. En virtud de representar esta dependencia como parte del modelo

semántico, el concepto *Functional Suitability* se vincula con los conceptos *Functional Appropriateness*, *Functional Completeness* y *Functional Correctness* por medio de las relaciones *is-decomposed-in-functional-completeness*, *is-decomposed-in-functional-correctness* e *is-decomposed-in-functional-appropriateness*, respectivamente (T3).

Con el objetivo de mejorar la especificación inicial del modelo a fin de unificar los tipos de elementos previamente identificados, se incorporaron dos conceptos: *Characteristic* y *Subcharacteristic* (T4). Luego, cada uno de los conceptos existentes fue asociado por medio de una relación "es-un" (*is-a*) al tipo de elemento correspondiente (T5). Esta definición dio como resultado la clasificación de los elementos en las distintas categorías propuestas por el estándar. Por ejemplo, el concepto *Functional Suitability* se vincula por medio de la relación *is-a* con el concepto *Characteristic*. En este sentido, el concepto *Quality Model* también fue incorporado como parte del modelo (T6). Dado que el modelo de calidad bajo análisis se compone de un conjunto específico de características, se definieron relaciones del tipo "contiene" (*contains)* a fin de vincular cada característica componente con la definición del modelo de calidad (T7). Tomando como ejemplo los conceptos *Functional Suitability* y *Quality Model*, se define la relación *contains-functional-suitability* para vincular ambos elementos.

Una vez detallados todos los conceptos necesarios para representar los elementos propuestos en el modelo de calidad del estándar ISO/IEC 25010, se incorporaron atributos que refieren a propiedades específicas de los conceptos (T8). Estas propiedades incluyen, entre otras, las descripciones de las características y subcaracterísticas (propiedades *characteristic description* y *subcharacteristic description* de los conceptos *Characteristic* y *Subcharacteristic*, respectivamente) haciendo uso de cadenas de caracteres (tipo de dato *string*).

ID	ISO/IEC 25010	ONTOLOGÍA	EJEMPLO
T1	Tipos de característica	Concepto	*La característica "Compatibility" se mapea al concepto "Compatibility"*
T2	Tipos de subcaracterística	Concepto	*La subcaracterística "Interoperability" se mapea al concepto "Interoperability"*
T3	Descomposición 'característica-subcaracterística'	Relación	*La relación entre "Compatibility" e "Interoperability" se traduce a la relación "is-decomposed-in-interoperability"*
T4	Definición de característica y subcaracterística de calidad	Concepto	*El término "Characteristic" se define como el concepto "Characteristic"*
T5	Herencias características y subcaracterísticas	Relación	*La dependencia de conceptos entre "Characteristic" y "Compatibility" se mapea a la relación "is-a"*
T6	Modelo de calidad	Concepto	*El término "Quality Model" se define como el concepto "Quality Model"*
T7	Descomposición 'modelo de calidad-características de calidad'	Relación	*La dependencia entre "Quality Model" y "Compatibility" se mapea en la relación "contains-compatibility"*
T8	Origen y descripción de característica/subcaracterística	Propiedad	*El atributo "description" de una característica se mapea a la propiedad "description" de "Characteristic"*

Tabla 5.2. Mapeos de los elementos del estándar ISO/IEC 25010 a la ontología QSO.

Además de los conceptos y relaciones detallados, el modelo incluye un conjunto de reglas SWRL para la especificación de un nuevo tipo de relación denominado "contiene" (*contains*). De acuerdo con T1, T2 y T3 (Tabla 5.2), las relaciones entre los conceptos que representan características y subcaracterísticas queda definida en base al tipo de relación "es-descompuesto-en". Bajo esta definición, toda característica queda descompuesta en sus subcaracterísticas (sin importar el caso sobre el cual se esté trabajando). Luego, es posible establecer una nueva relación que aplique a todos las descomposiciones por igual, a fin de obtener (en una etapa posterior) información

asociada a los casos generales. Esta nueva relación se denomina “contiene”, quedando definida de la siguiente manera:

“Sea X una característica de calidad e Y una subcaracterística de calidad asociada a X. Luego, X contiene a Y”.

La Ecuación 5.1 presenta a modo de ejemplo la regla SWRL detallada para la relación “contiene” en el caso de la subcaracterística “Integridad” (*Integrity*) asociada a la característica “Seguridad” (*Security*). En este ejemplo, las variables “?x” e “?y” refieren a individuos (es decir, instancias de conceptos), mientras que los términos "Security()" e "isDecomposedInIntegrity()" representan el concepto asociado a la característica seguridad y la relación del tipo “es-descompuesto-en” que vincula ambos individuos, respectivamente. De esta manera, la regla garantiza que cualquier instancia del concepto “Security()” que se encuentre asociada a una instancia diferente por medio la relación “isDecomposedInIntegrity()”, se encontrará también vinculada por medio de la relación “contains()”. Restricciones similares se incorporaron a fin de reflejar las asociaciones entre el conjunto de características del modelo de calidad y las subcaracterísticas asociadas.

$$\text{Security}(?x) \wedge \text{isDecomposedInIntegrity}(?x,?y) \rightarrow \text{contains}(?x,?y) \qquad (5.1)$$

En virtud de explicitar el caso inverso (es decir, dada una subcaracterística de calidad obtener la característica a la cual se asocia), el modelo incorpora la relación “pertenece” (*belongs*). Esta relación es inversa a la relación “contiene”, quedando definida como:

“Sea X una característica que contiene a la subcaracterística Y. Luego, la subcaracterística Y pertenece a la característica X”.

La Ecuación 5.2 muestra la regla SWRL formulada para definir esta nueva relación en base a los elementos existentes. Como puede observase, dadas dos instancias "?x" e "?y" vinculadas por medio de una relación "contains()", se infiere la relación "belongs()".

$$\text{contains}(?x,?y) \rightarrow \text{belongs}(?y,?x) \qquad (5.2)$$

La Figura 5.1 muestra una representación gráfica del modelo semántico obtenido. Como puede observase, el modelo consta de 42 conceptos, 42 relaciones y 3 atributos; los cuales (junto con las reglas SWRL) permiten generar instancias de conceptos asociados a la definición del modelo de calidad de producto de software propuesto en ISO/IEC 25010, para luego derivar nuevas relaciones de alto nivel entre los mismos.

Modelo Semántico 2: Métrica de Producto de Software

Dada la definición de esquema de calidad y, teniendo en cuenta que se busca definir las métricas sin documentar su relevamiento, previo al diseño del modelo semántico se establecieron los límites del dominio a representar.

Es importante diferenciar que, en el contexto de métricas de calidad, existen dos tipos de métricas relevantes: *métricas referidas al producto final* y *métricas asociadas al proceso de desarrollo*. En el caso de los esquemas de calidad, se trabaja con métricas de producto. Estas métricas corresponden a medidas asociadas a alguna propiedad de una parte del código o de los artefactos detallados como parte del producto de software bajo desarrollo (El-Haik y Shaout 2010).

Aunque todos los conceptos detallados en el apartado *"Métricas de Software"* definen el dominio a modelar como parte de la ontología, sólo algunos de ellos son de utilidad en lo que respecta a los esquemas de calidad. En este sentido, con el objetivo de documentar las métricas referidas a distintos atributos de calidad a ser evaluados sobre un producto de software específico, los siguientes elementos son irrelevantes:

- *Entradas del proceso de medición*: Este elemento requiere conocer la etapa del proceso de desarrollo en la cual el relevamiento de la métrica tendrá lugar. Es de utilidad cuando la métrica se elabora para ser utilizada en un contexto específico, conociendo las fuentes de información disponibles, su tipo y cantidad. Dado que la actividad de documentación solamente busca representar la forma de cálculo (sin indicar expresamente el momento en el cual se realizará esta estimación), es difícil establecer la fuente de información en etapas tempranas del proceso de desarrollo y, por lo tanto, este elemento no se incluye como parte del modelo.
- *Grupo de interés*: Al igual que en el caso precedente, este elemento tiene una dependencia directa con el uso de la métrica. Es difícil establecer los grupos de interés que harán uso de una métrica en etapas tempranas del proceso de desarrollo. Comúnmente, ante este desconocimiento se tienen dos posibles escenarios: *i)* métricas vinculadas a ningún grupo de interés o, *ii)* grupos interés identificados como parte de una métrica que nunca harán uso de la misma. A fin de no documentar información ambigua, se omite este elemento como parte del modelo.

Una vez identificados los términos a representar en el modelo semántico, se planificaron dos etapas de diseño. En una primera etapa se establecieron los principales componentes requeridos para construir un modelo base. Tomando como referencia este modelo base, en la segunda etapa se refinaron los conceptos y propiedades previamente identificadas de acuerdo al nivel de detalle requerido en los esquemas de calidad.

En la Figura 5.2 se presenta el modelo base formulado haciendo uso de los conceptos del estándar ISO 9126 (ISO/IEC TR 9126-2 2003; ISO/IEC TR 9126-3 2003), tomando como referencia el enfoque presentado en (García y colab. 2006). Tal como puede observase, la mayoría de los elementos definidos corresponden a la descripción de dominio detallada en el apartado *"Métricas de Software"*. Sin embargo, se evidencia

que la mayor parte de los elementos definidos se representa como atributos asociados a unos pocos conceptos. Sólo los términos de alta relevancia son modelados como conceptos, como ser "Métrica" (*Metric*), "Métrica Directa" (*Direct Metric*) y "Métrica Indirecta" (*Indirect Metric*), entre otros.

En virtud de mejorar la definición precedente (propuesta como modelo base), se mejoraron aspectos relacionados con:

i) La incorporación de términos faltantes haciendo uso de conceptos.
ii) La creación de nuevos conceptos que reemplazan atributos existentes.
iii) La especificación de clasificaciones para diferenciar elementos de un mismo tipo.

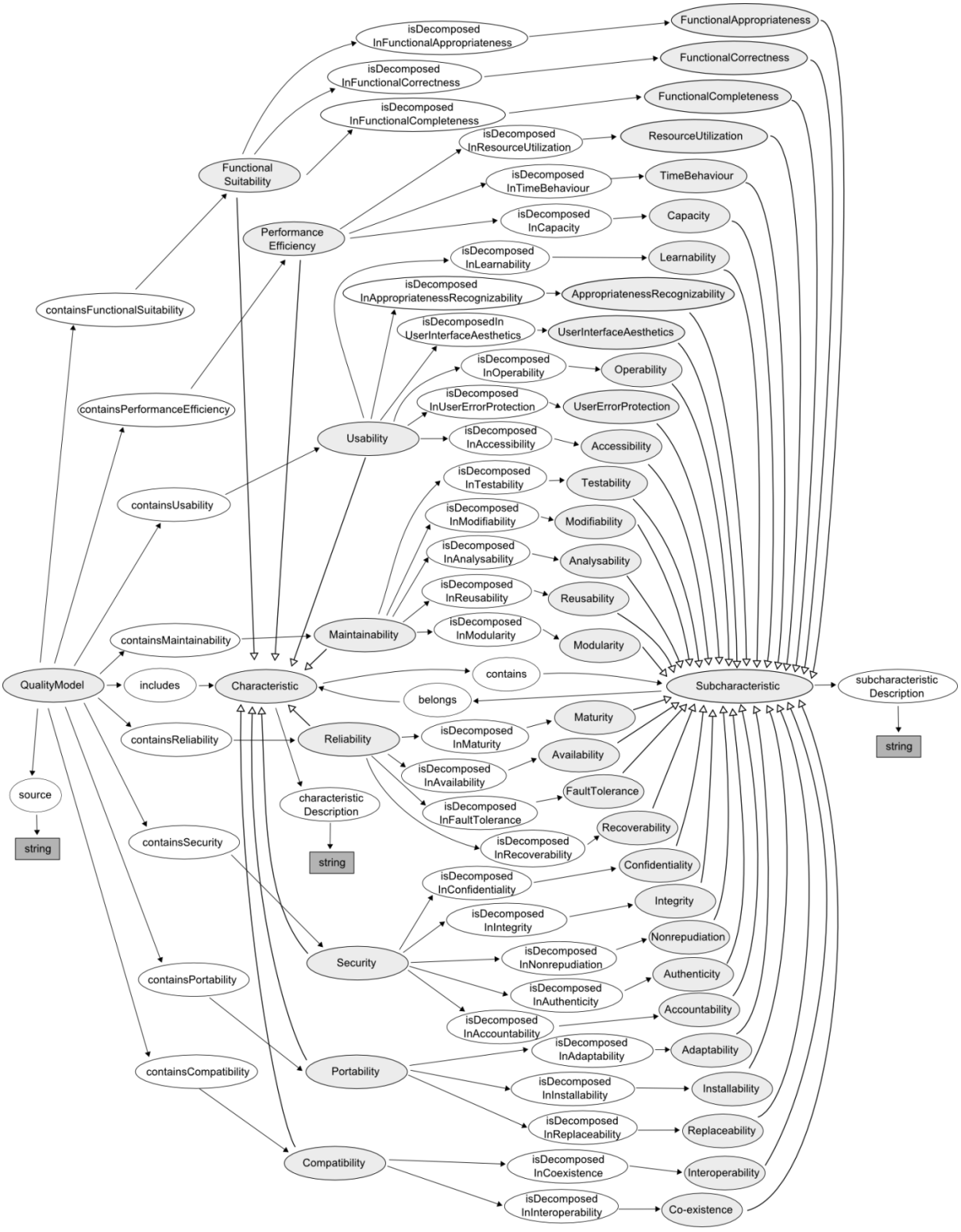


Figura 5.1. Modelo semántico de calidad de producto basado en ISO/IEC 25010.

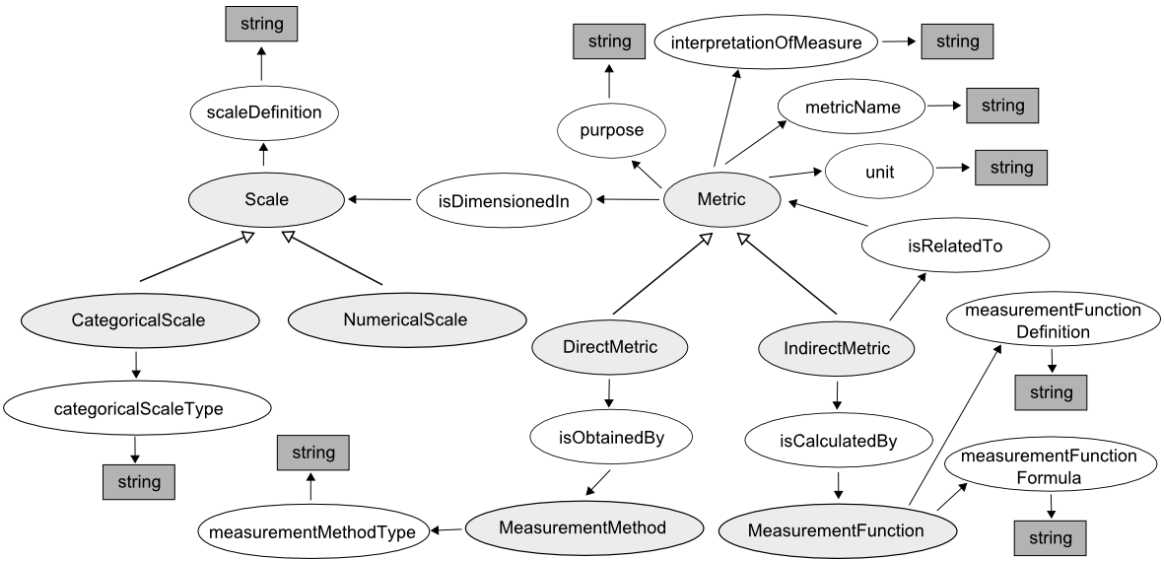


Figura 5.2. Modelo base del dominio de métricas de software.

El modelo resultante de estas adaptaciones se presenta en la Figura 5.3. Este diseño tiene una mayor cantidad de elementos ya que, según los lineamientos propuestos, las modificaciones realizadas conllevan a la conversión de elementos definidos conforme un tipo a su especificación como un tipo diferente, mejorando así su definición.

Siguiendo el lineamiento i), se incorporaron conceptos asociados a la definición de ecuaciones. Teniendo en cuenta que el concepto "Función de Medición" (*Measurement Function*) representa una fórmula matemática compuesta de términos, operadores y variables, el modelo semántico final incluye un conjunto de elementos que permite componer la especificación de la fórmula asociada a una métrica. Con este objetivo, el concepto "Ecuación" (*Equation*) fue incorporado. En este sentido, toda función de medición es una ecuación compuesta de distintos tipos de términos (representados por el concepto "Término" -*Term*-). Un término puede ser simple (concepto "Término Simple" –*Simple Term*-) o complejo (concepto "Término Complejo" –*Complex Term*-), según el tipo de operación matemática que tenga asociado. Sin embargo, también puede referir al

cálculo de una métrica (debido a que la formulación de métricas indirectas se basa en el uso de otras métricas). Por este motivo, el concepto "Métrica" se asocia como subtipo del concepto "Término". Como complemento de los conceptos detallados con anterioridad, el modelo final incluye una clasificación de las operaciones matemáticas comúnmente utilizadas en métricas de productos de software. Para esto, asocia un conjunto de elementos al concepto "Operación" (*Operation*), entre los que se destacan "Operación Simple" (*Simple Operation*) y "Operación Compleja" (*Complex Operation*). Los principales componentes identificados en (Castro, Rico y Castro 1995) fueron utilizados como modelo de referencia para refinar esta especificación.

Por otra parte, en relación con el lineamiento ii), se propuso modificar el atributo "unidad" (*unit*) por un conjunto de términos (y sus respectivas relaciones) asociado al concepto "Unidad" (*Unit*). La identificación de una unidad como atributo dentro del modelo semántico tiene múltiples inconvenientes, entre los que se destacan: *i)* permite asignar una unidad a una métrica con escala categórica y, *ii)* posibilita que una métrica adimensional se vincule con una unidad dada.

Con el objetivo de solucionar estos inconvenientes, la propuesta de (Rijgersberg, Wigham y Top 2011) fue tomada como referencia. En este sentido, se incorporaron los conceptos "Unidad Simple" (*Simple Unit*) y "Unidad Derivada" (*Derived Unit*). Una "Unidad Simple" representa una unidad básica (es decir, una unidad que únicamente puede ser obtenida por medio de su definición). Por su parte, una "Unidad Derivada" representa una unidad que se obtiene por medio de la operación entre otras unidades. Estas operaciones pueden ser simples o complejas. El concepto "Unidad Derivada por Operación Simple" (*Derived Unit By Simple Operation*) representa a aquellas unidades que se obtienen de las operaciones en las que interviene un único argumento. Esta categoría incluye las operaciones de potencia y raíz (conceptos "Unidad Derivada por Operación de Potencia" –*Derived Unit By Power Operation*- y "Unidad Derivada por Operación de Raíz" –*Derived Unit By Root Operation*- respectivamente). Aunque desde un punto de vista matemático ambas operaciones son binarias, en el contexto de métricas

de producto de software, es usual que estas operaciones se utilicen con el objetivo de alterar una única variable por un factor numérico.

Tómese como ejemplo la métrica propuesta en (Card y Glass 1990) para relevar la complejidad estructural de un módulo de software. Esta métrica queda definida como $S(i) = (f_{out}(i))^2$, donde $f_{out}(i)$ corresponde al número de entradas que pueden ser conectadas a un punto específico del módulo *i*. El valor de $f_{out}(i)$ es obtenido por medio del cálculo de otra métrica, la cual se encuentra definida conforme sus propias ecuaciones y unidades. En este ejemplo, se observa que la potencia (en este caso una base variable afectada por un exponente dos) aplica a un único término obtenido previamente por un cálculo independiente (realizado al nivel de otra métrica). Luego, aunque al colocar la potencia y la raíz como operaciones simples se simplifica el modelo resultante, se facilita la determinación de las unidades para los casos a ser representados por medio de la ontología como parte de un esquema de calidad.

En contraposición al concepto "Unidad Derivada por Operación Simple", se utiliza el concepto "Unidad Derivada por Operación Compleja" (*Derived Unit By Complex Operation*) con el objetivo de representar a aquellas unidades que se obtienen de operaciones en las que intervienen dos argumentos. Tales operaciones involucran la suma, resta, multiplicación y división; las cuales quedan representadas por los conceptos "Unidad Derivada por Suma" (*Derived Unit By Addition*), "Unidad Derivada por Resta" (*Derived Unit By Subtraction*), "Unidad Derivada por Multiplicación" (*Derived Unit By Multiplication*) y "Unidad Derivada por División" (*Derived Unit By Division*), respectivamente. Teniendo en cuenta que una unidad adimensional solo puede darse en el caso de una operación de división, el concepto "Unidad Adimensional" (*Adimensional Unit*) fue detallado como caso específico del término "Unidad Derivada por División".

Además de la incorporación de los conceptos descriptos, se especificaron dos relaciones que permiten vincular los nuevos elementos con el resto de los

componentes de la ontología. Estas relaciones se denominan "es-medido-en" (*is-measured-in*) y "tiene-como-unidad" (*has-as-unit*). La primera se utiliza para expresar la necesidad de tener una unidad asociada a las escalas numéricas, mientras que la segunda garantiza que si una métrica está asociada a una escala numérica, entonces su unidad de medición será la misma que la que se encuentra detallada en la escala. Esta última relación es una relación derivada, por lo que se obtiene por medio de una regla SWRL (la cual es detallada más adelante en este aparado).

Finalmente, en lo referente al lineamiento iii), una reformulación de la clasificación original del concepto "Escala" (*Scale*) tuvo lugar. El enfoque propuesto en (Olsina y Martín 2003) fue utilizado como base para esta reformulación. Se incorporaron conceptos y atributos, incluyendo los términos "Escala Numérica Discreta" (*Discrete Numerical Scale*) y "Escala Numérica Continua" (*Continuous Numerical Scale*) para identificar distintos tipos de escalas numéricas. En relación a este último elemento, se añadieron conceptos vinculados a la definición de rangos, entre los que se destacan "Rango" (*Range*), "Rango Mínimo" (*Min Range*), "Rango Máximo" (*Max Range*) y "Rango Mínimo-Máximo (*Min Max Range*). Estos elementos facilitan la representación de escalas definidas en intervalos numéricos específicos (ya sea con límites inferiores, superiores o ambos).

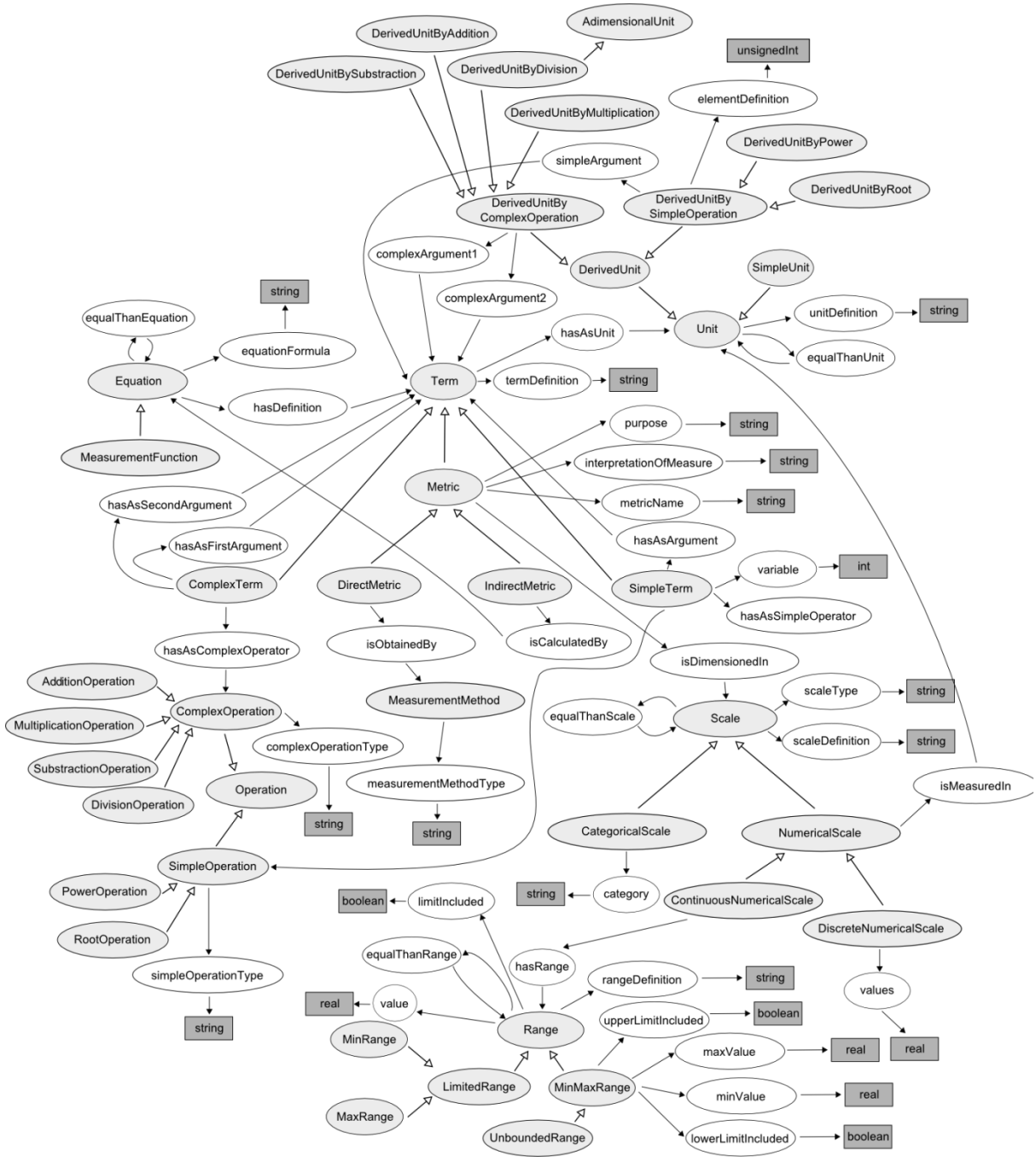


Figura 5.3. Modelo semántico completo para métricas de software.

Al igual que en el modelo semántico de calidad de producto de software, los conceptos, relaciones y propiedades definidas como parte de la ontología se complementaron con reglas SWRL. Estas reglas se utilizaron para derivar nuevas

relaciones de valor para el estudio del modelo, a partir de los individuos instanciados en base a los elementos incluidos como parte del modelo semántico.

La Ecuación 5.3 presenta la regla especificada para la relación "tiene-como-unidad", la cual ha sido descripta con anterioridad. En este caso, "?m" refiere a una instancia del concepto "Metric()" mientras que "?s" y "?u" refieren a instancias de los conceptos "Scale()" y "Unit()" respectivamente. Por su parte, los términos "isDimensionedIn()" e "isMeasuredIn()" se vinculan con las relaciones de dimensionamiento y medición definidas entre las instancias correspondientes de métrica, escala y unidad. Luego, si la métrica "?m" y la escala numérica "?s" se asocian por medio de una relación "es-dimensionado-en", al mismo tiempo que la escala "?s" y la unidad "?u" se vinculan por una relación "es-medido-en", se infiere una nueva relación "tiene-como-unidad" que afecta a la métrica "?m" y a la unidad "?u".

$$\text{Metric}(?m) \wedge \text{NumericalScale}(?s) \wedge \text{isDimensionedIn}(?m,?s) \wedge \text{isMeasuredIn}(?s,?u) \wedge \text{Unit}(?u) \rightarrow \text{hasAsUnit}(?m,?u) \quad (5.3)$$

Además de la regla enunciada, se explicitaron reglas SWRL para indicar relaciones de igualdad. Tales reglas buscan definir vínculos del tipo "es-igual-que" (*equal-than*), aplicándose a los conceptos de "Unidad", "Escala", "Ecuación" y "Rango". Tienen la particularidad de ser reflexivas, simétricas y transitivas, siendo derivadas por medio de la comparación de los atributos asociados a las distintas instancias. De esta manera, las reglas de igualdad no sólo facilitan la creación de individuos consistentes sino que, además, posibilitan la realización de comparaciones entre distintos individuos a fin de determinar si refieren a un mismo elemento.

A modo de ejemplo, las ecuaciones 5.4 y 5.5 presentan las reglas SWRL formuladas, respectivamente, para las relaciones derivadas "es-igual-que-rango" (*equal-*

than-range) y "es-igual-que-ecuación" (*equal-than-equation*). En este sentido, estas relaciones quedan definidas de la siguiente manera:

"Sean R1 y R2 dos rangos definidos en base a las descripciones D1 y D2 respectivamente. Considere que D1 es igual a D2. Luego, R1 es igual a R2."

"Sean E1 y E2 dos ecuaciones definidos en base a las fórmulas F1 y F2 respectivamente. Considere que F1 es igual a F2. Luego, E1 es igual a E2."

En el caso de la Ecuación 5.4, los identificadores "?r1", "?r2", "?d1" y "?d2" refieren a individuos (es decir, instancias de conceptos). Lo mismo ocurre con "?e1", "?e2", "?f1" y "?f2" en la Ecuación 5.5. Los predicados "Range()" y "Equation()" representan los conceptos "Rango" y "Ecuación" respectivamente. Los elementos "rangeDefinition()" y "equationFormula()" formalizan los vínculos de igual nombre que han sido indicados como parte del modelo semántico representado en la Figura 5.3, a fin de establecer las asociaciones requeridas entre conceptos. De esta forma, quedan definidas las reglas SWRL que representan las relaciones derivadas "equalThanRange()" y "equalThatEquation()".

$$\text{Range}(?r1) \wedge \text{rangeDefinition}(?r1,?d1) \wedge \text{Range}(?r2) \wedge \text{rangeDefinition}(?r2,?d2) \wedge \text{equals}(?d1,?d2) \rightarrow \text{equalThanRange}(?r1,?r2) \quad (5.4)$$

$$\text{Equation}(?e1) \wedge \text{equationFormula}(?e1,?f1) \wedge \text{Equation}(?e2) \wedge \text{equationFormula}(?e2,?f2) \wedge \text{equals}(?f1,?f2) \rightarrow \text{equalThanEquation}(?e1,?e2) \quad (5.5)$$

El resto de las relaciones del tipo "es-igual-que" se formularon en reglas SWRL similares a las expuestas en los ejemplos precedentes.

El modelo resultante consta de 40 conceptos, 19 relaciones y 22 atributos que, junto con las reglas SWRL, contribuyen a la creación de instancias de conceptos que ayudan a documentar el dominio de métricas de software.

Modelo Semántico 3: Producto de Software

El principal objetivo de este modelo semántico es la especificación del dominio de productos y sistemas de software, en virtud de identificar los componentes sobre los cuales se evaluará la calidad. Teniendo en cuenta este objetivo, el modelo desarrollado no es exhaustivo. Es decir, del conjunto de todos los elementos posibles del dominio bajo análisis, sólo se incorporaron los mínimos requeridos para representar la estructura básica de un producto de software. Esto se debe a que la estructura completa de un producto de software es compleja. Si se representa la totalidad de la estructura como parte de un esquema de calidad, se complejiza la definición del documento (debido a que quien lo especifica debe conocer de antemano todos los posibles componentes del producto). A fin de facilitar la especificación de los esquemas de calidad, se simplifica la representación de producto incorporando los elementos más significativos del dominio (descriptos en el apartado *"Producto de Software"*).

La Figura 5.4 visualiza el modelo semántico resultante. Como puede observase, los términos producto de software, programa de computadora, artefacto, entidad y atributo han sido representados como conceptos (respectivamente “Producto de Software” *-Software Product-*, “Programa de Computadora” *-Computer Program-*, “Artefacto” *-Artifact-*, “Entidad” *-Entity-* y “Atributo” *-Attribute-*). Los vínculos entre estos términos se modelaron como relaciones utilizando un nombre descriptivo. Por ejemplo, la relación “es-dividido-en” (*is-divided-in)* definida entre los conceptos “Artefacto” y “Entidad”, indica que un artefacto se divide en múltiples entidades. Además, a fin de poder individualizar instancias de forma representativa, se incorporaron atributos del tipo “nombre” (*name*) y “tipo” (*type*) a varios de los conceptos definidos. Tomando

nuevamente como ejemplo el concepto "Atributo", se tienen las propiedades descriptivas "nombre de atributo" (*attribute name*) y "tipo de atributo" (*attribute type*).

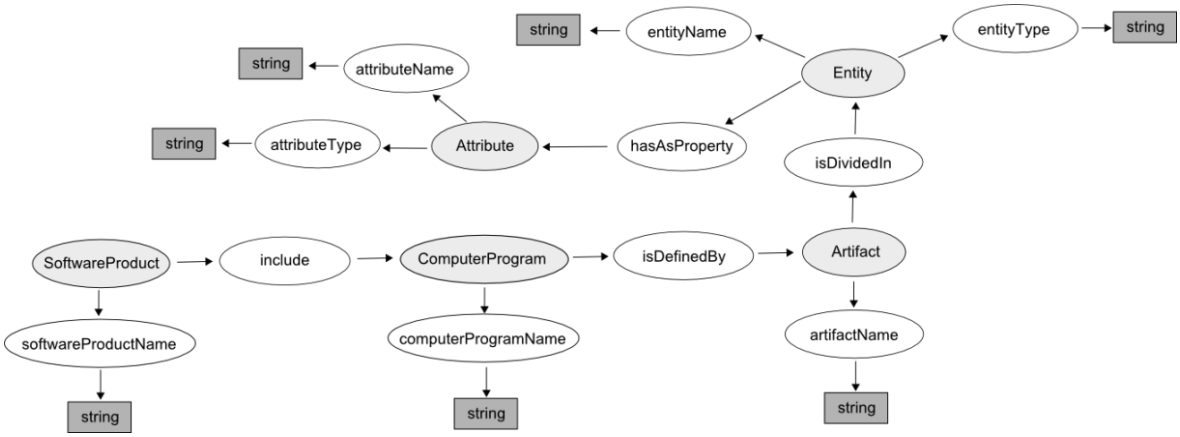


Figura 5.4. Modelo semántico de producto de software.

El modelo final consta de 5 conceptos, 4 relaciones y 7 atributos que permiten documentar los principales componentes de un producto de software. Dada la simplicidad del modelo, no fue necesario incluir reglas SWRL para derivar nuevo conocimiento.

5.1.2 Integración de los Modelos Semánticos (Ontología QSO)

El diseño de la ontología QSO se basa en las interacciones definidas para los modelos semánticos diseñados con el objetivo de representar cada uno de los componentes de un esquema de calidad. Cada dominio identificado detalla una parte importante de un esquema de calidad pero, en su conjunto, requieren de la especificación de nuevas relaciones que posibiliten la interacción de forma adecuada. En este sentido, es necesario unificar todos los dominios en un único modelo.

De acuerdo con (ISO/IEC 25001 2014), una *entidad de software* es un objeto que puede caracterizarse por medio de la medición de sus *atributos*. Siguiendo esta

definición, un *atributo de software* puede ser interpretado como un elemento que puede ser medido. Si un atributo puede ser medido, entonces debe existir alguna *métrica* asociada al mismo que posibilite su evaluación. Si esta métrica es una *métrica de calidad*, debe referir a alguna *propiedad de calidad* definida (por ejemplo, una *subcaracterística de calidad* del *modelo de calidad del producto de software de ISO/IEC 25010*). Luego, se puede enunciar que:

"En un contexto de calidad, un atributo de software debe ser medido en base a una métrica asociada a una subcaracterística (propiedad) específica."

Tomando como referencia este enunciado, se establecieron tres relaciones que vinculan los principales conceptos de los modelos semánticos previamente descriptos a fin de reflejar esta interacción, a saber:

- La relación "es-medido-por" (*is-measured-by*) que vincula el concepto "Atributo" (del modelo semántico de producto de software) con el concepto "Métrica" (del modelo semántico de métricas de software).
- La relación "es-descripto-por" (*is-described-by*) que asocia el concepto "Atributo" (del modelo semántico de producto de software) con el concepto "Subcaracterística" (del modelo semántico de calidad de producto).
- La relación "es-evaluado-por" (*is-evaluated-by*) que enlaza el concepto "Producto de Software" (del modelo semántico de producto de software) con el concepto "Modelo de Calidad" (del modelo semántico de calidad de producto).

La Figura 5.5 esquematiza estas relaciones. Tal como puede observarse, la incorporación de estas relaciones como intermediarias entre los distintos modelos semánticos contribuye a unificar los dominios conforme la definición de esquema de calidad (ver *"Definición Esquema de Calidad"*). Es decir, establece los vínculos necesarios entre un atributo de software, una métrica de producto y una subcaracterística de calidad para documentar de forma exhaustiva el conjunto de elementos de calidad asociado a un producto de software.

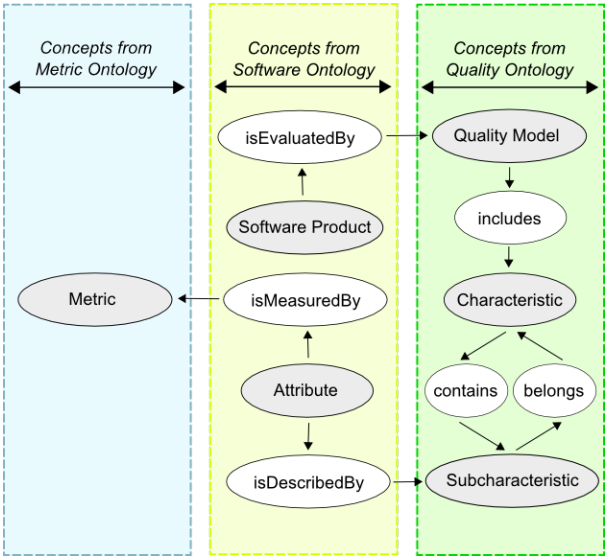


Figura 5.5. Relaciones definidas para vincular los modelos semánticos.

De esta forma, se define una única especificación de calidad en base a un conjunto de factores claramente identificados por medio del uso de modelos semánticos. La Ecuación 5.6 define formalmente los elementos que componen un esquema de calidad haciendo uso de los conceptos incluidos en la ontología QSO.

$$\text{Quality Scheme} = \{ \text{quality-element}_i \,/\, 1 \leq i \leq n \}$$

$$\text{donde } \textit{quality-element}_i = (m,a,s) \text{ con} \qquad (5.6)$$

$$m \in \textit{Metric}, a \in \textit{Software Attribute} \text{ y } s \in \textit{Subcharacteristic}.$$

Luego, para cada subcaracterística, la capacidad del producto de software queda determinada por el conjunto de atributos que pueden ser relevados por medio de métricas.

5.1.3 Implementación de la Ontología en Protégé

La ontología QSO fue implementada utilizando la herramienta Protégé[1]. Protégé es una herramienta que provee un entorno de trabajo que facilita el desarrollo de aplicaciones y la construcción de prototipos base de forma rápida y flexible (Knublauch y colab. 2005). Las ontologías implementadas en esta herramienta pueden exportarse a distintos formatos, incluyendo Web Ontology Language (OWL) y Resource Description Framework Scheme (RDF). Los modelos definidos en OWL 2 (W3C 2012) proveen clases, propiedades, individuos y datos que se intercambian como documentos RDF. La principal ventaja de utilizar modelos RDF por sobre otros esquemas de metadatos es que la información queda estructurada en grafos.

Cada uno de los modelos semánticos diseñados fue implementado en un archivo OWL independiente a fin de maximizar su posibilidad de reúso en contextos diferentes al de los esquemas de calidad (disponibles en *OWL - Ontologías Individuales*). De esta forma se garantiza la permanencia de los diseños originales. Los elementos incluidos en cada modelo se especificaron tanto en inglés como en español a fin de cumplir con lo definido en la fase de diseño. Además, como parte de la definición de conceptos, se incorporaron anotaciones que detallan breves descripciones de los objetivos asociados a cada uno de los elementos definidos. Las reglas SWRL también fueron implementadas como parte de cada documento.

Una vez implementados los tres modelos semánticos, la ontología QSO fue formulada importando los archivos individuales en un nuevo documento OWL (Figura 5.6). Este nuevo documento incorpora la definición de las tres relaciones que actúan como vínculos entre los conceptos de las ontologías (definidas en la Figura 5.5).

En la Figura 5.7 se visualiza una captura de pantalla del documento final implementado en Protégé (el cual se encuentra disponible en *OWL - QSO*). La clase

[1] Disponible en *http://protege.stanford.edu/*

seleccionada corresponde al concepto "Atributo" (elemento sombreado en el panel izquierdo), el cual tiene asociado propiedades referidas a su descripción y etiqueta en español (panel superior-derecho). Además, el concepto se vincula a axiomas que indican sus relaciones con el resto de los elementos (panel inferior-derecho). Bajo el axioma "equivalente-a" (*equivalent-to*) se incluyen las relaciones referidas a la clase "Atributo" (ver figuras 5.3 y 5.4). Por ejemplo, los primeros dos elementos refieren a las relaciones "es-medido-por" y "es-descripto-por".

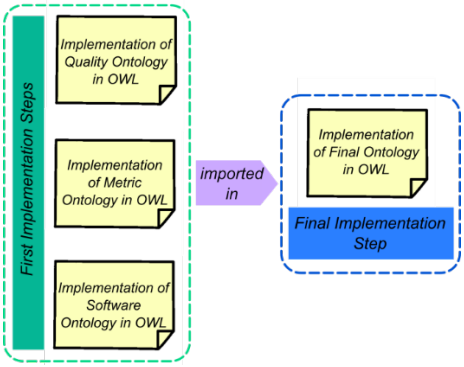


Figura 5.6. Integración de archivos OWL en la ontología final.

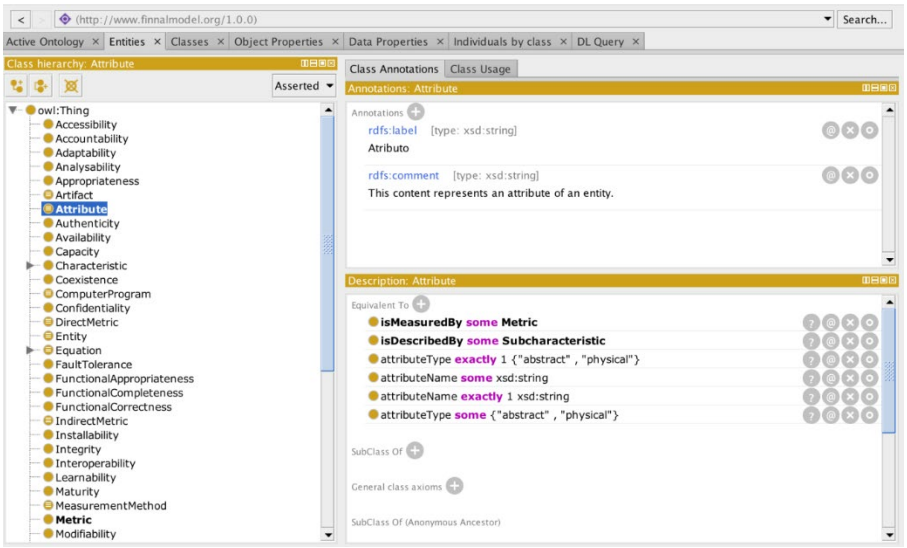


Figura 5.7. Captura de pantalla de la implementación de la ontología QSO.

5.1.4 Evaluación de la Ontología QSO

La fase de evaluación es importante dentro del proceso de diseño y desarrollo de ontologías debido a que permite detectar potenciales problemas derivados de la falta de experiencia de modelado. Esta etapa incluye las actividades de verificación y validación de la ontología bajo desarrollo. Mientras que la actividad de verificación refiere a la correcta implementación del modelo (es decir, que la ontología haya sido correctamente desarrollada), la actividad de validación apunta a que el modelo final cumpla con su objetivo (es decir, que brinde respuesta a las preguntas de competencia planteadas).

Actividad de Verificación: Detección y Resolución de Problemas de Modelado

La detección de problemas de modelado (junto con su resolución) se incluye como parte de la actividad de verificación. Su objetivo es garantizar la correctitud de la ontología desarrollada en lo referente a un conjunto de errores de modelado que se cometen con frecuencia durante la actividad de diseño.

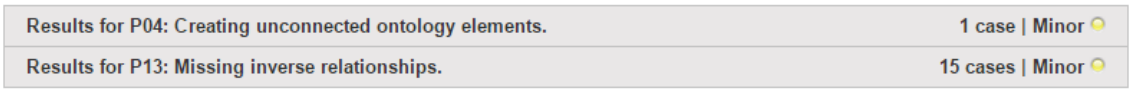
Una de las herramientas de verificación más comúnmente utilizadas es "OOPS!" (OntOlogy Pitfall Scanner[2]). Esta herramienta es una aplicación web que ayuda a detectar una gran variedad de problemas de modelado de ontologías. Su objetivo es brindar un soporte a los desarrolladores describiendo los problemas que se detectan en relación a una ontología específica, generando un reporte de problemas de modelado clasificados conforme un nivel de importancia (crítico, importante y menor). Un problema crítico es aquel que debe ser corregido para no afectar la consistencia, el razonamiento y la aplicación de la ontología. Un problema importante es un inconveniente que, aunque no es crítico para el funcionamiento de la ontología, es deseable que sea solucionado. Finalmente, un problema menor refiere a un aspecto

[2] Disponible en *http://oops.linkeddata.es/*

que no es vital que sea resuelto pero, en caso de ser solucionado, mejora el modelo de la ontología.

Para cada problema detectado, la herramienta muestra un ejemplo e indica la parte del modelo que lo causa. La actividad de resolución de los problemas detectados no forma parte del alcance de la aplicación. Esto quiere decir que quienes hayan diseñado e implementado la ontología deberán modificarla a fin de resolver los problemas detectados y, luego, volver a ejecutar la verificación.

Con el objetivo de verificar el modelo implementado de la ontología QSO y, teniendo en cuenta que el modelo final se encuentra compuesto de tres archivos independientes, la verificación fue ejecutada sobre cada uno de los archivos OWL desarrollados. La Figura 5.8 presenta los problemas detectados en los modelos semánticos de métrica (a), calidad (b) y software (c). Como puede observarse, todos los problemas detectados tienen un nivel de importancia menor.



(a) Modelo semántico de métrica de producto.

(b) Modelo semántico de calidad.

(c) Modelo semántico de producto de software.

Figura 5.8. Errores de modelado detectados en la ontología QSO.

En los tres casos, el problema de modelado "P13- Faltan relaciones inversas" (*missing inverse relationships*) fue detectado. Sin embargo, este error no es un verdadero problema debido a que la definición de relaciones inversas en una ontología se

relaciona con la naturaleza del dominio a ser representado. En el caso del dominio de métricas de software, las relaciones entre conceptos son unidireccionales y, por lo tanto, no se presenta la necesidad de incluir relaciones inversas como parte de la ontología. Lo mismo ocurre en el modelo de producto de software y en la mayoría de las relaciones definidas para el modelo de calidad. En este último modelo, sólo en algunos casos se incluyeron relaciones inversas a fin de garantizar la navegabilidad bidireccional entre conceptos. Por su parte, el problema "P04- Creación de elementos de la ontología no conectados" (*creating unconnected ontology elements*) identificado en el modelo semántico de métricas de producto, refiere a la especificación de componentes que no se vinculan con el resto de los conceptos y relaciones definidas. Puntualmente refiere al concepto "Operación", concepto desde el cual se define una jerarquía de operaciones específicas. Tales operaciones se vinculan directamente con las unidades y términos asociados, por lo que el concepto raíz no se encuentra relacionado con otros elementos. Sólo ha sido incluido en la ontología con el objetivo de abstraer los distintos tipos de operaciones disponibles. Luego, esta falta de conexiones (detectada por la herramienta) no representa un problema de modelado real.

Una vez analizados los archivos OWL correspondientes a cada uno de los modelos semánticos (habiendo corregido los problemas de modelado relevantes), la ontología final fue verificada haciendo uso de la misma aplicación. Sin embargo, debido a que el archivo OWL final importa las definiciones incluidas en los otros modelos, esta verificación sólo analiza la información definida como parte de la integración de conceptos (es decir, las relaciones definidas en el apartado *"Integración de los Modelos Semánticos (Ontología QSO)"*). Como resultado de este nuevo proceso de verificación, la falta de relaciones inversas fue nuevamente detectada. No obstante, al igual que en los casos precedentes, estas relaciones no son necesarias para cumplir con la definición de los esquemas de calidad.

Actividad de Validación: Especificación y Ejecución de Consultas SPARQL

De acuerdo a lo definido en el apartado *"SPARQL Protocol and RDF Query Language"*, el lenguaje SPARQL permite definir consultas para bases de datos semánticas a fin de recuperar información almacenada en formato RDF. Dado que los modelos OWL 2 pueden intercambiarse como documentos RDF, también puede aplicarse en este contexto.

A fin de validar la ontología implementada en OWL 2, las preguntas de competencia definidas en la Tabla 5.1 se especificaron como consultas SPARQL. Luego de poblar la ontología, se ejecutaron múltiples pruebas para garantizar que las consultas formuladas responden con la información esperada a cada pregunta de competencia plateada. Estas pruebas se llevaron a cabo en la herramienta Protégé.

A modo de ejemplo, la Figura 5.9 presenta la consulta asociada a la pregunta Q1 "¿qué métricas son útiles para evaluar la característica de calidad 'X'?". Como puede observarse, la consulta consta de tres cláusulas: *SELECT*, *WHERE* y *FILTER*. La cláusula SELECT define tres variables, detalladas como "?metricname", "?metricpurpose" y "?metricindividual". En base a estas variables, la consulta busca las tripletas válidas con la información obtenida desde la clausura WHERE. En este contexto, las variables representan el nombre, propósito e individuo asociado a cada una de las métricas vinculadas con la característica de calidad identificada como 'X'.

La búsqueda de las métricas queda restringida conforme lo especificado en la cláusula WHERE, donde se indica que:

- Deben buscarse todos los individuos del tipo 'X' con el valor especificado en la cláusula FILTER (línea 1).
- Haciendo uso de los individuos precedentes, deben buscarse todos los individuos que se vinculen al conjunto previamente identificado por medio de una relación "contiene" (línea 2). A este nivel, se obtienen todas las instancias

de subcaracterísticas de calidad asociadas a la característica de calidad que se encuentra bajo análisis.

- Del conjunto de subcaracterísticas encontradas, se buscan los individuos asociados por medio de la relación "es-descripto-por". Estos individuos refieren a instancias de atributos de software (línea 3).
- Con los atributos de software identificados, se buscan todos los individuos asociados a estos elementos mediante una relación "es-medido-por" (línea 4). El resultado será un conjunto de instancias de métricas. Estas métricas constituyen la respuesta a la pregunta de competencia planteada. Sin embargo, a fin de organizar los resultados obtenidos, para cada métrica encontrada se busca su nombre (línea 5) y propósito (línea 6).

```
SELECT DISTINCT ?metricname ?metricpurpose ?metricindividual
WHERE
{    ?characteristic rdf:type ?type .
     ?characteristic q:contains ?subcharacteristic .
     ?attribute sw:isDescribedBy ?subcharacteristic .
     ?attribute sw:isMeasuredBy ?metricindividual .
     ?metricindividual m:metricName ?metricname .
     ?metricindividual m:purpose ?metricpurpose.
     FILTER(regex(str(?type),"X"))
}
```

Figura 5.9. Pregunta de competencia Q1 especificada como consulta SPARQL.

Las consultas SPARQL restantes (formuladas para las preguntas de competencia Q2 a Q16) se especificaron siguiendo los mismos lineamientos que en el caso de Q1. En el *Apéndice A* se detallan cada una de las consultas especificadas.

Siguiendo el ejemplo del esquema de calidad de la juguetería online (propuesto en la Tabla 4.1), la Figura 5.10 visualiza el resultado de la ejecución de la consulta SPARQL asociada a la pregunta Q1 con 'X' = "Seguridad". Tal como puede observase, el resultado de la consulta (panel inferior) indica que las dos métricas asociadas a la característica "Seguridad" (*Security*) corresponden a "Cantidad de accesos" (*#Access*) y "Cantidad de accesos no autorizados" (*#Unauthorized Access*). Tales métricas coinciden

con las especificadas en la definición del esquema. Luego, la información brindada por la consulta es correcta.

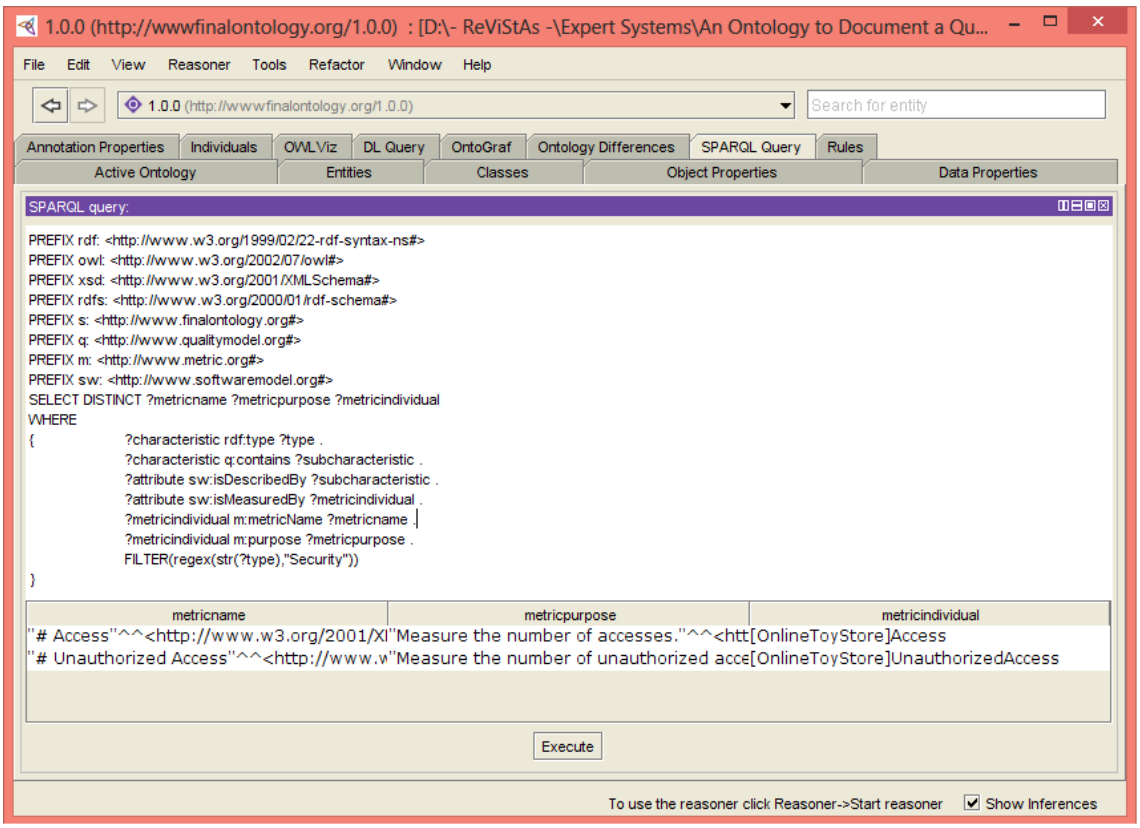


Figura 5.10. Ejecución de la consulta SPARQL asociada a la pregunta Q1.

Es importante destacar que puede ampliarse el conjunto de preguntas de competencia siempre y cuando la información incluida en la ontología posibilite reunir la información requerida para dar una respuesta correcta y coherente. En caso contrario, el modelo deberá ser modificado a fin de incluir el contenido faltante.

5.1.5 Calidad de la Ontología QSO

La evaluación de la calidad de una ontología de dominio es una tarea difícil debido a la falta de métodos formales para su estudio. Comúnmente, los trabajos relacionados a esta área se centran en el análisis de las dimensiones estructurales del modelo, estudiado propiedades tales como la cantidad de clases, cantidad de relaciones y

cantidad de instancias definidas. Este tipo de estudios brinda un marco de trabajo de utilidad que permite estimar la complejidad de los modelos semánticos haciendo uso de un conjunto de propiedades asociadas a su diseño. Sin embargo, es importante destacar que esta evaluación cuantitativa de la complejidad estructural contribuye a mejorar el desarrollo de los modelos pero no puede ser utilizada para probar las propiedades de completitud, coherencia y reusabilidad del modelo bajo análisis.

En este contexto, el conjunto de métricas estructurales utilizado por (Vegetti, Leone y Henning 2011) fue aplicado sobre el modelo final de la ontología QSO con el objetivo de evaluar su complejidad estructural. La Tabla 5.3 presenta las métricas utilizadas junto con los resultados obtenidos para cada una de ellas (columna "Valor").

Teniendo en cuenta que los dominios individuales poseen 42, 40 y 5 clases (para el modelo semántico de calidad, métrica y producto de software respectivamente), la cantidad total de clases definidas en la ontología QSO es 87. De estas 87 clases, 72 corresponden a clases vinculadas a relaciones de subsunción. En contraposición, las 15 clases restantes corresponden a conceptos que no poseen superclases. Por otra parte, la ontología contiene 140 relaciones divididas en dos grupos, a saber: *i)* relaciones de subsunción (72) y, *ii)* relaciones de asociación de conceptos (68). Estas últimas refieren a los vínculos definidos específicamente para representar asociaciones entre clases, quedando identificadas 42 relaciones en el modelo de calidad, 19 en el modelo de métrica, 4 en el modelo de producto de software y 3 en la integración de modelos. Finalmente, la cantidad de atributos incluidos en el modelo resultante de la integración es de 32 (3 para el dominio de calidad, 22 para el dominio de métricas y 7 para el dominio de producto de software).

Los valores obtenidos para cada una de las métricas evaluadas en la Tabla 5.3 parten del conjunto de propiedades descriptas con anterioridad. El valor de IR=0.827 junto con el de DOSH=5 permiten inferir que la ontología desarrollada tiene una naturaleza vertical, puesto que existe una gran proporción de subclases por clase pero

con un nivel de profundad considerable. Esto implica que el modelo representa conocimiento de un dominio específico, permitiendo generar instancias precisas de los esquemas de calidad. Esto se respalda en el hecho de que existe una proporción alta de atributos por clase (AR=0.367), lo que demuestra que, por medio del uso de estas características, se restringe el dominio al ámbito deseado. Un modelo que no contiene atributos es, en términos generales, un modelo más amplio que uno que si los tiene (debido a que no se requiere su identificación como parte de las instancias). Finalmente, el valor de RR=0.485 implica que la cantidad de relaciones jerárquicas es, por muy poco, más grande que la cantidad de relaciones de otro tipo. De esta forma, el modelo de la ontología QSO mantiene un balance adecuado entre las relaciones de herencia y las asociaciones.

DEFINICIÓN	ABREV.	DESCRIPCIÓN	VALOR
Cantidad de clases *(number of clases)*	NOC	Cantidad de clases.	87
Cantidad de relaciones *(number of relations)*	NOR	Cantidad de relaciones.	140
Cantidad de clases raíz *(number of root clases)*	NORC	Cantidad de clases sin superclases.	15
Cantidad de clases hoja *(number of leaf clases)*	NOLC	Cantidad de clases sin subclases.	72
Variedad de relaciones *(relationship richness)*	RR= |P|/(|H|+|P|)	Variedad de relaciones definidas en la ontología, donde: *|P| = cantidad de relaciones que no definen herencias y |H| = cantidad de relaciones que definen herencias.*	0.485 *|P|=68* *|H|=72*
Variedad de herencias *(inheritance richness)*	IR= |H|/|NOC|	Promedio de subclases por clase, donde: *|H| es el número de relaciones que definen herencias y |NOC| es el*	0.827 *|H|=72* *|NOC|=87*

		número total de clases.	
Profundidad de la jerarquía de herencias *(depth of the subsumption hierarchy)*	DOSH	Longitud del camino más largo, en una relación de herencia, entre una clase hoja y su clase raíz.	5
Variedad de atributos *(attribute richness)*	AR= |NAT|/|NOC|	Promedio de atributos por clase, donde: *|NAT | es la cantidad total de atributos y | NOC | es el número total de clases.*	0.367 *|NAT|=32* *|NOC|=87*

Tabla 5.3. Métricas aplicadas para evaluar la complejidad de la ontología QSO.

5.2 Análisis Información Requerida vs. Información Disponible

La ontología QSO define una estructura semántica que permite instanciar esquemas de calidad referidos a productos de software específicos. Sin embargo, una vez que se ha definido el documento de calidad en base a este modelo, es posible analizar el grado de cobertura que posee un subconjunto de información en relación al total de datos requeridos para la estimación de la calidad.

Para poder llevar a cabo este análisis, es necesario conocer el conjunto de datos requeridos para obtener valores asociados al esquema de calidad completo. Es decir, es necesario establecer la información mínima (necesaria y suficiente) que permita calcular la totalidad de las métricas propuestas. En este sentido, es importante comprender que los datos involucrados en el proceso de cálculo de las métricas definidas dentro de un esquema de calidad, no siempre serán independientes entre sí. Por ejemplo:

- Dos o más métricas pueden utilizar el mismo conjunto de datos pero, al momento de obtener un valor, operarlos de forma diferente.
- El cálculo de una métrica puede basarse en el resultado obtenido para otra métrica previamente especificada y calculada (es decir, definirse como una métrica indirecta), por lo que no toda la información requerida para su cálculo serán datos puros.

En este contexto, es posible definir tres conceptos: *información requerida*, *información derivada* e *información disponible*. La *información requerida* es el conjunto de datos puros que debe ser obtenido directamente del producto de software como resultado de un proceso de medición. La *información derivada* es el conjunto de datos que deber ser utilizado para calcular una métrica, pero que no corresponde a datos puros (es decir, que debe ser obtenido a partir del cálculo de otras métricas). La *información disponible* es el conjunto de datos puros que puede ser relevado en un momento dado sobre un producto de software específico como resultado de un proceso de medición.

(INFORMACIÓN REQUERIDA EN UN ESQUEMA DE CALIDAD) *Conjunto de datos puros que debe ser obtenido directamente del producto de software como resultado de un proceso de medición.*

La determinación de la información requerida en un esquema de calidad dado, garantiza el conocimiento de la totalidad de los datos puros necesarios para lograr un cálculo completo de las métricas propuestas (es decir, un 100% de cobertura del esquema de calidad). Una vez conocida la información requerida, es posible establecer el grado de cobertura del esquema de calidad en base a la información disponible. Este tipo de información es de utilidad para los desarrolladores ya que no sólo les permite conocer la información que debe relevarse como parte del proceso sino que, además, les permite comprender el impacto que tiene la falta de datos al momento de relevar la calidad (dando como consecuencia un alto grado de incertidumbre en relación al cumplimiento de los requisitos de calidad establecidos).

En virtud de proveer un marco de trabajo que permita realizar los análisis de cobertura propuestos en relación a la información requerida y disponible, se implementaron reglas SWRL y consultas SPARQL como complemento de la ontología QSO. Como parte de estos complementos, se definen nuevos elementos (conceptos y propiedades) que permiten vincular nuevo conocimiento con los elementos definidos en los modelos semánticos.

5.2.1 Identificación de Información Requerida y Marcado de Información Disponible

La Tabla 5.4 presenta el conjunto de reglas SWRL especificadas a fin de analizar tanto la información requerida como la información disponible sobre un esquema de calidad dado. Como puede observarse, es posible identificar dos tipos de reglas dentro del conjunto de reglas SWRL definido, a saber: *i)* reglas que contribuyen a identificar los datos puros que deben ser relevados sobre el producto de software bajo desarrollo a fin de poder evaluar las métricas de calidad propuestas como parte del esquema de calidad definido y, *ii)* reglas que ayudan a marcar la disponibilidad de la información requerida (es decir, transformarla en información disponible) a fin de estimar el porcentaje de requerimientos no funcionales referidos a las distintas características y subcaracterísticas del modelo de calidad que pueden ser evaluados haciendo uso del conjunto de datos existentes.

En virtud de garantizar un correcto análisis de cobertura sobre el esquema de calidad definido, se debe combinar la utilización de ambos tipos de reglas.

ID	REGLA SWRL
R1	DirectMetric(?x) → RequiredData(?x)
R2	RequiredData(?x) ∧ available(?x, true) → AvailableData(?x)
R3	DirectMetric(?x) ∧ AvailableData(?x) → CalculableTerm(?x)
R4	SimpleTerm(?s) ∧ CalculableTerm(?a) ∧ hasAsArgument(?s, ?a) → CalculableTerm(?s)

R5	ComplexTerm(?c) ∧ CalculableTerm(?a1) ∧ CalculableTerm(?a2) ∧ hasAsFirstArgument(?c, ?a1) ∧ hasAsSecondArgument(?c, ?a2) → CalculableTerm(?c)
R6	IndirectMetric(?m) ∧ MeasurementFunction(?f) ∧ CalculableTerm(?t) ∧ hasDefinition(?f, ?t) ∧ isCalculatedBy(?m, ?f) → CalculableTerm(?m)
R7	Metric(?m) ∧ CalculableTerm(?m) → CalculableMetric(?m)

Tabla 5.4. Reglas SWRL para derivar nuevo conocimiento en la ontología QSO.

Teniendo en cuenta que todas las métricas directas refieren a información que debe ser obtenida exclusivamente a partir del relevamiento del producto de software, toda instancia del concepto “DirectMetric()” debe ser también instancia del concepto “RequiredData()” (R1). En este caso, el término “Información Requerida” (*Required Data*) representa los datos puros a ser medidos, en un contexto ideal, sobre el producto de software (ver *"Definición de Información Requerida en un Esquema de Calidad"*). Si un dato puro se encuentra disponible, debe ser también marcado como dato disponible. Luego, se indica que toda instancia de “RequiredData()” que tenga asociada un atributo “available()” con valor “true”, debe ser también instancia del concepto “AvailableData()” (R2). Al igual que en el caso del término “Información Requerida”, el concepto “Información Disponible” (*Available Data*) refiere a los datos puros que pueden ser relevados sobre el producto de software (ver *"Definición Información Disponible para un Esquema de Calidad"*).

Una vez identificadas las instancias de “Información Requerida” e “Información Disponible”, es importante identificar la información calculable (es decir, aquella información que puede obtenerse por medio de la ejecución de un cálculo aplicado sobre un subconjunto de datos disponibles). Para estos fines, se crea un nuevo concepto denominado “Término Calculable” (*Calculable Term*). Existen cuatro situaciones diferentes en las cuales se puede derivar una instancia de “Término Calculable”, a saber:

i) Cuando una instancia de “DirectMetric()” es también instancia de “AvailableData()” (R3).

ii) Cuando una instancia de “SimpleTerm()” se encuentra vinculada a una instancia de “CalculableTerm()” por medio de la propiedad relacional “hasAsArgument()” (R4).

iii) Cuando una instancia de “ComplexTerm()” se encuentra vinculada a dos instancias diferentes de “CalculableTerm()” por medio de las propiedades relacionales “hasAsFirstArgument()” y “hasAsSecondArgument()” (R5).

iv) Cuando una instancia de “MeasurementFunction()” que se encuentra vinculada a una instancia de “IndirectMetric()” por medio de la propiedad relacional “isCalculatedBy()”, se relaciona con una instancia de “CalculableTerm()” por medio del vínculo “hasDefinition()” (R6).

En la primera situación, la información requerida para calcular la métrica (en este caso, directa) se encuentra disponible. Luego, la métrica es un término calculable. Este tipo de información es relevante debido a que, cuando una métrica directa puede ser calculada, todas las métricas indirectas que la utilizan también pueden ser calculadas. De acuerdo con la segunda situación, si un término simple tiene como argumento un término calculable, entonces el término simple es un término calculable. Lo mismo ocurre en la tercera situación pero referido a instancias de un término complejo (es decir, aquel término que involucra dos operandos para su resolución). Por último, la cuarta situación refiere a la definición de métricas indirectas. Específicamente, establece que toda instancia de métrica indirecta que tenga como función de medición una instancia basada en un término calculable, es también un término calculable.

Siguiendo estas cuatro reglas, es posible identificar el conjunto de todos los términos calculables dentro de un esquema de calidad (ya sea porque se obtienen directamente del producto o porque se derivan de un cálculo basado en la información disponible). Luego, se establece que toda instancia de “Metric()” que al mismo tiempo sea también instancia de “CalculableTerm()” es, asimismo, instancia de un nuevo

concepto denominado “CalculableMetric()” (R7). El concepto “Métrica Calculable” (*Calculable Metric*) es utilizado para identificar aquellas métricas que (independientemente de su tipo) pueden ser computadas debido a que todos sus términos son calculables (lo que implica que toda la información requerida para su determinación se encuentra disponible).

5.2.2 Consultas SPARQL para Ejecutar un Análisis de Cobertura

Tal como se ha indicado con anterioridad, los esquemas de calidad se definen en relación a un producto de software específico. Sin embargo, para cada subcaracterística de calidad, la capacidad del producto de software queda determinada por el conjunto de atributos internos que pueden ser relevados por medio de métricas. Esto implica que, aunque la definición se lleva a cabo a partir de un producto, la calidad es relevada a nivel de atributos de software. Tales atributos corresponden a entidades de software específicas.

Por este motivo, el análisis de cobertura propuesto para estudiar las métricas definidas en función del conjunto de datos disponibles, es ejecutado a nivel de entidad de software. Los resultados alcanzados a este nivel, son agrupados a fin de obtener mediciones de cobertura a nivel de artefacto, programa de computadora y producto de software. En el primer caso, el análisis utiliza el porcentaje de elementos calculables sobre el total de elementos definidos para determinar la cobertura sobre los aspectos de interés. En contraposición, en el segundo caso se agrupan los resultados de los niveles precedentes a fin de obtener valores de cobertura de calidad basados en indicadores predefinidos.

En términos generales, para que una entidad de software tenga una cobertura del 100%, todas las métricas asociadas a sus atributos deben ser calculables. Más específicamente, si se desea conocer la cobertura en relación a una característica o subcaracterística de calidad denominada Q, el porcentaje de cobertura sobre una

entidad de software nombrada E, queda definido matemáticamente como se indica en la Ecuación 5.7. De acuerdo con esta ecuación, la cobertura de un conjunto de datos disponibles sobre las métricas asociadas a una entidad de software en relación a un aspecto de calidad, es definida como el porcentaje de métricas calculables sobre el total de métricas definidas para el par (E,Q).

$$\frac{\text{Cantidad de métricas calculables de E referidas a Q}}{\text{Cantidad de métricas de E referidas a Q}} \times 100 \qquad (5.7)$$

A fin de relevar este porcentaje directamente desde las instancias de un esquema de calidad definido haciendo uso de la ontología QSO (en complemento con las reglas SWRL descriptas en el apartado precedente), una consulta SPARQL fue especificada e implementada haciendo uso de Protégé (Figura 5.11). Tal como puede observarse, la consulta utiliza la fórmula de cálculo definida en la Ecuación 5.7 como parte de la cláusula SELECT, con el objetivo de obtener el valor de respuesta requerido (identificado en la variable "?coverage"). Dicho valor queda definido como la relación entre la cantidad de instancias diferentes identificadas en las variables "?calculablemetricindividual" y "?metricindividual" respectivamente. Los individuos a ser contabilizados en ambos casos se obtienen a partir del conjunto de sentencias enumeradas dentro de la cláusula WHERE en combinación con los filtros indicados en la cláusula FILTER. Mientras que la cláusula WHERE determina el conjunto de instancias existentes referidas a las métricas (tanto calculables como definidas) en relación a una entidad E y a una subcaracterística de calidad Q, la cláusula FILTER restringe los valores deseados para las variables E y Q.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX f: <http://www.finalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT ((COUNT(DISTINCT ?calculablemetricindividual)/(COUNT(DISTINCT ?metricindividual))*100) AS ?coverage)
WHERE
{
        ?entity rdf:type sw:Entity .
        ?entity sw:entityName ?E .
        ?entity sw:hasAsProperty ?calculableattribute .
        ?calculableattribute rdf:type sw:Attribute .
        ?calculableattribute sw:isMeasuredBy ?calculablemetricindividual .
        ?calculablemetricindividual rdf:type sw:CalculableMetric .
        ?entity sw:hasAsProperty ?attribute .
        ?attribute rdf:type sw:Attribute .
        ?attribute sw:isMeasuredBy ?metricindividual .
        ?metricindividual rdf:type m:Metric .
        ?calculableattribute sw:isDescribedBy ?subcharacteristic .
        ?attribute sw:isDescribedBy ?subcharacteristic .
        ?subcharacteristic rdf:type ?Q .
        FILTER(?E = "E_Value" && regex(str(?Q),"Q_Value"))
}
```

Figura 5.11. Consulta SPARQL para analizar cobertura (entidad,subcaracterística).

La Figura 5.12 presenta una nueva consulta SPARQL, diseñada e implementada en Protégé a fin de analizar la cobertura en relación a una entidad de software E sobre una característica de calidad Q. El criterio aplicado para determinar el porcentaje de cobertura es idéntico al propuesto en la Ecuación 5.7, quedando definida la consulta de forma similar a lo expuesto con anterioridad. La diferencia radica en que la cláusula WHERE incorpora una sentencia adicional para determinar el valor de Q según una instancia de característica.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX f: <http://www.finalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT ((COUNT(DISTINCT ?calculablemetricindividual)/(COUNT(DISTINCT ?metricindividual))*100) AS ?coverage)
WHERE
{
        ?entity rdf:type sw:Entity .
        ?entity sw:entityName ?E .
        ?entity sw:hasAsProperty ?calculableattribute .
        ?calculableattribute rdf:type sw:Attribute .
        ?calculableattribute sw:isMeasuredBy ?calculablemetricindividual .
        ?calculablemetricindividual rdf:type sw:CalculableMetric .
        ?entity sw:hasAsProperty ?attribute .
        ?attribute rdf:type sw:Attribute .
        ?attribute sw:isMeasuredBy ?metricindividual .
        ?metricindividual rdf:type m:Metric .
        ?calculableattribute sw:isDescribedBy ?subcharacteristic .
        ?attribute sw:isDescribedBy ?subcharacteristic .
        ?subcharacteristic q:belongs ?characteristic .
        ?characteristic rdf:type ?Q .
        FILTER(?E = "E_Value" && regex(str(?Q),"Q_Value"))
}
```

Figura 5.12. Consulta SPARQL para analizar cobertura (entidad,característica).

Teniendo en cuenta que las consultas especificadas en SPARQL son de utilidad para analizar la cobertura a nivel de entidad y, considerando que en un esquema de calidad se definen múltiples niveles de referencia por encima del nivel entidad en relación al producto de software (como ser artefacto, programa de computadora y producto de software), se establecieron los lineamientos necesarios para analizar la cobertura de calidad en los niveles definidos por encima del nivel entidad.

En este contexto, no es válido expresar el análisis de cobertura de los niveles superiores de la misma manera que los realizados al nivel de entidad. A nivel entidad, se obtiene el porcentaje de métricas que pueden ser calculadas partiendo de un conjunto de datos disponibles. Tal porcentaje agrupa las métricas definidas para una característica/ subcaracterística de calidad Q en relación a una entidad E. Luego, cada valor de cobertura $C_{(E,Q)}$ indica el porcentaje de mediciones de calidad que pueden realizarse sobre E a fin de evaluar la característica/subcaracterística Q. Si se tiene un indicador asociado, este valor permite determinar si los datos disponibles son o no suficientes para evaluar el aspecto de calidad Q en relación a la entidad E en un momento dado. Sin embargo, por sí solo no contribuye a tener una visión global del estudio de calidad requerido sobre el producto de software. Es decir, ¿un valor de $C_{(E,Q)}$ bajo indica que no se podrán cumplir los requerimientos de calidad definidos?, ¿un valor alto indica lo contrario?, ¿es posible que un valor bajo de $C_{(E,Q)}$ no impacte negativamente en el relevamiento de la calidad final esperada/deseada?.

Para poder responder este tipo de preguntas, es necesario conocer la proporción de cumplimiento de la cobertura de calidad (definida como P) en función de un valor de referencia I (el cual debe ser definido por el interesado) a fin de establecer valores generales para Q. Al tener un valor general P referido al estudio de un aspecto de calidad Q en relación a un indicador I, es posible comparar múltiples características/subcaracterísticas $\{Q_1,Q_2,...,Q_m\}$ con el objetivo de determinar si los datos

disponibles permiten relevar un conjunto útil de métricas de calidad que permitan garantizar un mínimo cumplimiento I de los aspectos de calidad definidos. La Figura 5.13 presenta gráficamente el agrupamiento propuesto.

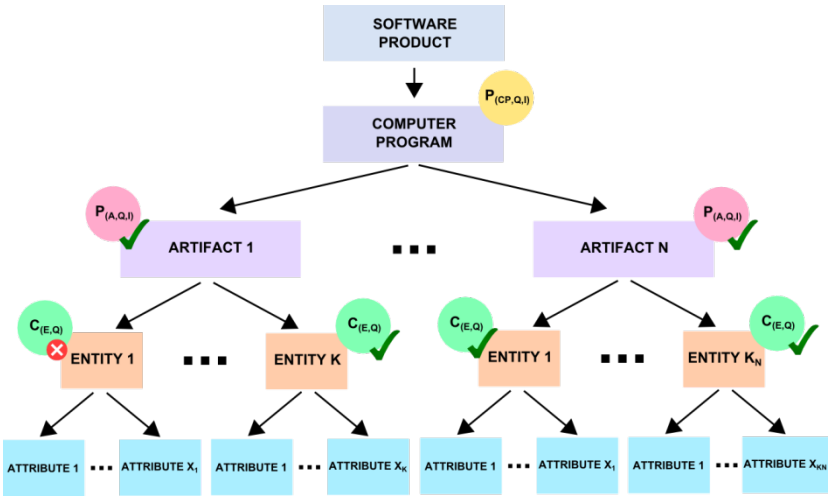


Figura 5.13. Análisis de cobertura a nivel artefacto, programa y producto de software.

Como puede observarse, la figura define un producto de software haciendo uso de un programa de computadora dividido en N artefactos. Cada i–ésimo artefacto (con $1 \leq i \leq N$), a su vez, queda definido en términos de K_i entidades. Cada entidad, tiene asociado un valor de cobertura $C_{(E,Q)}$ resultante del análisis detallado con anterioridad. Luego, para obtener valores de cobertura en los niveles superiores, debe definirse un valor de referencia de aceptación denominado I. En base a este valor, se marcan las entidades que cumplen con la restricción $C_{(E,Q)} > I$. En la figura propuesta, se utiliza una tilde verde y una cruz roja para indicar el cumplimiento e incumplimiento, respectivamente, de esta condición sobre cada uno de los valores de cobertura esquematizados. Luego, la proporción de cumplimiento de la cobertura de calidad en función de un valor de referencia I para un artefacto A, está dada por el valor $P_{(A,Q,I)}$ (el cual queda definido como se expresa en la Ecuación 5.8).

$$\frac{\text{Cantidad de elementos en } \{ C_{(E,Q)} > I \,/\, E \in A \} \times 100}{\text{Cantidad de elementos en } \{ C_{(E,Q)} \,/\, E \in A \}} \qquad (5.8)$$

El estudio de P sobre un artefacto A, puede replicarse nuevamente al nivel de programa de computadora si se plantea el mismo cálculo en función de los valores obtenidos en el nivel previo. Es decir, para un programa de computadora CP, se debe computar la relación entre la cantidad de porcentajes $P_{(A,Q,I)}$ que cumplen con $P_{(A,Q,I)} > I$ (para todo artefacto A que se encuentre incluido en el programa CP) y la cantidad de porcentajes definidos para todos los artefactos en CP. Este nuevo valor se denomina $P_{(CP,Q,I)}$. Luego, los valores obtenidos para un $P_{(x,Q,I)}$ deben ser interpretados como *"el porcentaje de cálculo certero que se podrá tener sobre la característica/subcaracterística de calidad Q en base a un umbral I para el elemento de software 'x' haciendo uso de la información disponible en un momento dado del proceso de desarrollo"*.

En este sentido, el análisis de cobertura en los niveles superiores no busca determinar el conjunto de elementos calculables asociados a dicho nivel, sino un valor de utilidad que permita decidir si los elementos calculables referidos a los niveles inferiores, en un momento dado, son suficientes (o no) para estudiar un aspecto de calidad definido. Teniendo en cuenta que un producto de software se define en términos de un programa de computadora (pudiendo existir más de uno), no es posible extender el análisis de cobertura más allá del nivel CP.

Es importante destacar que, aunque es posible especificar consultas SPARQL para obtener los valores de $P_{(A,Q,I)}$ y $P_{(CP,Q,I)}$, no es conveniente. Esto se debe a que tales consultas deben reiterar el proceso de obtención de los valores $C_{(E,Q)}$ (es decir, no pueden manipular valores previamente calculados). Sin embargo, haciendo uso de las preguntas de competencia en combinación con las consultas diseñadas para calcular $C_{(E,Q)}$, es posible determinar los valores de $P_{(A,Q,I)}$ y $P_{(CP,Q,I)}$ sin mayores dificultades.

5.3 Actividad para Especificación y Uso de Instancia QSO

Con el objetivo de indicar los lineamientos sobre cómo debe utilizarse la ontología QSO junto con sus reglas y consultas, se ha definido una actividad que indica el conjunto de interacciones requeridas para hacer uso de la definición del esquema de calidad y del análisis de cobertura propuesto. Las tareas identificadas en esta actividad son tentativas, ya que tanto la definición del esquema de calidad como así también las consultas a realizar sobre el mismo, quedan definidas por las intenciones del grupo de interés asociado a la definición y el desarrollo del producto de software.

En este contexto, la actividad propuesta identifica tareas asociadas a dos tipos de actores, a saber:

i) *Equipo de desarrollo (Development team)*: Identifica al principal usuario de la ontología QSO como el conjunto de personas involucradas en las distintas tareas a ser realizadas durante el proceso de desarrollo. Básicamente representa el grupo de interés asociado a la ontología.

ii) *Ontología QSO (QSO ontology)*: Refiere al modelo semántico diseñado como mecanismo de soporte para definir y analizar esquemas de calidad. Este actor debe ser considerado en términos de la definición semántica propuesta en las secciones precedentes, quedando definido por las herramientas ya mencionadas para la creación de instancias QSO o por herramientas de software basadas en ontologías creadas específicamente con el objetivo de automatizar la especificación y el análisis de instancias QSO.

La Figura 5.14 presenta un diagrama de actividad especificado en UML en el cual se detallan nueve tareas, a saber:

1. *Identificar el conjunto de elementos necesarios para definir el esquema de calidad (List the set of elements required to define the quality scheme)*: Se define el conjunto de tripletas del tipo <*atributo de software, métrica,*

subcaracterística de calidad>, las cuales serán utilizadas como elementos componentes del esquema de calidad asociado al producto de software bajo desarrollo.

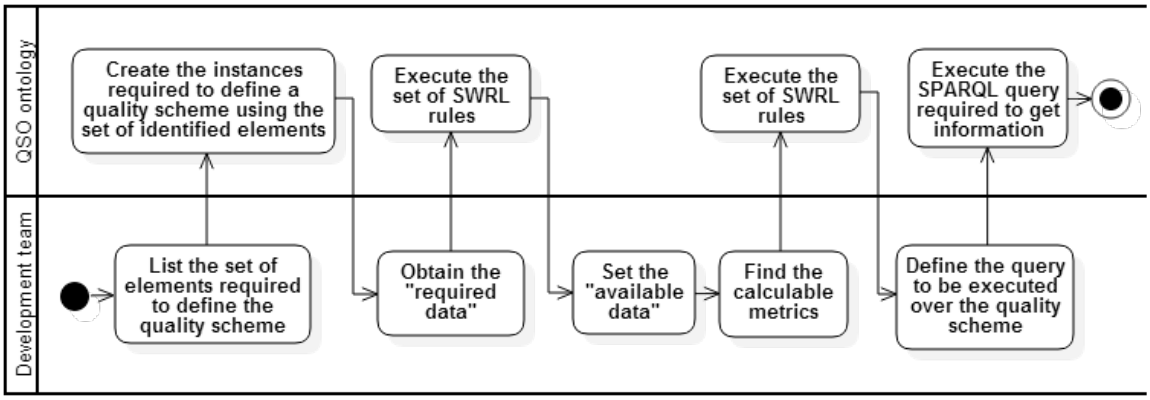


Figura 5.14. Diagrama de actividad para el uso de la ontología QSO.

2. *Crear las instancias requeridas para definir un esquema de calidad utilizando los elementos previamente identificados (Create the instances required to define a quality scheme using the set of identified elements)*: Se generan las instancias QSO para cada uno de los conceptos identificados como parte de la definición de las tripletas en virtud de representar los distintos elementos a ser incluidos como parte del esquema de calidad a documentar. Además, se ejecutan las reglas de derivación del conocimiento para cada uno de los dominios modelados (es decir, las reglas SWRL definidas en cada uno de los modelos semánticos), a fin de obtener todas las instancias QSO requeridas para la definición del esquema de calidad. El resultado de esta tarea es el esquema de calidad final definido en términos de la ontología QSO.
3. *Obtener el conjunto de datos que constituyen la "información requerida" (Obtain the set of "required data")*: Una vez definido el esquema de calidad, es importante obtener la información requerida para lograr el 100% de

cobertura del mismo. Esta información será de utilidad para establecer el conjunto de elementos de software a ser relevados y la forma de obtención de las mediciones requeridas (datos) sobre dichos elementos.

4. *Ejecutar el conjunto de reglas SWRL (Execute the set of SWRL rules)*: A fin de completar la obtención de la información requerida, se deben ejecutar las reglas SWRL detalladas en la Tabla 5.4. El objetivo de la ejecución de estas reglas es derivar las instancias del concepto "RequiredData()".
5. *Indicar el conjunto de datos que constituyen la "información disponible" (Set the "available data")*: Una vez especificado el esquema de calidad junto con la información requerida para su completo relevamiento, es posible utilizar el documento formulado en las distintas etapas de desarrollo a fin de evaluar las propiedades de calidad identificadas. Teniendo en cuenta que es posible que no se encuentre disponible toda la información requerida para analizar la calidad en todas las etapas de desarrollo (incluso podría no existir en etapas tempranas del proceso), es importante que los desarrolladores interesados en la evaluación de la calidad identifiquen la información disponible (es decir, los datos que pueden ser medidos en relación al producto final con respecto a la etapa de desarrollo actual). Esta información debe ser introducida en el esquema de calidad por medio de la especificación de instancias del concepto "AvailableData()".
6. *Determinar el conjunto de métricas calculables (Find the calculable metrics)*: En base al esquema de calidad definido y a los datos disponibles, se desea conocer el conjunto de métricas de software que pueden ser calculadas haciendo uso de esta información. Esto permite determinar si es necesario replantear las mediciones realizadas sobre el artefacto actual o si, por el contrario, la información actual es lo suficientemente completa como para evaluar la calidad esperada en el artefacto en esta etapa de desarrollo.
7. *Ejecutar el conjunto de reglas SWRL (Execute the set of SWRL rules)*: Con el objetivo de obtener las métricas calculables, se deben ejecutar nuevamente

las reglas SWRL detalladas en la Tabla 5.4. En este caso, luego de esta ejecución, se relevarán todas las instancias del concepto "CalculableMetric()". Haciendo uso de esta información, el grupo de interés deberá evaluar su conformidad (o no) en relación a la información disponible.

8. *Definir los elementos involucrados en la consulta a ejecutar sobre el esquema de calidad (Define the query to be executed over the quality scheme)*: Esta tarea debe ser realizada luego de que se han ejecutado las tareas 1 a 7, ya que busca encontrar una respuesta a una determinada pregunta formulada en términos de los elementos instanciados a partir de la ontología QSO (tanto de las instancias definidas como así también de las instancias derivadas). Esta tarea puede realizarse tantas veces como sea necesario, ya que pueden formularse múltiples preguntas en relación a un mismo esquema de calidad. Ya sea para preguntas de competencia o para preguntas referentes al análisis de cobertura, el objetivo de esta tarea es identificar las instancias sobre las cuales se deben formular las consultas.
9. *Ejecutar la consulta SPARQL requerida para obtener la información de cobertura necesaria para analizar el esquema de calidad en relación a la información deseada (Execute the SPARQL query required to get information)*: Una vez definido el conjunto de instancias de interés, debe ejecutarse la consulta SPARQL (asociada a la pregunta bajo análisis) sobre las instancias QSO que forman parte del esquema de calidad. Con la respuesta a dicha consulta, el grupo de usuarios interesados podrá tomar una decisión conforme a la información obtenida en relación al tipo de pregunta.

Por medio del seguimiento de las tareas propuestas como parte de la actividad, cualquier persona interesada podrá formular, estudiar y/o analizar la cobertura de esquemas de calidad definidos en términos de la ontología QSO (para un producto de software determinado). La decisión acerca del momento en el cual debe ser ejecutado un análisis de cobertura dependerá de la necesidad de realizar una evaluación de

calidad en un momento dado. Teniendo en cuenta que dicho análisis es realizado utilizando la información disponible, múltiples análisis de cobertura pueden ejecutarse sobre un mismo esquema de calidad en distintas etapas del proceso de desarrollo.

5.4 Beneficios y Limitaciones del Uso de la Ontología QSO

La ontología QSO definida en la sección precedente provee un mecanismo de soporte para la definición de un documento de calidad universal para un producto de software bajo desarrollo. Esta estrategia de documentación conlleva múltiples beneficios para los diversos grupos de interés, entre los que se destacan la formalización de una jerarquía de propiedades de calidad, la definición de métricas de software de forma explícita como parte de la especificación de calidad y el análisis de la información relevante para garantizar el cumplimiento de las propiedades de calidad definidas para el producto. En este contexto, los esquemas de calidad instanciados a partir de la ontología QSO posibilitan un estudio integral de los principales aspectos de calidad internos requeridos para un producto de software en particular.

Aunque como parte del proceso de desarrollo de software pueden construirse otro tipo de documentos para especificar la calidad, la definición y el estudio de un esquema de calidad conforme la actividad propuesta en el apartado *"Actividad para la Especificación y Uso de una Instancia de QSO"* garantiza un adecuado entendimiento de las propiedades de calidad y sus relaciones, contribuyendo a la formalización de un documento con contenido semántico que difiere de las perspectivas estipuladas en otros mecanismos. El conjunto de tareas definido como parte de la actividad propuesta permite dividir las responsabilidades de la documentación de la calidad en dos grandes grupos, dejando la responsabilidad de la definición y uso del esquema de calidad al equipo de desarrollo pero, al mismo tiempo, garantizando que la instanciación sea coherente en términos de los conceptos y relaciones propuestas como parte de los modelos semánticos. Esto permite, además, que la información conceptualizada como

instancia de la ontología QSO sea utilizada como base de conocimiento para derivar nuevas propiedades y relaciones entre los datos especificados. Esta es una de las principales ventajas que posee la aplicación de ontologías en combinación con reglas de derivación de conocimiento para la definición de esquemas de calidad.

Al integrar un conjunto de consultas propias del dominio de la definición semántica, se posibilita la obtención de información relevante tanto en términos de la definición de calidad como así también en relación al análisis de cobertura. En este último caso, se provee un mecanismo de consulta que ayuda a determinar el impacto de la información disponible sobre las características de calidad requeridas en términos de las métricas calculables. De esta manera, el análisis de cobertura contribuye al análisis de los riesgos asociados a la falta de mediciones de calidad en un determinado punto del proceso de desarrollo, permitiéndole al equipo de desarrollo reformular el estudio de los artefactos conforme se avanza en la construcción del producto. En este sentido, es importante destacar que un único esquema de calidad puede derivarse para un producto de software por medio de la unificación de las propiedades de calidad definidas para sus artefactos. Además, para un mismo producto de software, pueden existir múltiples definiciones de esquemas de calidad según la etapa del proceso de desarrollo en la cual se encuentre el producto y los artefactos involucrados en su construcción.

En relación al momento en el cual debe definirse el esquema de calidad, existen dos posibilidades: en etapas tempranas del proceso de desarrollo o ya avanzado el proceso de desarrollo. En el primer caso, una definición temprana permite tomar en consideración aspectos generales derivados de la especificación de requerimientos de software, los cuales pueden sentar las bases requeridas para fijar los datos a ser medidos en cada artefacto identificado. Esto permite plantear su desarrollo de forma tal que pueda obtenerse la información necesaria para evaluar las propiedades de calidad definidas. Sin embargo, nada garantiza que la definición realizada sea susceptible de ser efectuada sobre los artefactos propuestos. Por su parte, si la

especificación de la calidad se realiza en el momento en el que el artefacto es creado, el esquema tomará en consideración la información disponible por lo que su cobertura será casi completa. Sin embargo, es probable que esta definición se encuentre limitada por la visión del desarrollador, no incorporando aspectos vinculados a los requerimientos reales definidos por el usuario. Luego, ninguna de estas situaciones es ideal. En todos los casos, la recomendación es que la especificación y medición de la calidad sea una tarea continua a lo largo del proceso de desarrollo. Es decir, especificar un único esquema de calidad en etapas tempranas de desarrollo y, luego, refinar su contenido a medida que se avance con la construcción del producto.

Es importante destacar que no todos los artefactos creados a lo largo del proceso de desarrollo deben ser evaluados en términos de la calidad del producto. Solo aquellos artefactos que se relacionen (o puedan ser relacionados) directamente con las propiedades de calidad requeridas para el producto final deben ser asociados a las características y métricas definidas en un esquema de calidad. Los artefactos restantes no deben ser contemplados dentro de este documento. Esto se debe a que los esquemas de calidad permiten especificar propiedades de calidad definidas a partir de los requerimientos de usuario. Tales requerimientos se vinculan directamente al producto final por lo que, aunque pueden ser evaluados en los productos intermedios, no refieren a la calidad de los mismos. En estos casos, deben definirse nuevas estrategias de evaluación que contribuyan a garantizar su calidad. Estos aspectos no se encuentran incluidos como parte de la ontología QSO.

Aunque la ontología QSO ha sido definida en base a modelos semánticos provenientes de diferentes dominios, su integración en una única ontología contribuye a la especificación de un documento integral. Sin embargo, la independencia de los modelos semánticos brinda un nivel de modularidad que permite tanto el reemplazo como la reutilización de los modelos en otros escenarios y/o su combinación con otros dominios en virtud de obtener nuevos tipos de inferencias e instancias. Específicamente, el modelo semántico que refiere al dominio de calidad de producto de

software puede reemplazarse por un modelo de calidad diferente a fin de formalizar un nuevo tipo de esquema basado en una jerarquía de conceptos de calidad alternativa a la propuesta en la ontología QSO. Es decir, dado que la ontología QSO se centra en el modelo de calidad de producto de software definido en el estándar ISO/IEC 25010, el esquema de calidad resultante se centra en atributos internos del producto de software bajo análisis[3]. Sin embargo, si se reemplaza el modelo semántico de calidad de producto de software por el modelo de calidad de uso definido en el mismo estándar (el cual mantiene la jerarquía de conceptos en términos de características y subcaracterísticas), el documento de calidad resultante referirá a aspectos de calidad de uso del producto. En este caso, las relaciones definidas en la integración de los modelos semánticos actúan como interfaz entre los dominios. Al reemplazar uno de los modelos por otro equivalente, el enfoque propuesto permite reutilizar los modelos restantes en virtud de definir un nuevo tipo de documento de calidad. Aunque el modelo semántico de calidad también podría ser reemplazado por modelos de calidad de producto propuestos por otros autores, en el caso de que tales modelos no respeten las interfaces definidas, deben realizarse las adaptaciones requeridas a fin de lograr una correcta integración. En este último escenario, la incorporación de nuevas jerarquías de calidad conlleva a una redefinición de las reglas y consultas que permiten analizar la estructura de los modelos semánticos.

Dada la universalidad con la cual se ha definido el modelo semántico de métricas de software, sus conceptos y relaciones pueden aplicarse en contextos ajenos a la

[3] Tales atributos refieren a la calidad interna del producto. La calidad interna se define como *"la totalidad de atributos de un producto que determina su capacidad de satisfacer necesidades explícitas e implícitas cuando es usado bajo condiciones específicas"*. En contraposición, la calidad externa es *"el grado en el que un producto satisface necesidades explícitas e implícitas cuando se utiliza bajo condiciones específicas"*. Mientras que la calidad interna puede ser medida y evaluada por medio de atributos estáticos de documentos tales como especificación de requerimientos, arquitectura, diseño y piezas de código fuente; la calidad externa puede ser medida y evaluada cuando un módulo (o la aplicación completa) es ejecutada en una computadora o en una red simulando lo más cercanamente posible un ambiente real. En este sentido, un esquema de calidad definido haciendo uso de la ontología QSO especifica todos los atributos internos del producto (calidad interna). El aseguramiento de la calidad interna no es suficiente para asegurar calidad externa, debiendo entenderse que la calidad no es un concepto absoluto sino que depende de condiciones y contextos de uso específicos.

medición de calidad. Lo mismo ocurre en el caso del modelo semántico de producto de software. Aún más, los términos y relaciones incluidos en este modelo puede refinarse a fin de representar un mayor nivel de detalle en relación al producto de software bajo análisis. Una mejor descripción del producto permitiría un nuevo nivel de consultas SPARQL, dando lugar a un análisis de calidad profundo.

Conclusiones

El uso de ontologías como mecanismo de soporte para el estudio de diversos aspectos relacionados con la ingeniería de software ha crecido en los últimos años. En este contexto, las ontologías permiten realizar una definición explícita de una conceptualización asociada a un dominio bajo estudio (Gruber 1993), permitiendo especificar de forma no ambigua la estructura de conocimiento asociada a dicho dominio. Esto posibilita compartir, reusar y razonar a partir del contenido detallado como parte del modelo ontológico (Orgun y Meyer 2008).

En este capítulo se hace uso de la estrategia de documentación definida en el capítulo previo (es decir, los esquemas de calidad), formalizando su composición en base a una ontología de dominio. Se presenta una única ontología basada en tres modelos semánticos, la cual permite documentar esquemas de calidad haciendo uso del modelo de calidad de producto definido en el estándar ISO/IEC 25010. El modelo final permite generar instancias en un dominio específico, contribuyendo al entendimiento de la calidad requerida para el producto de software documentado en relación a distintas perspectivas (producto, métricas de software y propiedades de calidad). Además, por medio de un análisis de cobertura, un esquema de calidad puede ser utilizado como fuente de información para identificar los datos requeridos a fin de relevar las distintas propiedades de calidad en relación a los distintos artefactos.

La documentación de la calidad basada en modelos semánticos aprovecha todos los beneficios enunciados con anterioridad en relación al uso de ontologías, permitiendo a los desarrolladores documentar las propiedades de calidad de un producto de software haciendo uso de instancias que, luego, pueden ser utilizadas para derivar nuevo conocimiento sobre el estudio de la calidad.

En el siguiente capítulo se presenta una extensión/aplicación de la ontología QSO

específicamente diseñada para brindar soporte al estudio de la calidad en aplicaciones web desarrolladas sobre entornos de computación en la nube.

Capítulo 6. Esquema de Calidad para Servicios de Software en la Nube

Dado el rápido crecimiento de las tecnologías web, no es sencillo encontrar herramientas que den soporte al análisis de la calidad de servicios de software basados en entornos de computación en la nube. En el capítulo previo se ha presentado una estrategia de documentación basada en ontologías para la especificación de esquemas de calidad en productos de software genéricos. En este capítulo se enfatiza la necesidad de evaluar la calidad a nivel de capa de aplicación, extendiendo la propuesta precedente a fin de incorporar propiedades asociadas a servicios de software. De esta manera, se integra en un único documento la información relacionada a la definición y entendimiento de las propiedades de calidad web, junto con la forma en la cual estas propiedades deben ser relevadas. Se trabaja sobre el modelo de calidad tomado como base en la ontología QSO, complementando su definición con conceptos propios del dominio. Luego, para relevar cada atributo identificado, se proponen métricas. El esquema de calidad resultante contribuye al entendimiento de la calidad en este tipo de productos, posibilitando el análisis de la información requerida para su medición en un contexto específico.

6.1 Problemas para Especificar Propiedades de Calidad en Servicios de Software

En el caso de entornos de CC, además de las dificultades tradicionales para evaluar la calidad de software, se presenta una dificultad adicional en el entendimiento

y la evaluación de la calidad. Los entornos de desarrollo tradicionales presentan limitaciones al momento de aplicarse en la evaluación de calidad de SaaS. Esto se debe principalmente a la diferencia existente entre el paradigma computacional tradicional y el paradigma de CC, lo que da como consecuencia que tales entornos no realicen una evaluación efectiva de los aspectos de calidad específicos asociados a este tipo de computación (Lewis 2010).

En este contexto, en el Capítulo 4 se ha definido un documento para la especificación de la calidad en productos de software genéricos, el cual ha sido formalizado haciendo uso de ontologías (Capítulo 5). Como resultado, este documento se denomina *esquema de calidad* y permite especificar (de forma ontológica) un conjunto de propiedades de calidad (basadas en un modelo de calidad de producto de software estándar), para cada una de las cuales se define un conjunto de métricas que posibilitan su relevamiento sobre un artefacto de software específico.

Dado que la definición semántica de este esquema se basa en un modelo de calidad general (de acuerdo a lo expuesto en el apartado *"Tipos de Modelos de Calidad"*) y, en virtud de organizar los aspectos de calidad relevantes para los productos de software basados en servicios de CC, es posible refinar la definición del modelo de calidad de producto de software del estándar ISO/IEC 25010 a fin de obtener un modelo susceptible de ser utilizado como base en la definición de un esquema de calidad genérico aplicable a cualquier tipo de modelo de distribución basado en SaaS. En este contexto, en la siguiente sección se presenta un modelo de calidad para SaaS definido en base al modelo de calidad de producto de software detallado en el estándar ISO/IEC 25010.

6.2 Modelo de Calidad para Servicios de Software

6.2.1 Factores de Calidad Relevantes

Diversos autores han enfatizado la importancia de relevar la calidad en SaaS (Lee, Lee y Kim 2009; Gao y colab. 2011; Khan y Singh 2012; Wen y Dong 2013; Breu, Kuntzmann-Combelles y Felderer 2014; Zhao y Zhu 2014). Aunque sus líneas de investigación difieren, las propiedades identificadas (factores, atributos y características) son útiles para la construcción de un modelo de calidad SaaS. La Tabla 6.1 resume estos aspectos.

PROPIEDAD	DESCRIPCIÓN	FUENTE
Reusabilidad *(reusability)*	Capacidad de los componentes de un servicio de ser reutilizados en la construcción de otras aplicaciones del mismo tipo.	(Khan y Singh 2012; Wen y Dong 2013)
Disponibilidad *(availability)*	Si el SaaS no se encuentra disponible, los consumidores no podrán hacer uso de sus funcionalidades.	(Breu, Kuntzmann-Combelles y Felderer 2014; Lee, Lee y Kim 2009; Zhao y Zhu 2014)
Escalabilidad *(scalability)*	Habilidad de aumentar/disminuir de forma ágil y rápida los recursos asignados al entorno según el nivel de demanda.	(Breu, Kuntzmann-Combelles y Felderer 2014; Wen y Dong 2013; Gao y colab. 2011)
Personalización de servicios *(service customizability)*	Capacidad del SaaS de ser modificados de acuerdo a requerimientos específicos de los consumidores.	(Lee, Lee y Kim 2009; Wen y Dong 2013)
Uniformidad funcional *(functional feature commonality)*	Nivel de homogeneidad de las características funcionales del servicio de software.	(Lee, Lee y Kim 2009)
Uniformidad no funcional *(non-functional feature commonality)*	Nivel de homogeneidad de las características no funcionales del servicio.	(Lee, Lee y Kim 2009)

Uso de infraestructura *(infrastructure utilization)*	Cantidad de recursos de infraestructura reservados por el servicio que han sido efectivamente utilizados.	(Gao y colab. 2011)
Tiempo de invocación *(invocation time)*	Tiempo que tarda el sistema SaaS en ejecutar la invocación de un servicio por el cual el usuario está pagando.	(Lee, Lee y Kim 2009; Zhao y Zhu 2014)
Estabilidad del servicio *(service stability)*	Grado en el cual el servicio de software funciona sin que se den fallas y/o errores.	(Lee, Lee y Kim 2009)
Precisión del servicio *(service accuracy)*	Exactitud que posee el servicio de software al dar respuesta a una solicitud de usuario específica.	(Zhao y Zhu 2014)
Cobertura de recursos *(resource coverage)*	Promedio de recursos asignados en relación a la cantidad de recursos solicitados.	(Gao y colab. 2011)
Robustez del servicio *(robustness of service)*	Probabilidad de que el servicio de software pueda ser utilizado por los consumidores en un instante de tiempo dado.	(Gao y colab. 2011)

Tabla 6.1. Principales propiedades de calidad a evaluar en entornos SaaS.

6.2.2 Integración de los Factores al Modelo de Calidad ISO/IEC 25010

A fin de complementar el modelo de calidad de producto de software propuesto en el estándar ISO/IEC 25010 con las propiedades definidas en la Tabla 6.1, se tomaron como referencia los lineamientos propuestos en (Kläs, Lampasona y Münch 2013). El modelo de calidad resultante de esta especificación se ajusta a las características propias de los entornos SaaS. La Figura 6.1 esquematiza la composición del modelo resultante (tomando como base la esquematización de la Figura 4.3).

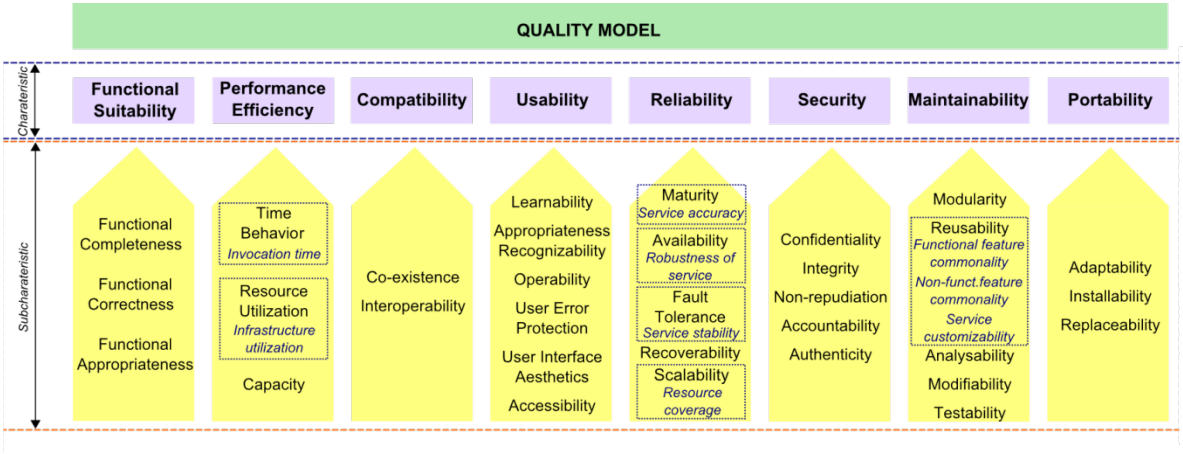


Figura 6.1. Modelo de calidad de producto basado en SaaS.

Tal como puede observarse, este modelo mantiene las características y subcaracterísticas propuestas en el estándar (modelo original) y, a su vez, las combina con nuevas subcaracterísticas y atributos. Estos últimos quedan representados por medio de etiquetas azules asociadas a subcaracterísticas específicas por medio de cuadros delimitados con línea de puntos.

No se incorporan nuevas características a fin de respetar las jerarquías propuestas en el estándar. En el caso de las propiedades SaaS que coinciden con elementos identificados en el modelo original, se mantiene el nivel de clasificación sugerido en el estándar a fin de no alterar el esquema base. Por su parte, aunque en el modelo de calidad original no se definen atributos (debido a que su identificación depende del tipo de producto sobre el cual se desee aplicar el estudio de calidad), el modelo resultante incluye atributos específicos que se derivan de la naturaleza SaaS del producto bajo estudio.

Estas modificaciones dan como resultado una taxonomía de propiedades de calidad clasificada conforme la jerarquía propuesta en el estándar ISO/IEC 25010 que es mucho más específica, detallada y completa en comparación con la original.

Incorporación de Subcaracterísticas de Calidad para Entornos SaaS

Del conjunto de propiedades identificadas en la Tabla 6.1, las propiedades "Reusabilidad" (*reusability*), "Disponibilidad" (*availability*) y "Escalabilidad" (*scalability*) se integraron a la jerarquía como subcaracterísticas de calidad.

La estructura del modelo original presenta una correspondencia directa de las propiedades "Reusabilidad" y "Disponibilidad" con subcaracterísticas de calidad definidas en la taxonomía original (incorporadas como descriptores de las características de calidad "Mantenibilidad" y "Confiabilidad", respectivamente). Sin embargo, no ocurre lo mismo para la propiedad "Escalabilidad", la cual no se encuentra definida como parte de la jerarquía propuesta en el modelo original.

En términos generales, la "Escalabilidad" refiere a una propiedad deseable de un sistema, red o proceso que indica su habilidad para: *i)* reaccionar y adaptarse sin perder calidad, *ii)* manejar el crecimiento continuo de trabajo de manera fluida, o *iii)* estar preparado para hacerse más grande sin perder calidad en los servicios ofrecidos. Considerando que la "Confiabilidad" es la capacidad de un sistema o componente de software de llevar a cabo sus funciones cuando se lo utiliza bajo condiciones definidas y/o periodos de tiempo determinados; es evidente que la posibilidad de adaptarse a los cambios en función de la demanda es un aspecto relevante a ser evaluado en relación a esta característica. Luego, la propiedad "Escalabilidad" (relacionada al entorno SaaS) se incorpora a la jerarquía del modelo original como una nueva subcaracterística de calidad asociada a la característica "Confiabilidad" (*reliability*).

Incorporación de Atributos de Calidad para Entornos SaaS

Tomando como base el modelo de calidad previo (obtenido como resultado de la incorporación de la subcaracterística de calidad "Escalabilidad"), se incorporaron las propiedades SaaS faltantes (subconjunto de las propiedades propuestas en la Tabla 6.1) como parte del nivel de la jerarquía correspondiente a los atributos de calidad.

En este contexto, las propiedades definidas como "Tiempo de Invocación" (*invocation time*) y "Uso de Infraestructura" (*infrastructure utilization*) se asociaron al modelo preliminar como atributos de calidad vinculados a las subcaracterísticas de calidad "Comportamiento Temporal" y "Utilización de Recursos", respectivamente (ambas incluidas como descriptores de la característica de calidad "Rendimiento"). En el primer caso, la asociación entre el atributo "Tiempo de Invocación" y la subcaracterística "Comportamiento Temporal" se justifica en la definición de la subcaracterística. Dado que refiere a la "evaluación de tiempos de respuesta, tiempos de procesamiento y tasas de throughput en condiciones de ejecución específicas"; resulta conveniente incluir esta propiedad SaaS (la cual, en términos generales, refiere al tiempo que tarda el sistema en ejecutar un servicio) como atributo descriptor de dicha subcaracterística. Una dependencia similar se presenta entre el atributo "Uso de Infraestructura" y la subcaracterística "Utilización de Recursos", ya que la subcaracterística refiere a "la cantidad y tipo de recursos usados cuando el software es ejecutado en contextos específicos" y la propiedad SaaS busca "examinar la forma en la cual son utilizados los recursos de la infraestructura de CC".

Por otro lado, como parte de la jerarquía de elementos asociada a la característica "Confiabilidad" se incorporaron cuatro atributos de calidad, a saber: "Precisión del Servicio" (*service accuracy*), "Robustez del Servicio" (*robustness of service*), "Estabilidad del Servicio" (*service stability*) y "Cobertura de Recursos" (*resource coverage*). Cada uno de estos atributos se anexa como descriptor de una subcaracterística específica, a saber:

- La propiedad "Precisión del Servicio" se asocia a la subcaracterística "Madurez": Esto se debe a que dicha subcaracterística refiere a "la capacidad del sistema o servicio de satisfacer las necesidades de confiabilidad en condiciones normales de ejecución". En este sentido, en un contexto ideal, un servicio con alto nivel de madurez debe tener un alto nivel de exactitud en sus respuestas (es decir, mayor precisión de servicio). Luego, la propiedad SaaS puede ser vista como un descriptor de esta subcaracterística.

- La propiedad “Robustez del Servicio” se incorpora a la subcaracterística “Disponibilidad”: Teniendo en cuenta que esta subcaracterística de calidad se relaciona con “la capacidad del sistema o servicio de estar operativo y accesible para su uso en un instante de tiempo dado”, es evidente que la propiedad SaaS actúa como un descriptor susceptible de estar asociado a esta subcaracterística.
- La propiedad “Estabilidad del Servicio” se vincula a la subcaracterística “Tolerancia a Fallos”: Esta subcaracterística de calidad corresponde a “la habilidad del software para operar según lo previsto ante la presencia de fallos de hardware y/o software”. Dado que la propiedad “Estabilidad del Servicio” busca “analizar el comportamiento del servicio sin fallas”, su incorporación como parte de la descripción de la subcaracterística de calidad “Tolerancia a Fallos” resulta apropiada.
- La propiedad “Cobertura de Recursos” se integra a la subcaracterística “Escalabilidad”: Tomando en consideración que la propiedad SaaS refiere a “la cantidad de recursos que el servicio ha utilizado para alojar componentes funcionales en relación a la cantidad de recursos solicitados para tales fines”, su incorporación como descriptor de la subcaracterística de calidad elegida posibilita dar cuenta del nivel de escalabilidad que ha tenido el servicio de software durante su ejecución.

Finalmente, como parte de la subcaracterística “Reusabilidad” (incluida en la jerarquía de la característica “Mantenibilidad”) se incorporaron tres atributos de calidad SaaS, a saber: “Uniformidad en las Características Funcionales” (*functional feature commonality*), “Uniformidad en las Características No Funcionales” (*non-functional feature commonality*) y “Personalización de Servicios” (*service customizability*). Debido a que la semejanza entre características (tanto funcionales como no funcionales) facilita el reúso de componentes y, considerando que un alto grado de personalización da

lugar a un mayor reúso del servicio, es posible afirmar que estas propiedades SaaS refieren a aspectos de calidad vinculados a la subcaracterística “Reusabilidad”.

6.3 Métricas para Evaluar la Calidad en Servicios de Software

De acuerdo con (Scalone 2006), y conforme lo definido en la especificación de los modelos de calidad, toda jerarquía de conceptos vinculados a propiedades de calidad debe, de una forma u otra, estar vinculada a un conjunto de métricas que permitan su relevamiento. Es decir, junto con la especificación de un modelo de calidad se deben incluir las métricas asociadas (al menos) al nivel más bajo de descomposición. Complementariamente, pueden incluirse métricas adicionales de utilidad para relevar los siguientes niveles de la jerarquía.

En este contexto, a fin de complementar la definición del modelo de calidad SaaS detallado en la sección precedente, se define un conjunto de métricas de software útiles para el relevamiento de los aspectos de calidad identificados. Las métricas que permiten evaluar la capa asociada a los servicios de software de una arquitectura basada en entornos de CC, se denominan *métricas SaaS*. Existen múltiples métricas aplicables a costos, planificación, gestión, desarrollo, promoción y adquisición de servicios de software (Singh, Bhagat y Kumar 2012), por lo que la selección de las métricas a ser usadas para relevar la calidad no es sencilla.

Teniendo en cuenta que el modelo de calidad SaaS prescribe el conjunto de atributos a ser medidos y, tomando en consideración que muchos de estos atributos coinciden (total o parcialmente) con atributos definidos para productos de software tradicionales, existe un subconjunto de métricas predefinidas que puede ser utilizado como base para la especificación del conjunto de métricas deseado. En este sentido, resulta necesario identificar únicamente el subconjunto de métricas asociadas a la evaluación del SaaS en sí mismo (las cuales actuarán como complemento de las métricas tradicionales). Además, dada la naturaleza *mixta* del modelo de calidad

detallado, el conjunto final de métricas de software identificadas es susceptible de ser ampliado con nuevas métricas según la necesidad de quien lo utilice.

(METRICAS SAAS) *Métricas de software que permiten evaluar la capa asociada a los servicios de software de una arquitectura de producto de software desplegada sobre un esquema de computación en la nube.*

Por este motivo, se define un conjunto de métricas SaaS como base para la evaluación de los atributos de calidad relacionados con productos de software diseñados como servicios en la nube. Además, se detalla una estrategia de agrupamiento por medio de la cual es posible integrar las mediciones obtenidas a nivel de atributos, a fin de obtener un relevamiento de las propiedades de calidad ubicadas en los niveles superiores de la jerarquía de conceptos propuesta en el modelo de calidad SaaS.

6.3.1 Definición de Métricas de Software

Comportamiento Temporal Percibido por el Usuario (Time Behavior from User Perspective)

Métrica definida como “la relación que existe entre el tiempo de ejecución y el tiempo total transcurrido desde que el servicio fue invocado hasta que finaliza su operación”.

La Ecuación 6.1 presenta su fórmula de cálculo. El resultado de dicha ecuación varía en el rango [0,1]; donde un valor de 1 indica que el SaaS tiene un alto nivel de eficiencia temporal de acuerdo a lo percibido por el usuario.

$$TBU = \frac{ET}{TSIT} \qquad (6.1)$$

donde:

ET = *execution time* = Tiempo de procesamiento de una operación[1].

TSIT = *total service invocation time* = Tiempo transcurrido desde que se realizó la solicitud de la operación hasta que emitió su respuesta.

Uso de Recursos de Hardware (Hardware Resources Utilization)

Métrica de servicio de software cuya definición establece "la relación existente entre los recursos asignados y los recursos predefinidos".

De acuerdo con esta definición, debe entenderse por "recursos asignados" a aquellos recursos de hardware que afectivamente han sido ocupados por el servicio durante su ejecución. Por su parte, el término "recursos predefinidos" hace referencia a los recursos de hardware que, aunque han sido asignados al servicio para que los ocupe durante su ejecución, no han alojado ningún componente de procesamiento durante todo el tiempo en el cual el servicio estuvo activo.

La Ecuación 6.2 muestra la fórmula de cálculo asociada a esta métrica. Como puede observarse, el valor resultante de dicha operación varía en el rango [0,1]. Un valor de 1 indica que el SaaS tiene un alto nivel de utilización de recursos de hardware.

$$HRU = \frac{AR}{PR} \qquad (6.2)$$

donde:

AR = *amount of allocated resources* = Cantidad de recursos asignados.

PR = *amount of pre-defined resources* = Cantidad de recursos de hardware predefinidos.

[1] ET = TSIT – WT donde WT = *waiting time* = tiempo de espera para la invocación de una solicitud.

Precisión en Respuestas (Replies Accuracy)

Esta métrica SaaS refiere al "grado en el cual la respuesta a una solicitud de usuario es correcta". Se considera que una respuesta es correcta si el sistema provee lo que el usuario desea en un período de tiempo específico.

La Ecuación 6.3 presenta su fórmula de cálculo. Su resultado varía en el rango [0,1], donde un valor de 1 indica que el servicio es totalmente preciso.

$$RA = \frac{CR}{TR} \qquad (6.3)$$

donde:

CR = *number of correct responses* = Cantidad de respuestas correctas emitidas por el servicio[2].

TR = *total number of requests* = Cantidad total de solicitudes que han sido ingresadas al servicio.

Robustez del Servicio (Service Robustness)

Métrica definida como "la relación entre el tiempo total de operación y el tiempo que el que el servicio ha estado disponible para ser invocado".

La Ecuación 6.4 define su fórmula de cálculo en base a un valor que varía en el rango [0,1]. Un valor cercano a 1 indica que es altamente probable que, en un instante de tiempo dado, el sistema se encuentre en servicio.

[2] CR = TR – IR donde IR = *number of incorrect responses* = cantidad de respuestas incorrectas. El valor de IR se corresponde con la cantidad de solicitudes que han provocado problemas en el servicio. Dado que una solicitud puede provocar un único problema (luego de este, no podrá ser respondida correctamente); el valor de la variable IR es igual a la cantidad de defectos que se provoquen en el servicio (*total number of faults*). Esto se debe a que los defectos actúan como marcador de las solicitudes que son respondidas de forma incorrecta.

$$SR = \frac{AT}{TT} \qquad (6.4)$$

donde:

AT = *available time* = Tiempo durante el cual es servicio puede ser invocado[3].

TT = *total time* = Tiempo total que el sistema estuvo operativo.

Cobertura de Tolerancia de Defectos (Coverage of Fault Tolerance)

Métrica de software que mide "la proporción de apariciones de defectos que no se corresponden a fallas".

En este caso, es importante destacar la diferencia entre *defecto* y *falla*. Un defecto (*fault*) es problema que permite que el sistema siga operativo, pero, da lugar a un comportamiento incierto del mismo. Por su parte, una falla (*failure*) es un defecto que deja al sistema no operativo, no permitiendo que el componente en cuestión funcione conforme lo esperado en futuras ejecuciones. Ante la ocurrencia de un defecto, se asume que el sistema no responderá correctamente en la ejecución actual (debido a que no estará operando en condiciones normales). Sin embargo, seguirá operativo una vez finalizada dicha ejecución.

La Ecuación 6.5 indica la fórmula de cálculo asociada a la métrica. Como puede observarse, su valor varía en el rango [0,1]. Un valor de 1 indica que el servicio tiene la habilidad de continuar operando aunque se presenten múltiples defectos.

$$CFT = \frac{FNF}{TF} \qquad (6.5)$$

donde:

FNF = *number of faults without becoming failures* = Cantidad de defectos que no causan fallas.

TF = *total number of faults* = Cantidad total de defectos.

[3] AT = TT – FT donde FT = *service failure time* = tiempo total en el cual el sistema no puede ser invocado debido a una falla en cualquiera de sus servicios.

(DEFECTO) *Problema que permite que el sistema o servicio continúe operativo, dando lugar a un comportamiento incierto del mismo.*

(FALLA) *Defecto que deja al sistema no operativo, no permitiendo que el componente en cuestión funcione conforme lo esperado.*

Cobertura de Recuperación de Fallas (Coverage of Failure Recovery)

Esta métrica mide "la proporción de fallas reparadas en un período de tiempo dado".

La Ecuación 6.6 muestra su fórmula de cálculo. El valor resultante varía en el rango [0,1]; donde un valor cercano a 1 indica que el SaaS tiene una alta tasa de reparación.

$$CFR = \frac{RF}{Tf} \qquad (6.6)$$

donde:

RF = *number of failures remedied* = Cantidad de fallas reparadas.
Tf = *total number of failures* = Cantidad total de fallas[4].

Cobertura de Escalabilidad (Coverage of Scalability)

Métrica SaaS que evalúa "la cantidad promedio de recursos asignados en relación a la cantidad de recursos pedidos como parte de una solicitud de recursos".

[4] Tf = TF – FNF donde TF = *total number of faults* = cantidad total de defectos y FNF = *number of faults without becoming failures* = cantidad de defectos que no causan fallas. Esto se debe a que, sobre el total de defectos, el total de fallas es el subconjunto obtenido luego de quitar los defectos que no se convierten en fallas.

La Ecuación 6.7 presenta su fórmula de cálculo, donde el valor resultado varía en el rango [0,1]. Un valor de 1 indica que todos los recursos pedidos han sido otorgados en respuesta a todas las solicitudes realizadas.

Similitud Funcional (Functional Commonality)

Esta métrica queda definida como "el ajuste entre los requerimientos planteados al formular el servicio y las características funcionales diseñadas".

La Ecuación 6.8 indica su fórmula de cálculo cuyo valor resultado varía en el rango [0,1]. En este caso, un valor de 1 revela que todas las características funcionales se vinculan con todos los requerimientos contemplados para el dominio.

$$COS = \frac{1}{k} \times \sum_{i=1}^{k} \frac{AR_i}{TRR_i} \qquad (6.7)$$

donde:

k = Cantidad de solicitudes realizadas para ampliar los recursos asignados.

AR_i = *amount of allocated resources of the* i^{th} *request* = Cantidad de recursos asignados para la i-ésima solicitud.

TRR_i = *total amount of requested resources of* i^{th} *request* = Total de recursos solicitados en la i-ésima solicitud.

$$FC = \frac{1}{n} \times \sum_{i=1}^{n} \frac{RFC_i}{Tr} \qquad (6.8)$$

donde:

n = Cantidad total de características funcionales.

RFC_i = *number of requirements applying the* i^{th} *functional feature* = Cantidad de requerimientos que se vinculan con la i-ésima característica funcional.

Tr = *total number of requirements analyzed in the domain* = Total de requerimientos analizados en el dominio.

Similitud No Funcional (Non-Functional Commonality)

Esta métrica es similar a la precedente, quedando definida como "el ajuste entre los requerimientos planteados al formular el servicio y las características no funcionales".

La Ecuación 6.9 expresa su fórmula de cálculo. Al igual que en el caso previo, su valor varía en el rango [0,1], siendo 1 el valor que indica que todas las características no funcionales se vinculan con todos los requerimientos contemplados en el dominio.

$$NFC = \frac{1}{m} \times \sum_{i=1}^{m} \frac{RNFC_i}{Tr} \qquad (6.9)$$

donde:

m = Cantidad total de características no funcionales.

$RNFC_i$ = *number of requirements applying the* i^{th} *non-functional feature* = Cantidad de requerimientos asociados a la i-ésima característica no funcional.

Tr = *total number of requirements analyzed in the domain* = Total de requerimientos analizados en el dominio.

Cobertura de Variabilidad (Coverage of Variability)

Métrica SaaS que indica "cuántos puntos de variación del dominio se encuentran incluidos en el servicio".

La Ecuación 6.10 presenta su fórmula de cálculo, donde el rango del valor resultante varía en el intervalo [0,1]. Valores cercanos a 1 sugieren que la cobertura del servicio en relación al dominio es alta.

$$CV = \frac{VP_{SaaS}}{VP_D} \qquad (6.10)$$

donde:

VP_{SaaS} = *number of variation points realized in the SaaS* = Cantidad de puntos de variación que existen en el SaaS.

VP_D = *number of variation points in the domain* = Cantidad de puntos de variación que existen en el dominio asociado al SaaS.

6.3.2 Agrupamiento de Mediciones basadas en Métricas SaaS

Dado que las métricas definidas en el apartado precedente se detallan al nivel más bajo de la jerarquía de propiedades de calidad, a fin de agrupar estos resultados en términos de un único valor que refiera a niveles superiores de aspectos de calidad genéricos, se propone utilizar una suma pesada. Esta estrategia (Ecuación 6.11) es susceptible de ser aplicada a múltiples niveles de descomposición (tantos como se desee), dando lugar a un enfoque integral de medición.

Es importante destacar que el conjunto de métricas SaaS detalladas en el apartado previo han sido definidas en el intervalo [0,1]. Esto garantiza un nivel de coherencia conveniente entre las mediciones, permitiendo afectarlas por un coeficiente (denominado W) específicamente definido de acuerdo a la relevancia de la propiedad de calidad en relación al nivel de la jerarquía bajo estudio.

Bajo este esquema, todas las mediciones realizadas (tanto las basadas en las métricas como las obtenidas por la suma pesada de las propiedades del nivel inferior), deben ser interpretadas como niveles de aceptación.

$$M_N=\{\sum_{i=1}^{k} W_{i,N-1}\times M_{i,N-1} \mid (\sum_{i=1}^{k} W_{i,N-1})=1\ \forall N,\ (P_N,M_N)\wedge(P_N,P_{i,N-1})\wedge(P_{i,N-1},M_{i,N-1})\ \forall i\} \quad (6.11)$$

donde:

M_N = Medición en el nivel N.

k = Cantidad total de mediciones tomadas en el nivel N-1.

$W_{i,N-1}$ = Peso asignado a la i-ésima medición del nivel N-1.

$M_{i,N-1}$ = I-ésima medición del nivel N-1.

(P_X,M_X) = Propiedad de calidad ubicada en el nivel X de la jerarquía, la cual se encuentra vinculada a la medición M_X.

$(P_N,P_{i,N-1})$ = Relación entre una propiedad de calidad del nivel N y la i-ésima propiedad de calidad del nivel N-1.

6.4 Definición de un Esquema de Calidad para Servicios Web

De acuerdo con (Dromey 1995), no se dará un aumento significativo de la calidad del software hasta que exista un modelo global de calidad aplicable a todos los productos de software disponibles. Sin embargo, en la actualidad, la variedad de productos de software existentes dificulta el estudio y análisis de la calidad desde una perspectiva global. Las diversas características que definen cada tipo de producto de software conllevan a múltiples propiedades de calidad que no siempre son compatibles entre productos. En este sentido, aunque los modelos de calidad y las métricas de software constituyen herramientas útiles para el estudio de la calidad, su utilidad práctica se evidencia únicamente cuando se considera un producto de software específico bajo análisis.

Tal como se ha establecido en el capítulo *"Calidad en Productos de Software"*, la construcción de un esquema de calidad posibilita la documentación de la información de calidad asociada a un producto de software específico. Tomando en consideración que se ha definido un modelo de calidad para SaaS y un conjunto de métricas aplicables para la evaluación de propiedades ubicadas en los distintos niveles de la jerarquía de calidad, las relaciones entre ambos elementos sientan las bases requeridas

para especificar un *esquema de calidad genérico* de utilidad para la evaluación de la calidad en entornos de servicios de software.

El esquema de calidad resultante es formalizado haciendo uso de los conceptos y de las relaciones que forman parte de la ontología QSO (detallada en el apartado *"Ontología para la Especificación de un Esquema de Calidad"*). De esta manera, se establecen los aspectos de interés que deben ser relevados para determinar la calidad de un SaaS. Además, se fija la forma en la cual estos aspectos deben evaluarse en relación a otras propiedades de calidad.

6.4.1 Ontología QSO: Integración del Modelo de Calidad SaaS*

Teniendo en cuenta que se ha definido un nuevo modelo de calidad para servicios de software, es necesario ampliar el conjunto de elementos incluidos en el modelo semántico que representa el dominio de calidad a fin de incorporar las adaptaciones realizadas.

En este sentido, haciendo uso de la esquematización diseñada en la Figura 6.1, se incorporaron elementos referidos al diseño del modelo de calidad sobre el modelo semántico descripto en el apartado *"Modelo Semántico 1: Modelo de Calidad de Producto de Software"*. Dado que, como resultado del estudio de las propiedades de calidad SaaS, se han identificado tanto nuevas subcaracterísticas como nuevos atributos y, considerando que los atributos no forman parte del modelo semántico de calidad; únicamente se han incorporado los elementos requeridos para adicionar adecuadamente la subcaracterística "Escalabilidad" (*Scalability*). Esta incorporación se ha realizado siguiendo el conjunto de transformaciones detalladas en la Tabla 5.2.

De acuerdo con estos lineamientos, toda subcaracterística debe ser incorporada al modelo como un concepto. Luego, se incorpora el concepto *Scalability*. Dado que este concepto representa parte de una jerarquía de elementos, se modela además la relación asociada "es-descompuesto-en" (*is-decomposed-in*). Esta relación queda

delimitada entre la característica *Reliability* y la nueva subcaracterística incorporada, quedando definida como *is-decomposed-in-scalability*. Además, dada la naturaleza del concepto *Scalability*, se lo vincula al concepto *Subcharacteristic* a fin de representar -por medio de una relación "es-un" (*is-a*)- su pertenencia a un tipo de elemento.

En relación a las reglas SWRL, dado que toda característica queda descompuesta en sus subcaracterísticas (sin importar el caso sobre el cual se esté trabajando), una nueva regla similar a la propuesta en la Ecuación 5.1 fue definida. Esta regla se muestra en la Ecuación 6.12, donde se definen dos individuos "?x" e "?y" junto con los predicados "Reliability()" e "isDecomposedInScalability ()" que representan la característica confiabilidad y su relación con la subcaracterística escalabilidad. Luego, por medio de esta regla, se establece que todas las instancias de la subcaracterística *Stability* deben asociarse a la característica *Reliability* por medio del predicado "contains()". Dado que el predicado "belongs()" se deriva de la relación "contains()" (Ecuación 5.2), no fue necesario realizar una definición explícita.

$$\text{Reliability}(?x) \wedge \text{isDecomposedInStability}(?x,?y) \rightarrow \text{contains}(?x,?y) \qquad (6.12)$$

La Figura 6.2 muestra una representación gráfica del modelo semántico obtenido luego de estas incorporaciones. Este modelo fue utilizado como parte de la ontología QSO (en reemplazo del modelo detallado en el apartado *"Modelo Semántico 1: Modelo de Calidad de Producto de Software"*), dando lugar a una variante de la ontología original. Esta variante fue denominada QSO*.

Implementación de la Ontología QSO*

Al igual que en el caso de la ontología QSO, el diseño propuesto para QSO* fue implementado utilizando la herramienta Protégé.

Teniendo en cuenta que los modelos semánticos que componen en la ontología QSO fueron implementados en archivos OWL independientes (a fin de maximizar su posibilidad de reúso y reemplazo en contextos diferentes al de los esquemas de calidad), la implementación de la ontología QSO* únicamente reemplaza el contenido del archivo asociado al modelo de calidad. Esta estrategia permite reutilizar el contenido ya definido, ampliando sus elementos para incorporar los nuevos componentes especificados para representar el modelo de calidad SaaS.

Luego, debido a que se respeta la definición de esquema de calidad presentada en la Ecuación 5.6, el documento final que importa los tres modelos semánticos requeridos para instanciar un esquema de calidad SaaS, mantiene la definición de las relaciones que actúan como vínculo entre los conceptos de las ontologías (definidas en la Figura 5.5). En base a este nuevo documento, es posible generar instancias de esquemas de calidad SaaS que sirvan como mecanismo de documentación y análisis de las propiedades de calidad relevantes en este tipo de productos.

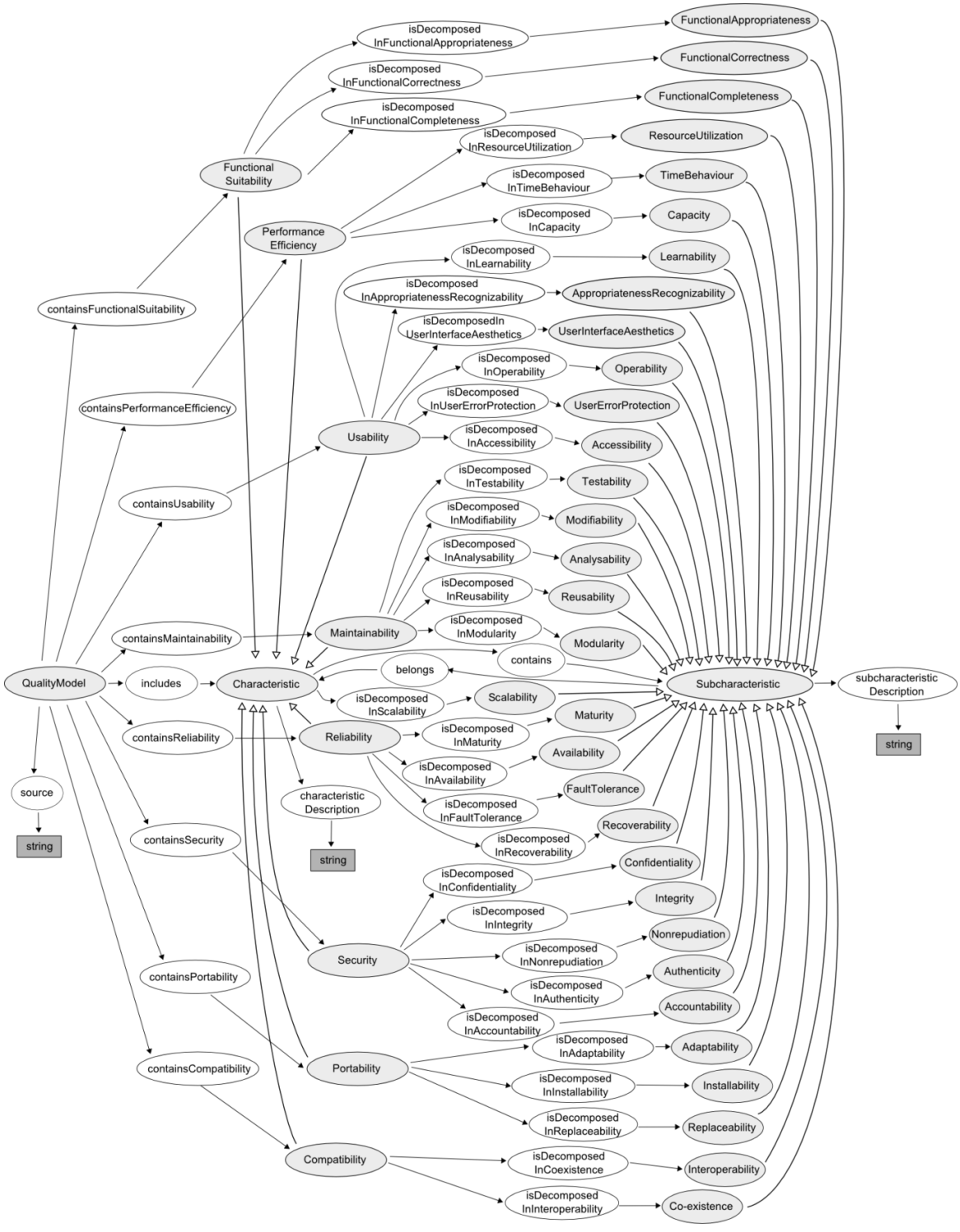


Figura 6.2. Modelo semántico de calidad de SaaS basado en ISO/IEC 25010.

6.4.2 Esquema de Calidad SaaS: Instanciación de la Ontología QSO*

Con el objetivo de especificar un esquema de calidad de servicios de software que combine de forma adecuada los atributos y métricas definidos en los apartados previos, en (Blas, Gonnet y Leone 2016a) se propone una instancia de la ontología QSO*. Esta instancia fue implementada en Protégé (como instancia de los archivos OWL detallados en el Capítulo 5) a fin de generar un modelo de documentación que permita analizar, posteriormente, su cobertura de acuerdo a la información disponible en un modelo específico (análisis detallado en el Capítulo 10, apartado *"Análisis de la Información asociada al Esquema de Calidad SaaS"*).

La Tabla 6.2 resume los principales componentes incorporados en la instancia creada para representar el esquema de calidad SaaS. Como puede observarse, para cada atributo derivado de la Tabla 6.1, se ha asociado una subcaracterística de calidad (conforme lo descripto en la sección *"Incorporación de Atributos de Calidad para Entornos SaaS"*). A su vez, para relevar cada par *(subcaracterística de calidad, atributo de software)*, se han vinculado una o más métricas SaaS (propuestas en el apartado *"Métricas para Evaluar la Calidad en Servicios de Software"*).

SUBCARACTERÍSTICA	ATRIBUTO DE SOFTWARE	MÉTRICA
Time Behavior	*Invocation Time*	*Time Behavior from User Perspective*
Resource Utilization	*Infrastructure Utilization*	*Hardware Resources Utilization*
Maturity	*Service Accuracy*	*Replies Accuracy*
Availability	*Robustness of Service*	*Service Robustness*
Fault Tolerance	*Service Stability*	*Coverage of Fault Tolerance*
		Coverage of Failure Recovery
Scalability	*Resource Coverage*	*Coverage of Scalability*

Reusability	*Service Customizability*	*Non-functional Commonality*
	Functional Feature Commonality	*Functional Commonality*
	Non-Functional Feat. Commonality	*Coverage of Variability*

Tabla 6.2. Resumen del esquema de calidad SaaS instanciado.

6.5 Ventajas y Desventajas del Esquema de Calidad Web

Mantener la calidad a lo largo del proceso de desarrollo de servicios de software es una tarea difícil. Frecuentemente, debido a que no existen mecanismos de soporte para las decisiones de calidad, la identificación de tales atributos se confunde y/o se pierde entre las diferentes etapas de desarrollo.

El esquema de calidad instanciado haciendo uso del modelo semántico QSO* proporciona una especificación simple y concisa de las relaciones existentes entre las principales propiedades de productos de software basados en SaaS y los elementos componentes del modelo de calidad de producto propuesto (diseñado en base al descripto en ISO/IEC 25010). Esta estrategia de documentación brinda un mecanismo de comunicación útil para el análisis de la calidad SaaS, facilitando tanto el entendimiento como la definición del conjunto de datos que conforman la información de calidad relativa a servicios de software basados en la nube (tanto nuevos como existentes).

El conjunto de propiedades de calidad identificadas para servicios de software es susceptible de ser ampliado con nuevas propiedades conforme se lo requiera. Lo mismo ocurre con el conjunto de métricas vinculadas a los atributos de calidad especificados. En este sentido, el esquema de calidad instanciado debe entenderse como un marco de trabajo básico que puede ser ampliado en cualquiera de sus

dominios a fin de mejorar la descripción de los aspectos de calidad asociados al servicio web.

Además, dado que el esquema propuesto surge como instancia de los conceptos incluidos en la ontología, es posible generar nuevas interpretaciones a su contenido una vez identificado el conjunto de valores asociados a las métricas propuestas. En este sentido, aunque la ontología no contempla la posibilidad de almacenar el valor resultante del proceso de medición como parte del documento instanciado, dicho valor puede ser almacenado en un documento complementario a fin de obtener mediciones referidas al grado de cumplimiento de las subcaracterísticas/características identificadas como parte del modelo de calidad. En este contexto, teniendo en cuenta que se propone una estrategia para agrupar a nivel N los valores de métricas relevados al nivel N-1 (apartado *"Agrupamiento de Mediciones basadas en Métricas SaaS"*), es posible utilizar la descripción del esquema en las distintas etapas del proceso de desarrollo a fin de analizar (evolutivamente) el estado de la calidad esperada en el servicio final. Sin embargo, para este tipo de análisis es necesario, previamente, contar con la información requerida para completar el relevamiento (es decir, no puede analizarse el grado de calidad en el nivel N si no se poseen datos sobre las mediciones de calidad de todos los elementos definidos en el nivel N-1). De esta manera, el esquema de calidad diseñado e instanciado da lugar a un estudio sistemático de productos de software basados en servicios brindando, además, la posibilidad de comparar resultados de mediciones de calidad tomando como base criterios de decisión uniformes.

Complementariamente, debido a que el esquema de calidad SaaS se deriva de la ontología QSO* (la cual a su vez se deriva de la ontología QSO), se mantienen sus beneficios y limitaciones (descriptos en el apartado *"Beneficios y Limitaciones del Uso de la Ontología QSO"*).

Conclusiones

Los esquemas de calidad dan lugar a nuevas estrategias de documentación para la información de calidad básica asociada a un producto de software específico. Para esto toman en consideración que la existencia de atributos de calidad conlleva a la necesidad de especificar métricas que faciliten su determinación en relación a modelos de calidad predefinidos. En este contexto, permiten explicitar los vínculos existentes entre los distintos atributos, las entidades que estos abarcan y la influencia de los mismos en la calidad resultante del producto o servicio.

En este capítulo se ha presentado un esquema de calidad que sintetiza las relaciones existentes entre los diferentes aspectos de calidad de servicios de software y las métricas disponibles para su relevamiento. Se muestra, por medio de una instanciación de la ontología QSO, la forma en cual estos elementos convergen en un único esquema de calidad genérico aplicable a productos de software basados en servicios en la nube. La definición del esquema resultante se basa en un modelo de calidad de producto de software existente, el cual es ampliado y modificado de acuerdo a características específicas de este tipo de servicios. Esta definición no es taxativa, por lo que el modelo de calidad propuesto puede ser ampliado con nuevas características, subcaracterísticas y/o atributos, mientras que el conjunto de métricas sugeridas puede ser mejorado y/o modificado de acuerdo a necesidades específicas.*

De esta manera, el esquema de calidad SaaS diseñado proporciona un conjunto de lineamientos básicos referidos a los principales aspectos de calidad a ser relevados en relación a servicios de software, proponiendo además estrategias de medición para su relevamiento.

Parte III

Diseño Arquitectónico de Software en la Nube

Capítulo 7. Arquitecturas de Computación en la Nube para Aplicaciones Web

Múltiples autores utilizan un diseño arquitectónico basado en un modelo de capas para la representación de los entornos de computación en la nube (Hu y colab. 2011; Khan y colab. 2013; Fehling y colab. 2014). En este capítulo se analizan algunas de las estructuras existentes, proponiendo una abstracción genérica de los componentes involucrados en el diseño de este tipo de entornos. Se presenta además el concepto de "sistemas de sistemas" como estrategia fundamental para el estudio de los entornos de computación en la nube. Este concepto es aplicado junto con la metodología de co-diseño a fin de estructurar un conjunto de actividades que contribuyen a agilizar el proceso de diseño de arquitecturas en la nube. Este proceso sienta las bases requeridas para la construcción de diseños arquitectónicos de aplicaciones de software basadas en la web.

7.1 Arquitecturas de Capas para Computación en la Nube

El estilo arquitectónico basado en capas corresponde a un patrón arquitectónico altamente difundido (Bass, Clements y Kazman 2012). En este estilo, la arquitectura del sistema de software queda modelada por una descomposición jerárquica funcional de sus elementos componentes en base a un conjunto de capas predefinidas. Las dependencias entre capas se controlan de forma tal que, en cada capa, solo se pueda acceder de forma directa a la capa de nivel inferior inmediato. Es decir, los

componentes residen sobre capas funcionales separadas y el acceso es permitido únicamente entre elementos de la misma capa y/o con su inmediata inferior (no permitiéndose el acceso entre capas no consecutivas).

Normalmente, las arquitecturas asociadas a los entornos de CC se definen haciendo uso de este tipo de patrones ya que permiten definir aplicaciones distribuidas con un bajo nivel de acoplamiento. Por naturaleza, los entornos de computación en la nube corresponden a grandes infraestructuras distribuidas que poseen múltiples recursos de tecnología de la información. Por lo tanto, las aplicaciones de software que se ejecutan sobre estos recursos deben descomponerse en un conjunto de componentes de aplicación individuales, los cuales sean aptos para ser distribuidos en los recursos disponibles del entorno (Fehling y colab. 2014). Sin embargo, dado que en un entorno de CC la cantidad de recursos de infraestructura varía constantemente, la cantidad de dependencias entre componentes debe minimizarse. Esto implica lograr un bajo acoplamiento a fin de reducir el impacto de posibles fallas en los componentes de aplicación y simplificar las tareas de asignación/remoción de recursos en el entorno.

(ESTILO ARQUITECTÓNICO BASADO EN CAPAS PARA CC) *Permite dividir la funcionalidad que debe proveer una aplicación de software web en múltiples componentes que pueden ser escalados de forma independiente y, al mismo tiempo, posibilita el uso de mecanismos de comunicación como elementos intermediarios a fin de separar las funcionalidades de los aspectos relacionados con los proveedores de servicios.*

Además, debido a que los entornos de CC se fundamentan en un modelo de negocio basado en servicios (Zhang, Cheng y Boutaba 2010), cada capa propuesta siguiendo este estilo de diseño arquitectónico es susceptible de ser implementada como un servicio a la capa superior.

7.1.1 Enfoques de Diseño Existentes

Diseño Basado en Servicios

Este diseño arquitectónico se presenta en (Modi y colab. 2012). En este caso se propone una arquitectura de alto nivel diseñada de acuerdo a los servicios que se ofrecen como parte del entorno de CC (Figura 7.1).

La arquitectura propuesta sigue un enfoque ascendente en el cual se identifican tres tipos de servicios, a saber:

i) *Infrastructure-as-a-Service (IaaS)*: A nivel de infraestructura se entrega poder de cómputo, el cual queda definido en términos de consumo de CPU y asignación de memoria.
ii) *Platform-as-a-Service (PaaS)*: Sobre el nivel de infraestructura, se encuentra el nivel de plataforma. Este nivel entrega un entorno de trabajo apropiado para el desarrollo de aplicaciones.
iii) *Software-as-a-Service (SaaS)*: Corresponde al nivel de aplicación (nivel superior del diseño). En este nivel se entrega software tercerizado a través de Internet (eliminando la necesidad de mantenimiento).

La ventaja que provee este esquema a nivel de aplicación es que los usuarios finales pueden hacer uso de los productos de software que se ejecutan en sitios remotos por medio del uso de ASP (Application Service Providers). De esta manera no necesitan comprar ni instalar software en sus equipos personales.

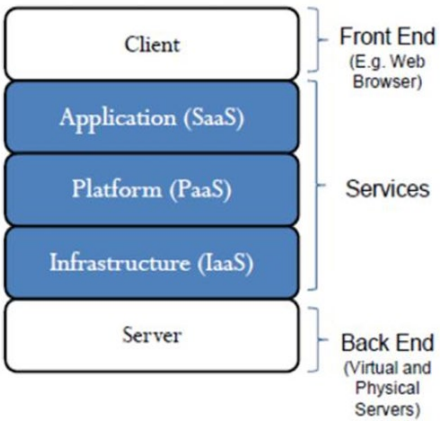


Figura 7.1. Arquitectura de computación en la nube basada en servicios[1].

Arquitectura Abierta para Computación en la Nube

En (Zhang y Zhou 2009) se propone una arquitectura para CC unificada, escalable y reusable que permite compartir cualquier tipo de recurso disponible. Su diseño se basa en un conjunto de principios fundamentales, los cuales han sido utilizados para definir su estructura en base a capas y bandas (Figura 7.2).

Del conjunto de principios identificados por los autores, se destacan:

i) *Administración integrada*: La arquitectura debe soportar la administración del entorno de CC a fin de proveer un contexto adecuado para el consumo/exposición de los recursos compartidos en el ambiente. Por ejemplo, los proveedores exponen las interfaces de interacción de sus operaciones internas y las capacidades de sus productos; mientras que los clientes simplemente utilizan los servicios web.

ii) *Virtualización de la infraestructura*: La arquitectura debe facilitar la virtualización de los recursos de tecnología de la información (como ser, por ejemplo, la administración de imágenes de software, la virtualización de los distintos recursos de hardware y el empaquetado de aplicaciones

[1] Tomado de (Modi y colab. 2012).

heredadas). En este contexto, existen dos enfoques complementarios que dan lugar a la virtualización en un ambiente de CC, a saber: *i)* virtualización de hardware (permite agregar o remover equipos sin afectar el funcionamiento de los equipos restantes), y *ii)* virtualización de software (uso de imágenes y de tecnología de virtualización de código que da lugar a la obtención de software compartido). Ambos enfoques deben ser contemplados a nivel de arquitectura.

iii) *Asignación de recursos extensible*: Un servicio de asignación de recursos extensible es la característica distintiva de un sistema de CC. Sin extensibilidad, la arquitectura sólo puede soportar cierto tipo de recursos compartidos.

iv) *Ofertas de CC configurables*: Las ofertas son los productos o servicios finales que provee una plataforma de CC. La mayoría de estas ofertas se acceden por medio de navegadores web. Las interfaces web han probado ser un efectivo canal de comunicación para que los usuarios de CC interactúen con los proveedores de los distintos tipos de servicios. Luego, el diseño propuesto para la arquitectura de CC debe garantizar que tales ofertas puedan ser accedidas por los usuarios desde diversos navegadores.

v) *Uniformidad en la representación de la información*: La representación de la información y el intercambio de mensajes de forma uniforme entre los recursos de tecnología de la información que componen un entorno de CC, son aspectos básicos que contribuyen a garantizar el correcto funcionamiento del modelo. Tales aspectos deben ser soportados por el diseño arquitectónico subyacente.

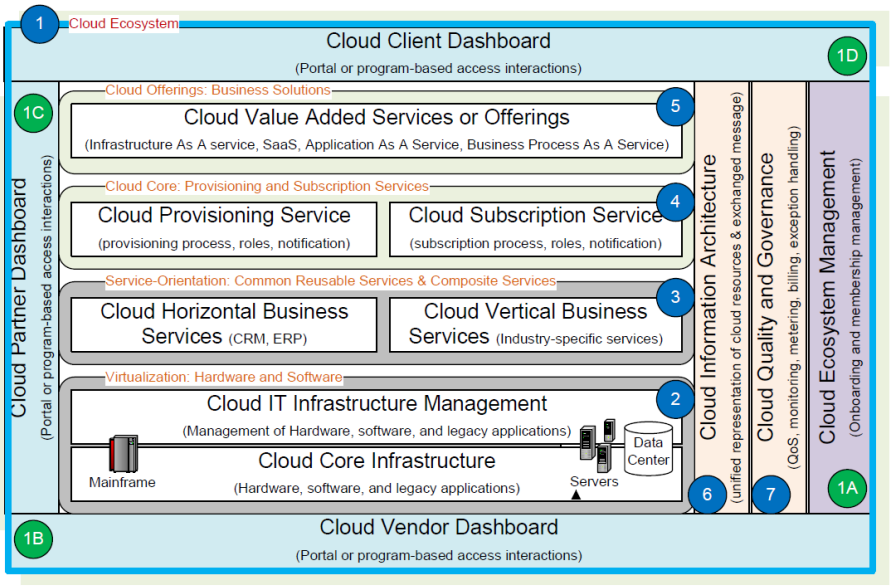


Figura 7.2. Arquitectura de computación en la nube abierta[2].

7.1.2 Arquitectura Genérica

Del análisis de los modelos descriptos en la sección precedente junto con el estudio de otros diseños alternativos (basados en capas) propuestos en la literatura (Rimal, Choi y Lumb 2009; Hu y colab. 2011; Fehling y colab. 2014), se observa que distintos autores proponen diferentes esquemas que varían tanto en el nombre y cantidad de capas, como así también en las responsabilidades vinculadas a cada una de ellas.

Dada la variabilidad que existe en los modelos planteados, en base a los lineamientos propuestos en (Khan y colab. 2013), se formula una única definición genérica para el diseño arquitectónico de entornos de CC que sintetiza las características fundamentales del paradigma (Figura 7.3). Esta arquitectura ha sido diseñada de acuerdo a una estructura de cinco capas, a saber:

[2] Tomado de (Zhang y Zhou 2009).

- *Hardware (Capa 1)*: Esta capa contiene los recursos de tecnología de la información sobre los cuales se ejecuta el entorno de CC. Es la responsable de lograr una asignación eficiente, rápida y fluida de estos recursos, brindando una base para el funcionamiento de las capas superiores. Por este motivo, se ubica en la parte inferior del modelo arquitectónico.
- *Software Kernel (Capa 2)*: Contiene un conjunto de funciones de software específicamente diseñadas para administrar los recursos de tecnología de la información ubicados en la capa subyacente (hardware). De esta manera, facilita la ejecución de los elementos de software ubicados en las capas superiores, brindando una estrategia de utilización eficiente de los recursos de hardware disponibles.
- *Software Infrastructure (Capa 3)*: Esta capa administra los recursos de red a las capas superiores, proporcionando un mecanismo de soporte estándar para la entrega de tecnología de la información bajo el formato de servicio.
- *Software Environment (Capa 4)*: Provee una plataforma de desarrollo para la construcción de aplicaciones web que serán ejecutadas sobre los recursos disponibles en el entorno de CC. Esta plataforma es utilizada por los programadores como complemento a las herramientas de desarrollo a fin de construir y ejecutar distintos tipos de aplicaciones web.
- *Application (Capa 5)*: Es la última capa de la estructura arquitectónica. En ella residen las aplicaciones web que se ejecutan sobre los recursos del entorno de CC, las cuales han sido desarrolladas haciendo uso de la plataforma situada en la capa previa (software environment). En este contexto, es la responsable de facilitar (a los usuarios finales) el acceso a las aplicaciones web.

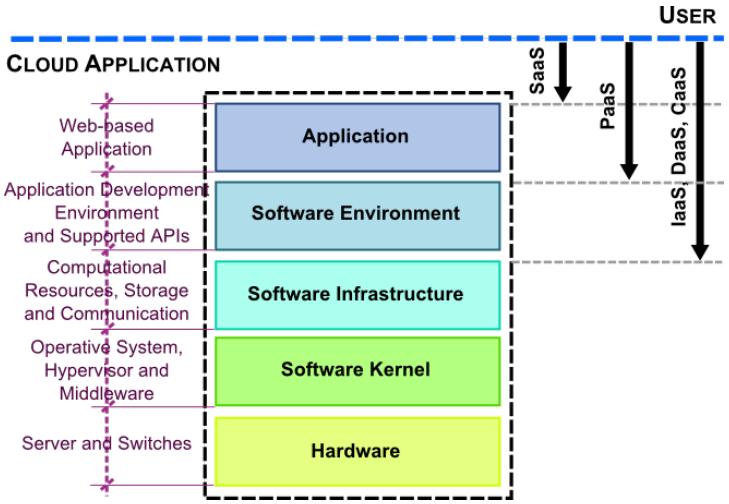


Figura 7.3. Arquitectura genérica para computación en la nube (modelo de capas).

Tal como se ha enunciado con anterioridad, el concepto detrás del modelo de CC es bajar el cómputo a los proveedores de recursos remotos. En este sentido, el proveedor de servicios en la nube es el responsable de ejecutar, administrar y actualizar (de acuerdo a los requerimientos de los usuarios) los recursos de tecnología de la información asociados al entorno. De esta manera, este modelo computacional permite que los usuarios utilicen los servicios bajo demanda, según su consumo, a través de Internet.

Sin embargo, a diferencia de otros esquemas computacionales, en un entorno de CC no se ofrece un único servicio. En este sentido, el paradigma brinda la posibilidad de ofrecer a sus usuarios un conjunto de servicios vinculados a las capas propuestas como base de su arquitectura.

Como parte del esquema propuesto en la Figura 7.3 se identifican cinco tipos de servicios, los cuales han sido definidos como: infraestructura como servicio (*infrastructure as a service, IaaS*), almacenamiento de información como servicio (*data storage as a service*), comunicación como servicio (*communication as a service, CaaS*), plataforma como servicio (*platform as a service, PaaS*) y software como servicio (*software as a service, SaaS*). Cada uno de estos servicios se logra ofreciendo a los usuarios

(personas u otros sistemas de software) un subconjunto de las capas incluidas en el diseño. Por ejemplo, cuando un usuario utiliza una aplicación web accede a un servicio de software (SaaS). En este caso, el término SaaS refiere a la facilidad con la cual se proporciona el acceso a aplicaciones de software web, bajo demanda, a través de Internet.

7.1.3 Abstracción de los Modelos basados en Capas

Tomando como referencia la arquitectura genérica descripta en la sección precedente, y teniendo en consideración el conjunto de capas que usualmente se incluyen en los esquemas arquitectónicos propuestos en la literatura, es posible abstraer el contenido del modelo de capas completo y generar un esquema de trabajo equivalente basado en dos grandes capas: *infraestructura* y *aplicación* (Figura 7.4). Bajo esta conceptualización, la capa de *infraestructura* (*infrastructure*) reúne el conjunto de capas del modelo genérico que brindan soporte a los recursos de tecnología de la información que son utilizados por los servicios que se ejecutan como parte de la capa de *aplicación* (*application*).

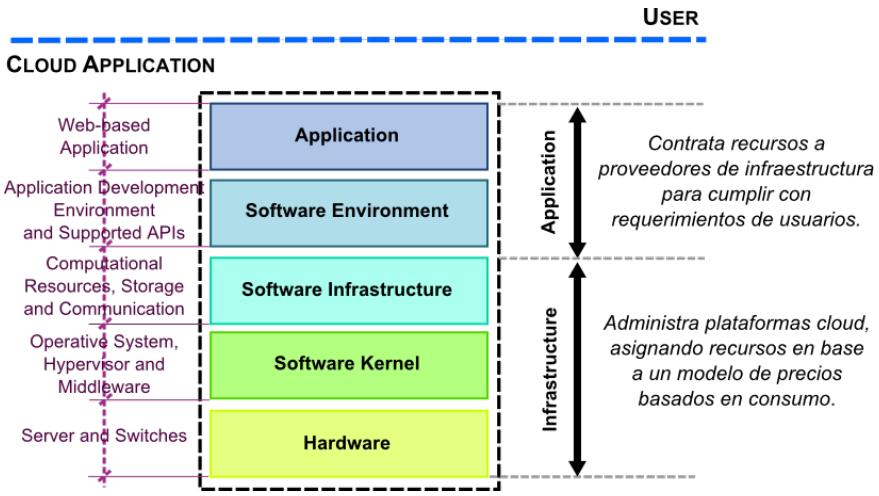


Figura 7.4. Abstracción del diseño basado en capas propuesto.

Esta abstracción permite analizar la definición e implementación de ambas capas de forma independiente ya que una misma aplicación puede ser ejecutada sobre diferentes infraestructuras y, al mismo tiempo, una infraestructura puede ejecutar diferentes tipos de aplicaciones. Sin embargo, ninguna de las capas abstractas es autónoma. Ambas capas requieren de su contraparte para lograr cumplir los requisitos globales (funcionales y no-funcionales) de los usuarios finales del entorno de CC.

Es importante destacar que el conjunto de servicios identificados como parte del modelo genérico sigue siendo válido en el modelo abstracto ya que:

- *SaaS* y *PaaS* quedan disponibles en la capa de aplicación (que engloba las capas 4 y 5 del modelo genérico).
- *IaaS*, *DaaS* y *CaaS* quedan disponibles en la capa de infraestructura (que engloba las capas 1 a 3).

Interpretación del Modelo Abstracto como Sistema de Sistemas

El entorno de CC modelado conforme la abstracción propuesta en el apartado previo puede ser analizado como un sistema compuesto de dos grandes sistemas, los cuales quedan definidos de acuerdo a sus componentes en términos de servicios de software (aplicaciones) y recursos de tecnología de la información (elementos de hardware e infraestructura). Luego, ambos componentes del sistema de CC son (a su vez) sistemas en sí mismos.

La teoría que respalda este tipo de construcciones para el estudio, diseño e implementación de sistemas se denomina “sistemas de sistemas”. Múltiples trabajos de investigación han aplicado esta teoría para el estudio de entornos de CC (Strowd y Lewis 2010; Dowell y colab. 2011; Karnouskos y Colombo 2011; Zio y Sansavini 2013). De acuerdo con esta perspectiva, el sistema de CC puede ser analizado como un sistema de sistemas (Zeigler y Sarjoughian 2013).

7.2 Sistemas de Sistemas (Systems of Systems)

El crecimiento y desarrollo evidenciado en los últimos años en el área de ingeniería de software y sistemas de información ha llevado a que múltiples autores aborden temáticas vinculadas a las capacidades de los sistemas (Yeh, Lee y Pai 2012; Kashkoush y ElMaraghy 2017; Aydiner 2017). En este contexto, un *sistema* es un grupo relacionado a nivel funcional, físico y/o de comportamiento de elementos básicos interdependientes que forman un todo unificado (Department of Defense 2008). Desde esta perspectiva, un sistema es visto como una combinación de elementos organizados que interactúan en busca del logro de uno o más objetivos definidos grupalmente (ISO/IEC/IEEE 24765 2010). Por su parte, se entiende por *competencia* a la habilidad de alcanzar un efecto deseado bajo condiciones y estándares específicos a fin de realizar un conjunto de tareas mediante la combinación de diferentes formas y medios (Department of Defense 2012).

A mediados de 1950 surge el concepto de *sistemas de sistemas* (SoS por sus siglas en inglés, *systems of systems*) como perspectiva de análisis alternativa a los enfoques existentes (Nielsen y colab. 2015). De acuerdo con (Department of Defense 2004), este concepto refiere a un conjunto o arreglo de sistemas que se presenta cuando sistemas útiles e independientes son integrados en un único sistema mayor que entrega capacidades únicas. Bajo este punto de vista, tanto el sistema SoS como los sistemas que lo componen (denominados *sistemas componentes*) se corresponden con la definición tradicional de *sistema*. Esto se debe a que, en ambos casos, los sistemas constan de un conjunto de partes y relaciones, en base a las cuales queda definido un todo que es más grande que la mera suma de sus partes.

(SISTEMAS DE SISTEMAS - SYSTEMS OF SYSTEMS -) *Sistemas que se encuentran compuestos por sistemas independientes que actúan de forma conjunta, en dirección hacia un objetivo común, por medio de la sinergia que existe entre ellos (Nielsen y colab. 2015).*

7.2.1 Descomposición SoS vs Subsistemas-Sistemas-Suprasistema

Los niveles de organización o jerarquías que se definen en la teoría de sistemas buscan generar un orden conceptual por medio de la identificación de sistemas simples que conforman sistemas complejos. Un ejemplo de este tipo de organización se evidencia en la estructura *subsistema-sistema-suprasistema*. En esta estructura clásica se identifican tres niveles de organización en los cuales se ubica un subsistema, dentro de un sistema, que a su vez forma parte de un suprasistema de mayor tamaño.

De acuerdo con (ISO/IEC/IEEE 24765 2010), un *subsistema* es un sistema subordinado o secundario que forma parte de un sistema más grande. Mientras que un *sistema* es un conjunto ordenado de elementos interrelacionados que interactúan para lograr un objetivo; un *subsistema* es un conjunto de partes e interrelaciones que se encuentran de forma estructural y funcional dentro de un sistema mayor. En el caso de la relación *sistema-suprasistema* se aplica una definición similar.

En este contexto, la identificación de los sistemas que componen la jerarquía *subsistema-sistema-suprasistema* requiere del establecimiento de los límites que definen los sistemas asociados a cada uno de los niveles. Sin una clara definición de estos límites, es muy difícil establecer una distinción entre *subsistema*, *sistema* y *suprasistema*. En este punto es importante comprender que los *subsistemas* (al igual que los *suprasistemas*) son *sistemas*, por lo que la diferencia entre estos términos existe únicamente desde un punto de vista organizacional o jerárquico.

Luego, el uso de los términos *subsistema*, *sistema* y *suprasistema* es relativo a la posición del observador. Por este motivo, su aplicación varía de acuerdo al modelo mental que tenga el observador en relación al sistema bajo estudio.

Por su parte (tal como se ha establecido con anterioridad), el término SoS también refiere al concepto de *sistema*. Sin embargo, su definición no se elabora en base a niveles jerárquicos. En este caso, la definición se fundamenta en la persecución de un

objetivo global al mismo tiempo en que cada componente mantiene su objetivo individual. Desde esta perspectiva, cada uno de los sistemas involucrados no constituye necesariamente un subsistema a ser analizado por descomposición jerárquica. En su lugar, estos sistemas son vistos como piezas fundamentales del objetivo definido, las cuales actúan como vehículo en la persecución de un objetivo mayor.

Luego, los criterios utilizados para identificar el conjunto de *sistemas componentes* en un SoS, difieren del conjunto de criterios utilizado para determinar la descomposición *subsistema-sistema-suprasistema*. El enfoque que se tome como base de análisis para reconocer los componentes, determina la adecuación de los sistemas a cada una de las categorías.

7.2.2 Propiedades Fundamentales

Aunque todo SoS es un sistema, no todo sistema es un SoS. Para lograr una verdadera clasificación de un sistema como SoS se debe identificar un conjunto de propiedades (vinculadas tanto al sistema global como a sus sistemas componentes) en virtud de garantizar el cumplimiento de una serie de premisas.

La Tabla 7.1 resume el conjunto de propiedades propuesto en (Maier 1996) a fin de identificar un SoS. Como puede observarse, estas propiedades se dividen en grupos según el contexto en el cual deben ser evaluadas.

GRUPO	PROPIEDAD
Sistemas componentes	Independencia de operación
	Independencia de gestión
SoS	Comportamiento emergente
	Desarrollo evolutivo
	Distribución
Sistemas	Soporte basado en software

Tabla 7.1. Propiedades de los sistemas-de-sistemas en ingeniería de software.

Dentro del grupo *sistemas componentes* (es decir, aquellos sistemas que se analizan como parte del SoS) se identifican dos propiedades, a saber:

- *Independencia de operación*: Los componentes deben operar de forma independiente y autónoma, ejecutándose sobre sus propios recursos en virtud de cumplir una misión individual.
- *Independencia de gestión*: Los componentes deben ser gestionados de forma independiente, pudiendo evolucionar de formas no previstas al momento en el cual se unieron al SoS.

Por otra parte, dentro del grupo *SoS* (es decir, cuando se analiza el SoS como un todo), se detallan tres propiedades:

- *Comportamiento emergente*: Es posible que se presenten comportamientos en tiempo de ejecución que no hayan sido previstos en el diseño original.
- *Desarrollo evolutivo*: Los componentes evolucionan continuamente, lo que implica que el SoS también lo hace. Además, por su naturaleza, el SoS evoluciona debido a cambios en su entorno.
- *Distribución*: Los componentes se encuentran distribuidos (no necesariamente de forma geográfica).

Finalmente, en el grupo *sistemas* se identifican propiedades aplicables a todos los sistemas intervinientes de la estructura (es decir, tanto a SoS como a sus sistemas componentes). En este grupo se identifica la propiedad de *soporte basado en software* que refiere a la influencia del software en el diseño, construcción, implementación y/o evolución de todos los sistemas involucrados.

En este escenario, en (Nakagawa y colab. 2013) se definen cuatro lineamientos básicos para la identificación de un SoS. De acuerdo a estas premisas, un sistema de software puede ser clasificado como SoS si:

- Resulta de la interoperación de componentes independientes (tanto a nivel organizativo como a nivel de gestión), los cuales poseen misiones individuales pero, que al mismo tiempo, participan (de forma consiente o no) del cumplimiento de una misión global.
- Posee un desarrollo evolutivo que se da por medio de la adaptación de los sistemas componentes y/o de cambios en su entorno.
- Presenta comportamientos emergentes (que pueden o no haber sido diagramados en tiempo de diseño), los cuales surgen como resultado de la interacción de los sistemas componentes en tiempo de ejecución.
- Depende del software como tecnología de soporte para su diseño y desarrollo evolutivo.

7.2.3 Arquitecturas Dinámicas

Del conjunto de propiedades detalladas en el apartado precedente, se desprende la necesidad de definir los SoS desde un punto de vista dinámico. En este contexto, usualmente se utilizan arquitecturas dinámicas como base para garantizar una correcta estructuración de este tipo de sistemas (Fang y DeLaurentis 2014).

Desde esta perspectiva, según (Gonçalves 2016), la arquitectura del SoS es vista como una estructura dinámica que comprende los sistemas componentes (independientes), sus propiedades visibles de forma externa, sus relaciones y el conjunto de restricciones que guían su diseño y evolución (que se evidencia cuando del sistema emergen misiones, no siempre esperadas, en tiempo en ejecución).

Modelos y Arquitecturas de Software

En (Nielsen y colab. 2015) se utiliza el término *modelo* para referir a una descripción abstracta de un sistema de interés. Las abstracciones utilizadas al construir un modelo quedan determinadas por el propósito para el cual dicho modelo ha sido construido. Así, los modelos pueden ser utilizados para representar objetos del mundo

real o para describir sistemas en una etapa previa a su desarrollo (por ejemplo, durante las fases de diseño).

En ingeniería de software se utilizan diferentes clases de modelos durante el diseño, desarrollo y mantenimiento de sistemas. En el caso de SoS, los modelos descriptivos de los elementos existentes pueden combinarse con modelos de diseño de elementos que aún no han sido desarrollados a fin de dar lugar a nuevos esquemas. Siguiendo este lineamiento, los modelos pueden ser evaluados individualmente pero además, en una etapa posterior, pueden ser utilizados para evaluar aspectos estructurales, funcionales, de comunicación y de comportamiento del SoS bajo estudio.

Un modelo apropiado para este tipo de análisis es el diseño de la arquitectura de software ya que, complementado con otras técnicas, sirve como vehículo para la realización de distintos tipos de estudios asociados al sistema final. En el caso de arquitecturas dinámicas, para ganar confianza en que una arquitectura SoS respeta las propiedades requeridas, es fundamental tener un modelo preciso de componentes y conectores a partir de los cuales se construye el diseño (incluyendo tanto sus propiedades como la definición del entorno subyacente). De esta manera, se compensan diseños alternativos en etapas tempranas de desarrollo y, a su vez, se determinan los contratos requeridos entre los sistemas componentes y el sistema global (Nielsen y colab. 2015).

Arquitecturas de Software y Entornos de Computación en la Nube

En el apartado *"Interpretación del Modelo Abstracto como Sistema de Sistemas"* se ha detallado la forma en la cual se vincula el enfoque de SoS con los entornos de CC. En este caso, a partir de la arquitectura abstracta propuesta como base de análisis, se identifican dos sistemas que interactúan conforme las propiedades requeridas. Luego, la arquitectura del sistema SoS es, en sí misma, definida como un SoS.

De acuerdo a los lineamientos planteados en la sección precedente, es posible desarrollar una arquitectura SoS en base a un conjunto definido de elementos arquitectónicos (modelos de componentes y conectores). Dicho modelo debe permitir capturar las particularidades de los sistemas que componen el SoS, dando lugar a una definición adecuada de la forma en la cual tales sistemas interactúan.

Siguiendo esta propuesta como base para la definición de arquitecturas de CC, es posible diagramar un esquema de diseño arquitectónico en términos de un *sistema componente de software* y un *sistema componente de infraestructura*. Sin embargo, dadas las dificultades inherentes al proceso de diseño arquitectónico, como complemento al modelo de especificación es necesario definir un conjunto de pautas que guíen su aplicación. En este sentido, estas pautas no deben prescribir elementos específicos (el proceso de diseño es un proceso creativo que no debe verse limitado por los componentes y conexiones disponibles), sino que deben describir la forma en la cual deben combinarse las distintas clases de elementos componentes.

Luego, la definición de un proceso de diseño de arquitecturas para entornos de CC sienta las bases requeridas para ayudar al arquitecto en: *i)* la definición de los sistemas de software e infraestructura (componentes), y *ii)* la determinación de la forma en la cual deben establecerse las interacciones entre ambos sistemas como parte del entorno.

En el apartado *"Proceso para la Especificación de Arquitecturas CC"* se define el proceso de diseño propuesto como base para la construcción de arquitecturas de CC. Sin embargo, previo a su especificación, es necesario introducir el concepto de *co-diseño* (Butler 1994; Hild, Sarjoughian y Zeigler 2002).

7.3 Co-Diseño

El co-diseño refiere a la partición de un sistema bajo diseño en términos de sus componentes de software y hardware, de forma tal que cada uno de ellos pueda ser desarrollado de manera independiente y, luego, integrado en un único diseño que represente el sistema global. Este enfoque ha sido utilizado con éxito en el diseño, simulación y desarrollo de sistemas embebidos (Wolf 1994; Edwards y colab. 1997).

Su objetivo es lograr diseños de sistemas robustos haciendo énfasis en mejorar las interacciones hardware-software. Para esto, el enfoque brinda la libertad requerida para la construcción independiente de los diseños, permitiendo a los arquitectos de software realizar especificaciones según su nivel de conocimiento y experiencia. Tales especificaciones, en una etapa posterior, son integradas para dar con la especificación del sistema global. Luego:

- *PROP#1*: El sistema de software tiene una especificación propia.
- *PROP#2*: El sistema de hardware tiene una especificación propia.
- *PROP#3*: La interacción entre ambos sistemas se especifica en el sistema global.

De esta manera, la flexibilidad que provee el enfoque para la construcción de diseños independientes posibilita la realización de una tarea de diseño diferenciada para cada componente requerido. La especificación del sistema global (*PROP#4*) se obtiene como consecuencia de llevar a cabo un proceso de diseño arquitectónico basado en este enfoque (*PROP*#1 ∧ *PROP*#2 ∧ *PROP*#3 → *PROP*#4).

7.3.1 Diseño de Arquitecturas en la Nube combinando Co-Diseño y SoS

Haciendo uso de la perspectiva de co-diseño, es posible analizar los sistemas componentes de las arquitecturas de CC (derivados del estudio de las características SoS) de la siguiente manera:

- El *sistema componente de software* como *sistema de software* que brinda soporte a los servicios y aplicaciones web.
- El *sistema componente de infraestructura* como *sistema de hardware* que brinda soporte a los recursos de tecnología de la información.
- El *sistema de CC* como *sistema global*.

La Figura 7.5 esquematiza esta propuesta mostrando las principales relaciones entre los sistemas involucrados. De acuerdo con esta partición, se propone construir el diseño arquitectónico del sistema de CC en base a los siguientes lineamientos:

- Aplicar la *PROP#1* a nivel de capa de aplicación.
- Aplicar la *PROP#*2 a nivel de capa de infraestructura.
- Aplicar la *PROP#3* a nivel de entorno de CC.

Luego, tomando en consideración los fundamentos del enfoque de co-diseño, se obtiene la arquitectura de CC (*PROP#4*).

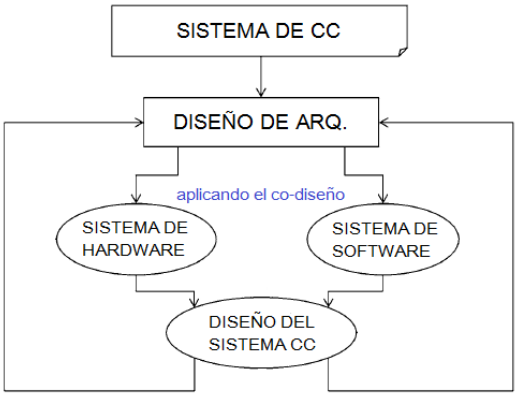


Figura 7.5. Esquematización del proceso de diseño arquitectónico propuesto[3].

[3] Adaptado de (Zeigler y Sarjoughian 2013).

7.4 Proceso para la Especificación de Arquitecturas en la Nube

Tal como se ha establecido con anterioridad, los entornos de CC vistos como SoS permiten la construcción de sus arquitecturas en base al enfoque de co-diseño. Si se toma como referencia un conjunto de componentes predefinidos y se establecen las conexiones requeridas entre estos, es posible definir una serie de lineamientos que contribuyan al diseño de la arquitectura de una aplicación web en un entorno de CC.

De acuerdo a lo expuesto en el apartado *"Desafíos e Inconvenientes"* existen dos grandes dificultades al diseñar arquitecturas para aplicaciones web: *i)* representación del diseño arquitectónico, y *ii)* dependencia de la infraestructura subyacente. El primer problema refiere a la ausencia de un mecanismo universal para la construcción de los diseños web, mientras que el segundo trata las dificultades inherentes a la separación de componentes entre infraestructura/aplicación.

En este contexto, el uso de la estrategia de co-diseño contribuye a la resolución de *ii)* ya que simplifica la definición de las arquitecturas en términos de dos diseños desarrollados de forma independiente (los cuales, luego, se integran en un único diseño). En el caso del problema *i)*, la identificación de *clases de componentes* comúnmente utilizados en este tipo de diseños contribuye a lograr una primera aproximación para la representación uniforme de dichas aplicaciones.

Luego, cuando las soluciones de *i)* y *ii)* se integran en un único conjunto de pasos, pautas y lineamientos se establecen las bases de trabajo necesarias para la definición de arquitecturas web. En los siguientes apartados se detalla la estructura de componentes generales utilizada como base para la definición de un proceso de diseño genérico destinado a la especificación de este tipo de arquitecturas.

En este punto es importante destacar que existen dos maneras de plantear la especificación de arquitecturas web de acuerdo a la forma en la cual se disponen los recursos de tecnología de la información, a saber:

i) *Recursos propios*: Quien define la estructura de los componentes de software también define la estructura de los componentes de hardware.

ii) *Recursos de terceros*: Se utilizan ofertas de proveedores de infraestructura para respaldar los recursos de tecnología de información requeridos en la aplicación.

En esta tesis se considera que las aplicaciones web a ser diseñadas se implementaran tomando como referencia el segundo enfoque (recursos de terceros). Esto se debe a que, usualmente, los sistemas de hardware suministrados por proveedores externos son diseñados en base a componentes optimizados que brindan un comportamiento estándar configurable de acuerdo a las necesidades de los clientes (en este caso, los desarrolladores de las aplicaciones web).

Luego, la mayoría de las aplicaciones web desarrolladas contratan diferentes tipos de servicios de hardware (denominados *ofertas de servicios*) para dar soporte a las funcionalidades de infraestructura requeridas en sus productos. Bajo este esquema de tercerización, el arquitecto de software no se ve obligado a diseñar las funciones de infraestructura.

7.4.1 Identificación de Componentes Arquitectónicos Generales

La Figura 7.6 presenta una clasificación de componentes arquitectónicos basada en los lineamientos propuestos en (Fehling y colab. 2014).

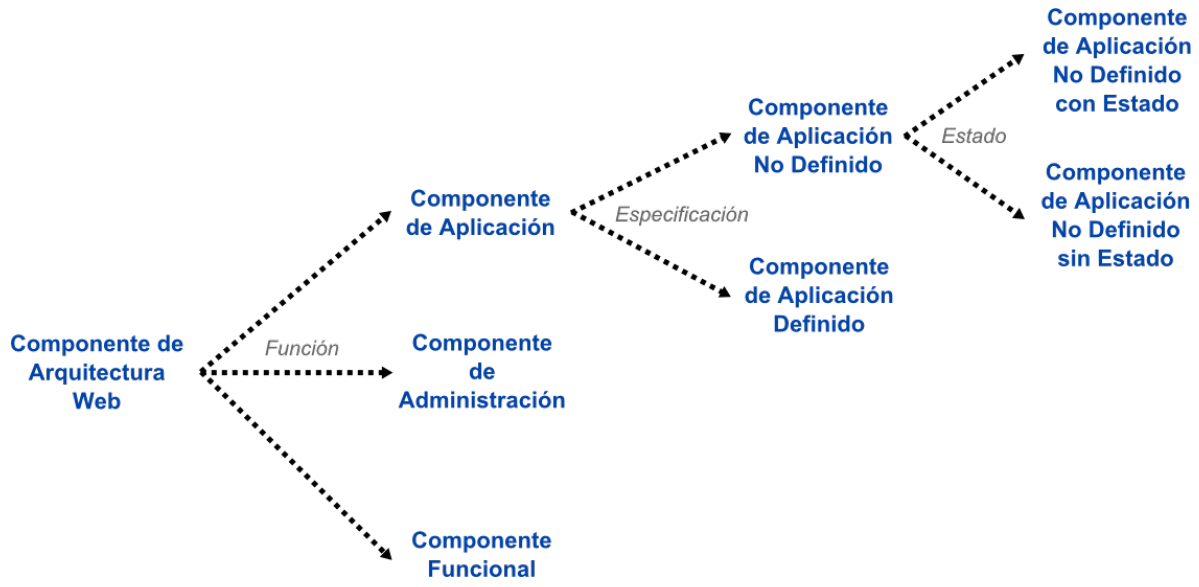


Figura 7.6. Clasificación de componentes para arquitecturas de aplicaciones web.

De acuerdo con este esquema, los componentes de las aplicaciones web se clasifican (según su función) en *componentes de aplicación*, *componentes de administración* y *componentes funcionales*. Un *componente de aplicación* refiere a un elemento definido a nivel arquitectónico con el objetivo de brindar soporte para el cumplimiento de las funciones requeridas por los usuarios. Un *componente de administración* corresponde a un elemento que permite gestionar las réplicas de los *componentes de aplicación* y controlar el correcto desempeño del software. Finalmente, un *componente funcional* define una tarea elemental (genérica) que posibilita la construcción de tareas más complejas por medio de su interacción con otros elementos del mismo tipo. En este último caso se refiere a una responsabilidad básica (función) por lo que, normalmente, el componente posee un único punto de entrada y un único punto de salida.

A su vez, los *componentes de aplicación* se dividen en dos clases según la forma en la cual son especificados:

- *Componente de aplicación definido*: Componente de aplicación que muestra un elemento comúnmente utilizado en la definición de las aplicaciones web (por ejemplo, un balanceador de carga).
- *Componente de aplicación no definido*: Componente de aplicación específico que contribuye a dar respuesta a uno o más requerimientos funcionales establecidos por el usuario (es decir, corresponde a un componente de dominio). Su especificación se basa en la interacción de *componentes funcionales*.

Finalmente, según la forma en la cual manejen la información asociada a su estado, los *componentes de aplicación no definidos* se clasifican en:

- *Componente con estado*: Las múltiples instancias del componente de aplicación sincronizan su estado interno para proveer un comportamiento uniforme a nivel de aplicación.
- *Componente sin estado*: El estado del componente de aplicación se maneja de forma externa, lo que facilita la creación de réplicas (nuevas instancias) consistentes, dando lugar a que la aplicación web tenga un mayor grado de tolerancia a fallos en relación a este componente.

7.4.2 Proceso de Diseño: Pasos, Pautas y Lineamientos Generales

Actividad para el Diseño de la Capa de Aplicación (Sistema de Software)

La Figura 7.7 esquematiza el conjunto de pasos a seguir para llevar a cabo la actividad propuesta con el objetivo de diseñar la arquitectura del sistema de software asociado al entorno de CC. En las siguientes subsecciones se detallen los objetivos y lineamientos aplicables para cada uno de los pasos identificados.

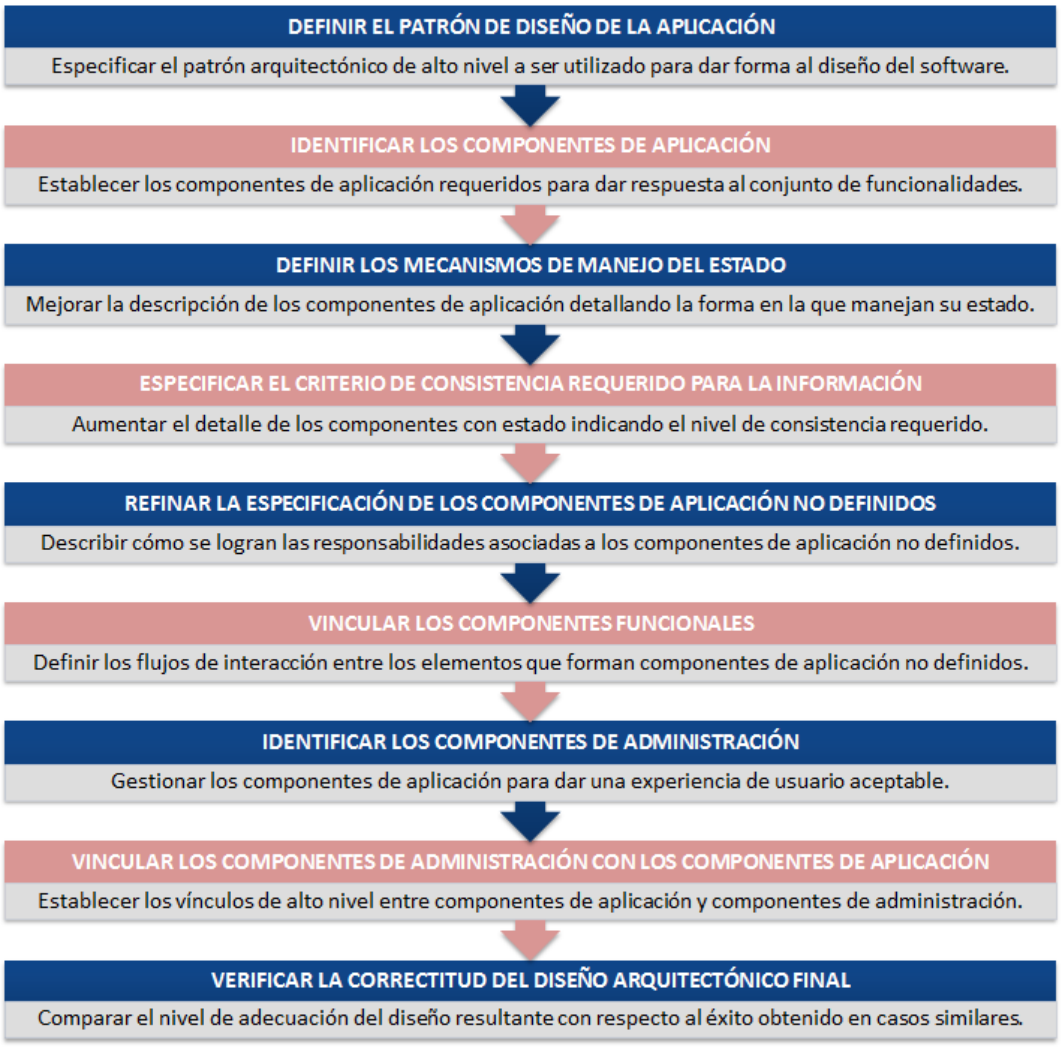


Figura 7.7. Pasos para el diseño del sistema de software (capa de aplicación).

Es importante destacar que, aunque se presenta un conjunto de pasos secuenciales, el arquitecto de software tiene la libertad de volver a un paso previo (no necesariamente inmediato) y retomar (a partir de este punto) la actividad el diseño en función de nuevas metas u objetivos.

Paso 1: Definir el Patrón de Diseño de la Aplicación

El objetivo de este paso es especificar el patrón arquitectónico de alto nivel a ser utilizado para dar forma al diseño del software.

En el caso de las aplicaciones web, existen dos tipos de patrones comúnmente utilizados, a saber:

- *Patrón "N-bandas"*: Se separa la lógica de las funcionalidades del manejo de la información a fin de escalar de forma independiente cada responsabilidad (banda). Usualmente las aplicaciones utilizan patrones de 2-bandas (que separan la lógica de presentación/negocios del manejo de la información) o patrones de 3-bandas (que separan la lógica de presentación, lógica de negocios y manejo de la información).
- *Patrón "Red de distribución de contenido"*: A fin de lograr que diferentes grupos de usuarios tengan una performance de acceso adecuada, los componentes asociados a las funcionalidades y al manejo de la información, se distribuyen globalmente. De esta manera, el patrón asegura que distintos grupos de usuarios (distribuidos globalmente) puedan acceder a la aplicación.

El arquitecto de software debe utilizar el patrón que mejor se ajuste a los requerimientos funcionales y no funcionales de la aplicación bajo desarrollo.

Paso 2: Identificar los Componentes de Aplicación

A nivel arquitectónico, las funcionalidades de una aplicación web se dividen en componentes de aplicación. En el contexto de CC, esta clase de componentes puede replicarse de forma independiente, por medio de la intervención de componentes de gestión, según su demanda.

El objetivo de este paso es establecer (con un alto nivel de abstracción) los componentes de aplicación necesarios para dar respuesta al conjunto de funcionalidades requeridas por los usuarios. Estos componentes deben clasificarse

según correspondan a *componentes de aplicación definidos* o *componentes de aplicación no definidos*.

El tipo de descomposición a realizar es funcional. Esto se debe a que el arquitecto debe especificar (según el patrón de alto nivel seleccionado en el paso precedente), las responsabilidades de los componentes de aplicación futuros en virtud de realizar una primera aproximación de los componentes requeridos sobre el esquema de la arquitectura final. Luego, el grado de descomposición a utilizar depende del nivel de detalle deseado por el arquitecto de software sobre el diseño resultante.

Es importante destacar que, en este paso, no se debe indicar la forma en la cual se llevaran a cabo las implementaciones de los componentes de aplicación definidos (en virtud de cumplir sus responsabilidades). Este nivel de detalle se incorpora como parte de la arquitectura en los pasos subsiguientes.

Paso 3: Definir los Mecanismos de Manejo del Estado

El estado dentro de una aplicación web puede interpretarse de acuerdo a dos puntos de vista diferentes, a saber:

- *Estado de sesión*: Refiere al estado de la interacción de la aplicación web con un cliente específico (por ejemplo, el carrito de compras en una tienda online).
- *Estado de aplicación*: Representa la información que maneja la aplicación a fin de garantizar un servicio adecuado a todos sus clientes (por ejemplo, datos o información enviada por los clientes que debe ser almacenada dentro de las bases de datos de la aplicación).

El objetivo de este paso es mejorar la descripción de los *componentes de aplicación* identificados en el paso precedente, a fin de establecer la forma en la cual manejan su estado (tanto de sesión como de aplicación). Luego, para cada componente definido se indica si corresponde a un *componente con estado* o *componente sin estado*.

En términos generales, el arquitecto de software debe buscar que el estado no se maneje dentro de los componentes. Este tipo de soluciones afecta el nivel de tolerancia a fallos de la aplicación debido a que, si un componente de aplicación maneja su propio estado y se produce una falla en el mismo, es imposible recuperar el estado del componente previo al fallo. Luego, el componente no se recuperara del problema y, además, la información asociada a su último estado conocido no se encontrara disponible.

En este contexto, la recomendación es: *i)* Gestionar el estado de sesión usando las solicitudes como vehículo de la información, y *ii)* Recopilar el estado de aplicación utilizando las ofertas de almacenamiento (accediendo al mismo por medio de las ofertas de comunicación).

Paso 4: Especificar el Criterio de Consistencia requerido para la Información

La consistencia de la información en una aplicación web es un aspecto clave que debe ser trabajado a lo largo de todo el proceso de desarrollo. Este aspecto impacta directamente sobre las propiedades no funcionales de la aplicación, motivo por el cual es necesario definir los lineamientos que ayudan a garantizar un nivel de consistencia adecuado en la información que se maneja.

El objetivo de este paso es aumentar el nivel de detalle de los *componentes con estado* identificados en el paso precedente por medio de la identificación del tipo de consistencia requerida para la información. En este caso, para cada componente definido se indica si corresponde a un criterio de *consistencia estricta*, *consistencia eventual* o *consistencia por abstracción de la información* (Tabla 7.2).

NOMBRE	DESCRIPCIÓN
Consistencia estricta	La información se almacena en distintas locaciones a fin de mejorar el tiempo de respuesta y evitar la pérdida de datos ante fallas en los componentes de almacenamiento. Se asegura, en todo momento, la consistencia de la información almacenada en las réplicas.
Consistencia eventual	En el caso que se presente una partición de red, este tipo de consistencia permite mantener el rendimiento y la disponibilidad de la información almacenada a fin de asegurar la consistencia de los datos de forma eventual (no en todo momento como en el caso de *consistencia estricta*).
Consistencia por abstracción de información	Se abstrae la información por medio del uso de aproximaciones y modelos conceptuales con el objetivo de brindar un mecanismo de respuesta adecuado para la información en base a un nivel de consistencia eventual de los datos almacenados.

Tabla 7.2. Distintos criterios para el manejo de la consistencia de la información.

Paso 5: Refinar la Especificación de los Componentes de Aplicación No Definidos

El objetivo de este paso es describir (con un nivel de detalle apropiado) la forma en la cual se llevarán a cabo las responsabilidades asociadas a los distintos *componentes de aplicación no definidos* que han sido identificados y refinados (de acuerdo al manejo de estado y consistencia de la información) en los pasos previos.

Esta tarea no aplica a los *componentes de aplicación definidos* ya que, como provienen de módulos comúnmente utilizados en las aplicaciones web, su funcionamiento es usualmente conocido por los desarrolladores (es decir, se encuentran familiarizados con su accionar). Esto no ocurre en el caso de los *componentes de aplicación no definidos* ya que su especificación se basa en las características propias de las funcionalidades requeridas en la aplicación. Luego, tales componentes corresponden a elementos que dependen del dominio.

El arquitecto de software es libre de utilizar el mecanismo de especificación que desee para estructurar la forma en la cual estos componentes cumplirán con sus responsabilidades. Sin embargo, a fin de complementar el esquema modular propuesto en términos de distintos tipos de elementos, se propone definir la funcionalidad interna

de los *componentes de aplicación no definidos* haciendo uso de uno o más *componentes funcionales*.

Teniendo en cuenta que cada *componente funcional* refiere a una responsabilidad básica, la combinación de dichas funcionalidades posibilita la especificación de nuevos comportamientos (que tendrán una complejidad mayor que la de los elementos originales). De esta forma, es posible definir el accionar de los componentes de dominio en base a comportamientos predefinidos. Además, se garantiza la comprensión de las funcionalidades requeridas en los componentes por parte de los desarrolladores (ya que se descomponen en funcionalidades básicas predefinidas).

Paso 6: Vincular los Componentes Funcionales

Una vez especificado el conjunto de *componentes funcionales* que conforman cada *componente de aplicación no definido*, se deben establecer las conexiones internas entre los mismos a fin de detallar el flujo de responsabilidades que debe seguir el componente de mayor nivel para ejecutar su funcionalidad.

El objetivo de este paso es definir los flujos de interacción requeridos entre los diversos *componentes funcionales* que han sido asociados a los *componentes de aplicación no definidos* como resultado del paso precedente.

Teniendo en cuenta que los *componentes funcionales* poseen una única entrada y una única salida, las conexiones deben realizarse como salida/entrada. El componente funcional que origina la secuencia de interacciones no lleva vínculos en su conexión de entrada, mientras que el componente que termina la secuencia no lleva vínculos en su conexión de salida. Todas las conexiones se consideran bidireccionales ya que representan interacciones.

No se recomienda especificar relaciones de recursividad directas ya que este tipo de estructuras requiere de la incorporación de nuevos elementos a fin de definir las condiciones de trabajo. Un esquema de este tipo (Figura 7.8), no contribuye a un único

entendimiento del diseño arquitectónico (ya que, probablemente, exista una definición funcional alternativa que divida las responsabilidades asociadas al camino A y B en dos nuevos componentes funcionales). Luego, el uso de este tipo de conexiones puede dar lugar a inconvenientes relacionados con la tarea de implementación (por ejemplo, generando implementaciones incorrectas o demasiado complejas del componente en cuestión).

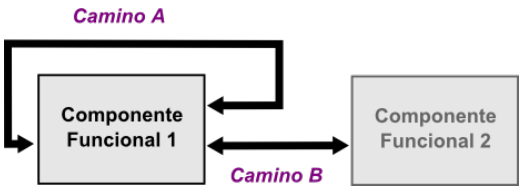


Figura 7.8. Relaciones recursivas en componentes funcionales.

Paso 7: Identificar los Componentes de Administración

Los *componentes de aplicación* identificados en los pasos precedentes requieren de un esquema de administración que realice las gestiones entre el software de aplicación y el hardware de infraestructura. Luego, el objetivo de este paso es especificar la forma en la cual deben administrarse los *componentes de aplicación* a fin de lograr una experiencia de usuario aceptable.

Para esto, el arquitecto de software debe hacer uso de los *componentes de administración* como elementos básicos para gestionar las funcionalidades detalladas. La meta en este tipo de procesos de administración es utilizar los mensajes y las solicitudes de usuario para determinar la cantidad de instancias requeridas de los *componentes de aplicación* detallados (es decir, la cantidad de réplicas). Una cantidad adecuada a la carga de trabajo específica en un momento dado permite no afectar la experiencia de uso de los usuarios y/o aplicaciones externas.

No necesariamente todos los componentes deben ser gestionados. Normalmente, este tipo de mecanismos se aplica en los componentes de dominio específicos de la

aplicación bajo diseño (esto es, componentes que ejecutan las principales funcionalidades del software).

Paso 8: Vincular los Componentes de Administración con los Componentes de Aplicación

El objetivo de este paso es establecer los vínculos de alto nivel entre los *componentes de aplicación* (tanto definidos como no definidos) y los *componentes de administración* identificados en el paso precedente.

Existen tres tipos de vínculos posibles, a saber: *i) componente de aplicación definido/ no definido* con *componente de administración*, *ii) componente de administración* con *componente de administración*, y *iii) componente de administración* con *componente de aplicación definido/no definido*. En este contexto, los *componentes de aplicación* de igual clase no pueden vincularse de forma directa sino que deben llevar (como intermediarios) uno o más *componentes de administración*. Estos componentes actúan como nexo de las solicitudes que se intercambian a fin de gestionar el correcto desempeño de la aplicación.

Paso 9: Verificar la Correctitud del Diseño Arquitectónico Final

El objetivo de este paso es comparar el nivel de adecuación del diseño desarrollado como resultado de la ejecución de los pasos precedentes con respecto al éxito obtenido en casos similares diseñados en base a *componentes de aplicación*, *componentes funcionales* y *componentes de administración*.

Un enfoque útil para realizar esta comparación es el uso de patrones de diseño arquitectónicos. Tomando en consideración la estructura de este tipo de soluciones, es posible analizar el diseño desarrollado a fin de estimar su correctitud.

Actividad para el Diseño de la Capa de Infraestructura (Sistema de Hardware)

En la actualidad existen muchos proveedores que brindan múltiples tipos de servicios para la infraestructura de las aplicaciones web, entre los que se destacan

Amazon, Google e IBM. La principal característica de estos servicios es que el usuario paga únicamente por el consumo que realiza por lo que, al contratar el servicio, el desarrollador sólo debe preocuparse por definir el tipo de instancia adecuado para garantizar el cumplimiento de los requerimientos de la aplicación de software subyacente. Sin embargo, aunque no es necesario seguir una relación entre los proveedores elegidos, al momento de realizar la selección debe tenerse en cuenta la compatibilidad.

En esta actividad se describe el subconjunto de ofertas más relevantes para el diseño de una infraestructura de CC. Para cada oferta, se detallan algunos de los servicios disponibles.

A diferencia de la actividad previa, se definen pautas a seguir con el objetivo de contextualizar al lector en el formato de las aplicaciones web con las que se continúa trabajando en el resto de la tesis. Aunque las pautas se encuentran numeradas, no existe un orden entre ellas.

Pauta 1: Definir las Ofertas de Procesamiento

Las *ofertas de procesamiento* son utilizadas como soporte para la ejecución de procesos o tareas. Usualmente, las características que definen este tipo de componentes refieren a estructura y tipo de procesador, cantidad de CPU virtuales y memoria.

La Tabla 7.3 resume algunas de las *ofertas de procesamiento* más utilizadas en el contexto de CC.

PROVEEDOR	NOMBRE	DESCRIPCIÓN
Amazon	Amazon Elastic Compute Cloud (Amazon EC2)	Servicio web que proporciona capacidad de cómputo en la nube en base a un tamaño modificable[4].
Google	Google Compute Engine	Ofrece máquinas virtuales que se ejecutan en los centros de datos de Google y están conectadas a través de una red de fibra a nivel mundial[5].
IBM	IBM Bluemix	Permite elegir instancias públicas o dedicadas (para controlar la ubicación de la carga de trabajo)[6].

Tabla 7.3. Ofertas de procesamiento para entornos de computación en la nube.

Pauta 2: Definir las Ofertas de Almacenamiento

En términos generales existen dos clases de *ofertas de almacenamiento* que refieren a la forma en la cual se entrega y/o solicita información sobre los componentes:

- *Almacenamiento basado en bloques*: El almacenamiento se centraliza en los servidores bajo la forma de un disco duro local administrado por el sistema operativo. Los servidores imitan el comportamiento de un dispositivo de bloques tradicional. Esto permite depositar y/o solicitar bloques específicos sobre el componente. Se debe montar un sistema de archivos sobre este almacenamiento a fin de mapear archivos sobre la secuencia de bloques. Para esto, la mayoría de los proveedores ofrece (sobre las ofertas basadas en bloques) esquemas de almacenamiento local persistente para, por ejemplo, bases de datos SQL y NoSQL.
- *Almacenamiento basado en sistema de archivos*: Al componente de almacenamiento se le entregan/solicitan archivos. Usualmente, estas ofertas proveen una interfaz a un sistema de archivos junto con la semántica de

[4] Para mayores detalles visitar *https://aws.amazon.com/es/ec2/*
[5] Para mayores detalles visitar *https://cloud.google.com/compute/*
[6] Para mayores detalles visitar *https://www.ibm.com/cloud-computing/bluemix/es*

acceso al mismo (a fin de indicar la información que debe ser utilizada por los consumidores).

La Tabla 7.4 resume algunas de las *ofertas de almacenamiento* existentes para la persistencia de la información en entornos de CC.

PROVEEDOR	NOMBRE	TIPO	DESCRIPCIÓN
Amazon	Amazon Elastic Block Store (Amazon EBS)	Bloques	Bloques persistentes que se replican automáticamente para proteger la información de errores en los componentes[7].
	Amazon Glacier	Sistema de archivos	Servicio que permite archivar datos y realizar backups a largo plazo. Para bajar los costos y resolver distintas situaciones, brinda tres tipos de acceso a los archivos[8].
Google	Block Storage	Bloques	Se disponen de recursos con distintas características, como ser tamaño y cantidad de IOPS[9] por instancia contratada[10].

Tabla 7.4. Ofertas de almacenamiento para entornos de computación en la nube.

Pauta 3: Definir las Ofertas de Comunicación

Las *ofertas de comunicación* se utilizan como mecanismo base para dar soporte al intercambio de información entre los distintos recursos de tecnología de la información que componen el entorno de CC.

La Tabla 7.5 resume algunas de las *ofertas de comunicación* disponibles por parte de los distintos proveedores de entornos de CC.

[7] Para mayores detalles visitar *https://aws.amazon.com/es/ebs/*
[8] Para mayores detalles visitar *https://aws.amazon.com/es/glacier/*
[9] Input / Output Operations per Second.
[10] Para mayores detalles visitar *https://cloud.google.com/compute/docs/disks/*

PROVEEDOR	NOMBRE	DESCRIPCIÓN
Amazon	Amazon Virtual Private Cloud (Amazon VPC)	Permite reservar una porción de la nube de Amazon (aislada de forma lógica), en la que puede definirse una red virtual propia con el objetivo de vincular recursos específicos. Quien contrata el servicio puede controlar todos los aspectos del entorno (como ser, por ejemplo, rango de direcciones IP, creación de subredes y configuración de tablas de ruteo y puertas de enlace de red)[11].
Google	Google Cloud Virtual Network	Permite provisionar recursos de Google Cloud Platform, conectarlos entre sí y aislarlos unos de otros por medio de la definición de una nube privada virtual[12].
IBM	IBM Virtual Private Network (VPN)	Provee conectividad segura a nivel de capa IP entre los componentes de la infraestructura de CC de IBM Bluemix[13].

Tabla 7.5. Ofertas de comunicación para entornos de computación en la nube.

Actividad para Unificar los Diseños Parciales en el Entorno de CC (Sistema Global)

Una vez definido el diseño de los dos sistemas involucrados en la construcción del sistema global, es necesario establecer los vínculos requeridos para especificar el diseño arquitectónico final del sistema de CC.

Para esto, tomando como referencia los elementos que componen los diseños desarrollados en las actividades precedentes, en los siguientes apartados se proponen tres lineamientos a seguir para lograr una estructura final adecuada.

Lineamiento 1: Desplegar los Componentes de Aplicación sobre las Ofertas de Procesamiento

Teniendo en cuenta que los *componentes de aplicación* definen el comportamiento de la aplicación web, estos componentes deben ser asignados a las *ofertas de procesamiento* a fin de dar lugar a su eventual ejecución. Sin embargo, tal como se ha establecido con anterioridad, los *componentes de administración* se encargan de

[11] Para mayores detalles visitar *https://aws.amazon.com/es/vpc/*
[12] Para mayores detalles visitar *https://cloud.google.com/vpc/*
[13] Para mayores detalles visitar *https://console.bluemix.net/docs/infrastructure/vpc/*

determinar la cantidad de réplicas de los *componentes de aplicación* que se requieren para cumplir con las cargas de trabajo solicitadas por los usuarios. Luego, tales componentes también deben interactuar con las *ofertas de procesamiento* a fin de monitorear el comportamiento de los *componentes de aplicación* en busca de opciones de ejecución para nuevas réplicas. De esta manera, se deben establecer las relaciones de alojo y desalojo de los *componentes de aplicación* sobre las *ofertas de procesamiento* teniendo en cuenta el monitoreo proporcionado por los *componentes de administración*.

En este punto es importante tener en cuenta que una aplicación web se considera distribuida si sus componentes se alojan en más de un recurso de procesamiento. Por este motivo, el arquitecto de software debe decidir si desea o no este tipo de despliegue al momento de ajustar la relación entre los *componentes de aplicación* y las *ofertas de procesamiento*.

Lineamiento 2: Vincular los Componentes que Manipulan Información con las Ofertas de Almacenamiento

El arquitecto de software debe identificar el conjunto de *componentes funcionales* que requieren acceso a datos para cumplir con sus responsabilidades básicas. Tales elementos deben, necesariamente, conectarse con las *ofertas de almacenamiento* a fin de dar de alta/baja información, modificar los datos almacenados y/o consultar los datos existentes en virtud de cumplir con sus funcionalidades.

Además, el conjunto de *componentes de aplicación no definidos* cuyo estado de sesión y/o estado de aplicación requiera de un mecanismo de administración externo, también debe vincularse a las *ofertas de almacenamiento* a fin de dar soporte a la persistencia de tales estados.

Lineamiento 3: Relacionar los Componentes que Envían/Reciben Mensajes/Transacciones con las Ofertas de Comunicación

A fin de garantizar un correcto funcionamiento en el envío de mensajes (es decir, que no se presentarán inconsistencias y/o problemas de procesamiento duplicado), es necesario indicar la forma en la cual se respalda el envío de mensajes y/o transacciones. Para esto, el arquitecto de software debe asociar el conjunto de componentes que manipula mensajes y/o transacciones con las *ofertas de comunicación*. De esta manera, se explicita a nivel arquitectónico el mecanismo de soporte de los componentes que envían y/o reciben información (mensajes y/o transacciones) en base a la estructura que proveen las *ofertas de comunicación*.

7.5 Beneficios: SoS, Co-Diseño y Arquitecturas Web

El proceso de diseño propuesto en la sección previa establece un conjunto de lineamientos generales que permiten vincular dos sistemas diseñados de forma independiente (esto es, el sistema de la aplicación web y el sistema de la infraestructura basada en ofertas de proveedores) en virtud de obtener un diseño arquitectónico factible de ser utilizado sobre un entorno de CC. Es decir, define una posible forma de trabajo para el arquitecto de software a fin de lograr un diseño final que aproveche los beneficios de los componentes comúnmente utilizados en las aplicaciones web.

En este contexto, es importante destacar que la propuesta:

- Independiza el diseño de la aplicación del diseño de la infraestructura subyacente, posibilitando la contratación de distintas ofertas de servicios para los recursos de tecnología de la información requeridos en relación a un mismo diseño web.
- Propone un conjunto de clases de elementos básicos a ser utilizados para estructurar los diseños arquitectónicos brindando la posibilidad, a futuro, de

generar un mapeo directo entre tales elementos y uno o más módulos de software.

- Permite distribuir las responsabilidades de diseño en función del conocimiento y experiencia de los arquitectos, contribuyendo al aprendizaje por medio de un proceso guía que utiliza un conjunto de componentes predefinidos.

Conclusiones

El diseño de arquitecturas de software en entornos de computación en la nube es una tarea compleja que requiere de profesionales con un alto nivel de conocimiento sobre estrategias, técnicas y herramientas aplicables a este tipo de problemas. En este sentido, la representación de este tipo de diseños de software aún no encuentra la estabilidad deseada, dando lugar a especificaciones arquitectónicas asociadas a la infraestructura que resultan ambiguas en relación a la lógica de las aplicaciones web.

A fin de contribuir al desarrollo de estas arquitecturas, en este capítulo se ha definido un proceso de diseño que utiliza como marco de referencia la perspectiva de sistemas-de-sistemas junto con el enfoque de co-diseño. El proceso propuesto se basa en un modelo abstracto de capas que posibilita el estudio y análisis de las arquitecturas de computación en la nube en términos de dos grandes sistemas: sistema de aplicación y sistema de infraestructura. Los componentes fundamentales de este proceso quedan delimitados de acuerdo a una clasificación de tipos componentes web comúnmente encontrados en las arquitecturas de software.

En este contexto, el conjunto de actividades de diseño propuestas (definidas en términos de pasos, lineamientos y pautas) ayudan al arquitecto de software a generar un diseño válido para aplicaciones web ejecutadas sobre una infraestructura de servicios contratados en la nube. Bajo esta perspectiva, el arquitecto tiene la posibilidad de construir el diseño de una aplicación de software web de forma independiente a la infraestructura subyacente.

En el capítulo siguiente se presenta un conjunto de patrones de diseño arquitectónicos asociados al desarrollo de aplicaciones web, el cual es tomado como base para la definición de componentes específicos del dominio.

Capítulo 8. Diseños de Aplicaciones Web Basados en Patrones Arquitectónicos

Las arquitecturas de software han evolucionado de simples representaciones estructurales a esquemas centrados en decisiones (Kruchten, Capilla y Dueas 2009). Los atributos de calidad relevantes usualmente se ven reflejados en patrones de diseño que ayudan al arquitecto a resolver problemas frecuentes en términos de composiciones que son utilizadas como plantillas u esquemas de diseño. En el capítulo previo se han identificado clases de componentes web siguiendo los lineamientos propuestos por (Fehling y colab. 2014). Haciendo uso de esta clasificación, en este capítulo se presenta un metamodelo que permite diseñar arquitecturas de software adecuadas para entornos web de acuerdo a un proceso de verificación sustentado en patrones de diseño. Este metamodelo incluye los principales componentes web y un conjunto de relaciones definidas en función del análisis de los patrones de diseño existentes. Su especificación se detalla en UML y es complementada por un conjunto de restricciones OCL que ayuda a verificar la integridad de las representaciones diseñadas. Se presenta además una herramienta de modelado gráfica que brinda soporte a la instanciación de metamodelo y, posteriormente, a la verificación de los diseños creados a partir del mismo.

8.1 Patrones de Diseño

En la actualidad, existe una amplia variedad de patrones arquitectónicos que contribuyen a la resolución del diseño de productos de software. Sin embargo, usualmente se presentan problemas al intentar trasladar los patrones de diseño tradicionales a los esquemas de diseño de sistemas de software más novedosos basados en modelos de ejecución modernos. Este es el caso de los esquemas de CC.

Tal como se ha enunciado en el *Capítulo 1*, un *patrón arquitectónico* define una familia de sistemas en base a un esquema de organización estructural. De acuerdo con esta perspectiva, los patrones de diseño arquitectónico describen pares de elementos *{problema; solución}* con el objetivo de brindar soluciones abstractas a problemas recurrentes en un dominio específico.

En este sentido, los patrones de diseño para aplicaciones web describen soluciones abstractas a problemas recurrentes en el dominio de CC a fin de capturar conocimiento independiente de los proveedores de servicios de infraestructura[1]. Estos patrones ayudan a determinar la forma en la cual deben ser distribuidos los distintos tipos de componentes de una aplicación. Una vez definido el estilo de alto nivel (partición basada en capas para los entornos de CC de acuerdo al esquema definido en el apartado *"Abstracción de los Modelos basados en Capas"*), es necesario definir el conjunto de patrones de diseño a utilizar como base para la especificación de los componentes internos. Esta selección no debe realizarse de forma arbitraria, sino que siempre se encuentra relacionada con los requerimientos no funcionales de la aplicación.

No todos los patrones de CC son nuevos. Muchos de los patrones existentes para las arquitecturas de software tradicionales han sido adaptados para su uso en contextos de aplicaciones web basadas en CC.

[1] En este punto es importante comprender que de aquí en adelante sólo se trabaja sobre el diseño de las aplicaciones ubicadas en la capa superior de una arquitectura de CC.

Uno de los enfoques más completos en esta área ha sido propuesto en (Fehling y colab. 2014), donde se presenta un conjunto de patrones de diseño para la especificación de arquitecturas de aplicaciones de software contextualizadas en entornos de CC. Esta propuesta abarca patrones que involucran tanto la infraestructura de la aplicación como la lógica de negocios requerida para dar respuesta a requerimientos de software específicos, definiendo un conjunto de elementos básicos a ser utilizados como herramienta constructiva durante el proceso de diseño arquitectónico.

En virtud de obtener un conjunto acotado de elementos que facilite la definición de arquitecturas web a nivel de componentes de software, en los siguientes apartados se establece el conjunto de pautas aplicadas sobre los patrones propuestos a fin de identificar, seleccionar y resumir un conjunto de elementos básicos aplicables para la elaboración de este tipo de diseños.

8.1.1 Propiedades Ideales de las Aplicaciones Web

En términos generales, existe un conjunto de propiedades que (idealmente) toda aplicación web debe poseer, entre las que se destacan:

- Manejo del estado de componentes de software específicos de forma descentralizada.
- Utilización de múltiples recursos de tecnología de la información.
- Elasticidad mediante escala dinámica (tanto por la adición y/o remoción de recursos de infraestructura como por el incremento y/o decremento de las capacidades de los recursos de tecnología de la información asignados a los componentes de software específicos).
- Influencia de componentes de software limitada.
- Soporte de ejecución en entornos distribuidos globalmente.
- Administración automatizada.

En este contexto, los patrones arquitectónicos analizados en este capítulo buscan mantener este conjunto de propiedades como resultado de los diseños arquitectónicos propuestos, a fin de garantizar la construcción de estructuras arquitectónicas acordes a las expectativas de este tipo de productos.

8.1.2 Estilos Arquitectónicos Básicos

Existen dos esquemas fundamentales sobre los cuales se detallan los estilos arquitectónicos básicos que deben ser utilizados por los arquitectos y desarrolladores de software durante el diseño y la construcción de aplicaciones web: *aplicación distribuida* y *bajo acoplamiento*.

Bajo el *esquema de aplicación distribuida* las funcionalidades se dividen en múltiples componentes de aplicación que pueden ser replicados de forma independiente. En el caso del diseño arquitectónico de las aplicaciones web, el arquitecto debe intentar hacer uso de la distribución y replicación propia del entorno de CC a fin de beneficiarse de la estructura del ambiente como parte de su solución. Por este motivo, usualmente las aplicaciones web se valen de múltiples recursos redundantes de tecnología de la información que les permiten dividir sus funcionalidades (de acuerdo a una determinada función) en diversos componentes autónomos. Este tipo de separación da lugar a una descomposición lógica de la aplicación web final.

Por su parte, en el *esquema de bajo acoplamiento* se plantea la existencia de un intermediario de comunicación que separa la funcionalidad de la aplicación de los aspectos relacionados con la infraestructura subyacente (como, por ejemplo, la ubicación de los recursos, el tiempo de comunicación y el formato de intercambio de datos). Luego, se simplifica el intercambio de información entre los componentes de software y las tareas de administración asociadas a los mismos (como ser réplicas, manejo de fallos y administración de actualizaciones) ya que los componentes de aplicación se tratan individualmente y las dependencias entre los mismos se mantienen

reducidas al mínimo. De esa manera, se logra: *i)* autonomía de plataforma (accesos en distintos lenguajes de programación), *ii)* autonomía de referencia (ruteo entre diferentes ubicaciones), *iii)* autonomía de tiempo (comunicaciones a distintas velocidades), y *iv)* autonomía de formato (transformación de distintos formatos de información).

8.2 Componentes Arquitectónicos

Siguiendo la clasificación detallada en el apartado *"Identificación de Componentes Arquitectónicos Generales"* y con el objetivo de describir un conjunto de componentes específicos para la construcción de arquitecturas de software de aplicaciones web, en las siguientes secciones se describen los elementos seleccionados a partir de los patrones de diseño analizados. Tales elementos corresponden a las clases hoja (componentes de más bajo nivel) de la clasificación propuesta (Figura 8.1). Los componentes restantes solo se definen para representar la clasificación (es decir, no son utilizados para modelar las arquitecturas).

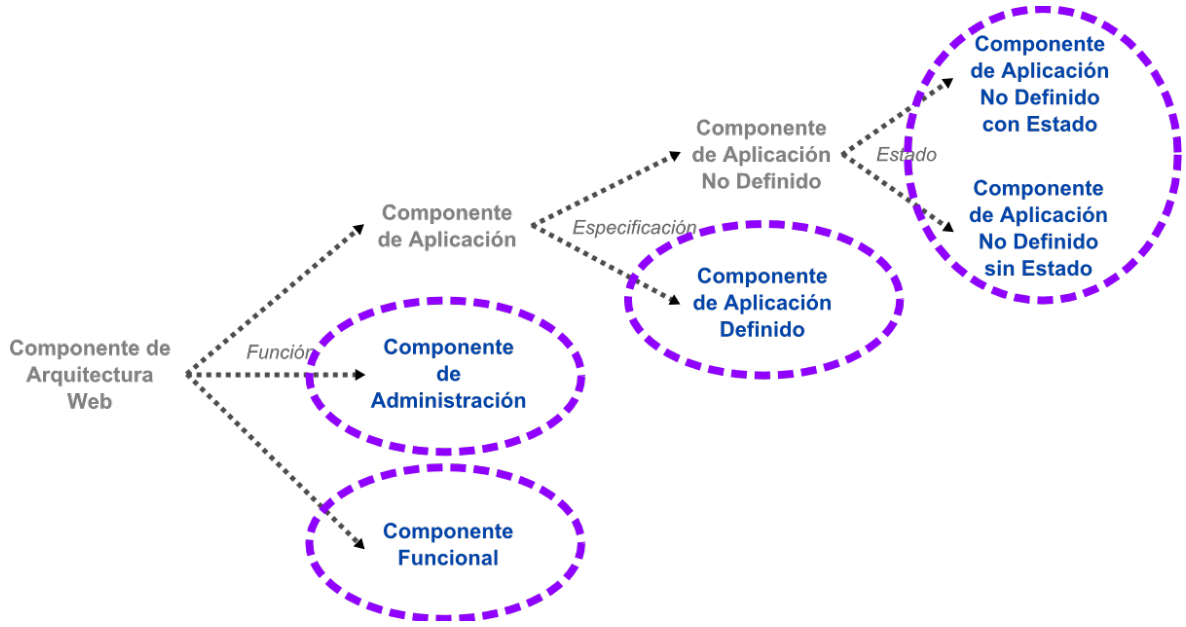


Figura 8.1. Clases de componentes definidos para arquitecturas de software web.

En cada caso se presenta una tabla resumen que incluye una descripción general del componente seleccionado junto con el ícono con el cual (Fehling y colab. 2014) hace referencia al elemento.

8.2.1 Componentes de Aplicación (No Definidos)

La Tabla 8.1 presenta los dos componentes no definidos disponibles para la especificación de las arquitecturas web. Dado que estos elementos son componentes de aplicación, su comportamiento debe definirse haciendo uso de *componentes funcionales*.

COMPONENTE	SIGLA	ICONO	DESCRIPCIÓN
Componente con Estado	CACE		Las instancias de un componente de aplicación sincronizan su estado interno para proveer un comportamiento uniforme.
Componente sin Estado	CASE		El estado de los componentes de aplicación es manejado de forma externa para facilitar su instanciación y hacer que la aplicación tenga una mayor tolerancia a fallas.

Tabla 8.1. Componentes arquitectónicos - Nivel aplicación no definido (resumen).

Componente de Aplicación con Estado (Stateful Component)

A fin de beneficiarse del ambiente de ejecución distribuido que proveen los entornos de CC, los componentes de aplicación deben desplegarse en múltiples recursos de tecnología de la información (por lo que sus instancias deben ser escalables).

En este contexto, existe un subconjunto de componentes que si o si necesitara mantener su estado internamente para cumplir de forma adecuada con sus funcionalidades. En este caso, el desafío es que todas las instancias individuales de un mismo componente de aplicación mantengan el mismo estado interno para que presenten un comportamiento uniforme.

En un componente CACE el estado interno se replica en todas las instancias utilizando poca cantidad de información compartida. De esta manera, los componentes de aplicación son escalables mientas mantienen un estado interno sincronizado. Es importante destacar que si el volumen de información a compartir es grande, no es adecuado utilizar este tipo de componentes (ya que se requiere un mayor esfuerzo para lograr una sincronización adecuada).

Componente de Aplicación sin Estado (Stateless Component)

En el caso de los componentes CACE, el factor clave que complica la adición o remoción de réplicas de componentes es el estado interno que deben mantener las distintas instancias. Ante una eventual falla en alguna de las réplicas a ser sincronizadas, es posible que se pierda información antes de lograr la sincronización completa de todas las instancias del componente. Luego, no es bueno que todos los componentes de aplicación manejen su estado de forma interna sincronizada.

Los componentes CASE son implementados de forma que no tengan estado interno. Su estado y configuración se almacena en un medio externo (haciendo uso de ofertas de almacenamiento o de algún tipo de repositorio específico) del cual se obtiene la información requerida cuando el componente la solicita. Este mecanismo facilita la creación de réplicas (ya que no deben sincronizarse) y genera una aplicación más tolerante a fallas (ya que no existe la posibilidad de que se pierda información).

Por medio del uso de componentes CASE, se logra elasticidad y robustez en componentes de aplicación escalables.

8.2.2 Componentes de Aplicación (Definidos)

La Tabla 8.2 presenta dos componentes de aplicación de comportamiento definido (es decir, que su funcionamiento no depende del dominio de trabajo). Tales elementos son comúnmente utilizados en las arquitecturas de software de aplicaciones web.

Este listado no es exhaustivo por lo que, si el arquitecto lo desea, puede incorporar nuevos elementos de trabajo.

COMPONENTE	SIGLA	ID	DESCRIPCIÓN
Balanceador de Carga	BC		Determina el número de accesos síncronos a la aplicación.
Cola de Mensajes	CM		Determina el número de accesos asíncronos a la aplicación (es decir, el acceso a través de mensajes).

Tabla 8.2. Componentes arquitectónicos - Nivel aplicación definido (resumen).

Balanceador de Carga (Load Balancer)

Este componente de aplicación posee las siguientes características:

- Se reciben solicitudes de trabajo.
- Se envían solicitudes de trabajo.
- Su salida se encuentra conectada a múltiples réplicas de un mismo *componente de aplicación no definido* que debe procesar una única vez cada trabajo recibido.

En este contexto, su objetivo es distribuir las solicitudes recibidas sobre los componentes de salida a fin de garantizar su correcto procesamiento. Para lograr una distribución adecuada, el componente BC debe tomar en consideración la carga de trabajo actual de los componentes de salida a fin de garantizar un trabajo equitativo (no ralentizando el tiempo de respuesta de la aplicación).

Cola de Mensajes (Message Queue)

El componente CM se utiliza para distribuir solicitudes asincrónicas (mensajes) entre las múltiples instancias de componentes de aplicación no definidos. Cuando no existen componentes libres para procesar los mensajes recibidos, mantiene los

mensajes en una estructura FIFO (*first-in-first-out*) que garantiza su posterior procesamiento según el orden en el cual arribaron al componente.

8.2.3 Componentes de Administración

La Tabla 8.3 presenta el conjunto de componentes de administración que el arquitecto de software puede utilizar para describir la forma en la cual deben gestionarse las funcionalidades diseñadas en la arquitectura de software de una aplicación web (las cuales quedan definidas en términos de *componentes de aplicación*).

COMPONENTE	SIGLA	ICONO	DESCRIPCIÓN
Balanceador de Carga Elástico	BCE		La cantidad de accesos sincrónicos al componente es utilizada para determinar la cantidad de instancias requeridas del componente de aplicación.
Cola de Mensajes Elástica	CME		Se utiliza la cantidad de accesos al componente junto con mensajes para ajustar la cantidad de instancias requeridas del componente de aplicación.
Adaptador para Proveedor	AP		Encapsula en una interfaz abstracta las implementaciones específicas de interacción con un proveedor dado.
Perro Guardián	PG		Las aplicaciones hacen frente a las fallas por medio del monitoreo y remplazo de las instancias de componentes de aplicación.
Administrador de Configuración	AC		Guarda las configuraciones de los componentes de aplicación en ofertas de almacenamiento central (las cuales son accedidas periódicamente por las instancias).
Administrador Elástico	AE		Monitorea los recursos de tecnología de la información para garantizar un despliegue adecuado de los componentes de aplicación.

Tabla 8.3. Componentes arquitectónicos - Nivel administración (resumen).

Balanceador de Carga Elástico (Elastic Load Balancer)

Los *componentes de aplicación no definidos* deben replicarse de forma automática. En este contexto, la cantidad de solicitudes enviadas a la aplicación puede ser utilizada como indicador de la carga de trabajo actual.

Haciendo uso de esta información, el componente BCE deduce la cantidad de instancias de *componentes de aplicación* requeridas para dar respuesta a las peticiones de los usuarios. De esta manera, define la cantidad de instancias de un componente de aplicación específico de acuerdo a la cantidad de solicitudes sincrónicas manejadas por el *balanceador de carga* que controla dichas réplicas.

Cola de Mensajes Elástica (Elastic Message Queue)

El componente CME monitorea las *colas de mensajes* que se utilizan para distribuir las solicitudes asincrónicas entre múltiples instancias de un conjunto de *componentes de aplicación* dado. De acuerdo al número de mensajes encolados, ajusta el número de instancias (réplicas) de dichos componentes a fin de manejar de forma adecuada los requerimientos.

Adaptador para Proveedor (Provider Adapter)

Usualmente los proveedores de ofertas en la nube ofrecen múltiples interfaces a ser usadas por los *componentes de aplicación* con el objetivo de acceder a un determinado servicio. Si un *componente de aplicación* interactúa directamente con estas interfaces, su implementación se vuelve altamente dependiente de las funciones específicas ofrecidas por el proveedor y del protocolo que este utilice como base de comunicación.

En este sentido, el componente AP encapsula en una interfaz abstracta el conjunto de implementaciones específicas propias de un proveedor web. Por ejemplo, para un proveedor de ofertas de almacenamiento se deben incluir implementaciones que abstraigan los mecanismos específicos asociados a la autenticación y el formato de

los datos (entre otros). De esta manera, el componente asegura una separación de responsabilidades entre los componentes de aplicación que acceden a las funcionalidades de los proveedores y los componentes de aplicación que proveen funcionalidades de aplicación.

Luego, las interfaces de los proveedores son encapsuladas y mapeadas para unificar las interfaces utilizadas en las aplicaciones a fin de separar los aspectos de interacción con el proveedor de la funcionalidad de aplicación.

Perro Guardián (Watchdog)

Cuando una aplicación web distribuida queda definida en términos de múltiples *componentes de aplicación*, su funcionamiento depende de la disponibilidad de las instancias de todos sus componentes. Para lograr una alta disponibilidad en estas condiciones, las aplicaciones deben hacer uso de componentes con instancias redundantes. En este escenario, un eventual fallo de componentes debe, necesariamente, ser detectado y puesto en evidencia de forma automática.

El objetivo del componente PG es monitorear las instancias de los *componentes de aplicación* a fin de detectar eventuales fallas en su funcionamiento, generando un reemplazo del componente dañado si el proveedor de recursos asegura que existe la disponibilidad requerida para su incorporación.

Administrador de Configuración (Configuration Manager)

Los *componentes de aplicación* usualmente poseen parámetros de configuración. El almacenamiento de la información de configuración junto con la implementación de este tipo de elementos suele ser poco práctico ya que resulta en una mayor carga de trabajo ante cambios en la configuración (debido a que cada instancia del componente que se encuentre en ejecución debe ser actualizada de forma individual).

En estos casos, el componente AC permite almacenar las configuraciones en una oferta de almacenamiento central (comúnmente planteada como una base de datos relacional, almacenamiento clave-valor o almacenamiento *Binary Large Objects*), la cual es accedida periódicamente (por medio del AC o mediante envío de mensajes) por las instancias en ejecución del componente a configurar. De esta manera, los *componentes de aplicación* utilizan una misma configuración (almacenada en un espacio centralizado) a fin de proveer un comportamiento unificado que puede ser ajustado simultáneamente sin obstaculizar la carga de trabajo.

Administrador Elástico (Elastic Manager)

Las diferentes réplicas de los *componentes de aplicación* web son normalmente distribuidas en una infraestructura o plataforma elástica. Tales instancias deben ser asignadas y desasignadas automáticamente en los recursos de tecnología de la información disponibles (de acuerdo a la carga de trabajo experimentada en la aplicación).

El componente AE monitorea los recursos de tecnología de la información sobre los cuales se despliegan las instancias de los distintos *componentes de aplicación* a fin de determinar la instanciación, asignación y remoción de estos componentes sobre los recursos disponibles.

8.2.4 Componentes Funcionales

La Tabla 8.4 presenta el conjunto de componentes funcionales que el arquitecto de software puede utilizar para especificar el comportamiento asociado a las funcionalidades diseñadas en los *componentes de aplicación no definidos*. Como puede observarse, cada componente detallado posee una función específica (elemental) que, por medio de la combinación con otros elementos del mismo tipo, contribuye a la construcción de funciones más complejas.

Componente de Procesamiento (Processing Component)

El procesamiento de las distintas funcionalidades incluidas en una aplicación web debe ser manejado por múltiples instancias de componentes de aplicación cuya operación se disponga de forma independiente. Las instancias de estos componentes deben ser agregadas y removidas con facilidad a la aplicación como parte de las operaciones de escala.

En este contexto, el componente CP posibilita la ejecución de las funcionalidades de procesamiento dando lugar a un posterior manejo de múltiples componentes individuales (lo que permite escalar elásticamente los componentes requeridos). Bajo esta perspectiva, la funcionalidad de procesamiento permite dividir las funciones en bloques que, luego, son asignados a ofertas de procesamiento independientes. Por cada ocurrencia de un componente CP, se debe indicar una oferta de procesamiento asociada que facilite su ejecución.

Normalmente este componente funcional se utiliza en *componentes de aplicación no definidos sin estado*, ya que suele requerir de un gran volumen de información a ser procesado (por lo que debe obtenerse de las solicitudes o desde ofertas de almacenamiento).

COMPONENTE	SIGLA	ICONO	DESCRIPCIÓN
Componente de Procesamiento	CP		El procesamiento es manejado en componentes que pueden ser escalados elásticamente.
Componente de Procesamiento Batch	CPB		Las solicitudes son demoradas hasta que las condiciones del entorno posibilitan su procesamiento.
Componente de Interfaz de Usuario	CIU		Actúa como puente entre el acceso sincrónico de los usuarios humanos y las comunicaciones asincrónicas utilizadas por los componentes de aplicación.
Imagen Multicomponente	IM		Los servidores virtuales almacenan múltiples componentes de aplicación (posiblemente no activos) a fin de reducir el costo de las operaciones de asignación/remoción.

Procesador de Mensajes basado en Tiempo	PMT		Envía una confirmación luego de procesar correctamente el mensaje recibido.
Procesador de Mensajes basado en Transacciones	PMTR		Extiende el contexto transaccional al procesamiento de los mensajes.
Componente de Acceso a Datos	CAD		Integra los mecanismos de acceso y manipulación de diferentes proveedores de almacenamiento.
Abstracción de Datos	AD		La información obtenida de las ofertas de almacenamiento es abstraída para lograr consistencia eventual.
Procesador Idempotente	PI		Asegura que los mensajes duplicados y los datos inconsistentes no afecten la funcionalidad del componente de aplicación asociado.

Tabla 8.4. Componentes arquitectónicos - Nivel funcional (resumen).

Componente de Procesamiento Batch (Batch Processing Component)

Las aplicaciones distribuidas asignan la funcionalidad de procesamiento a diferentes componentes independientes. Si estos componentes son accedidos de forma asincrónica, ciertas condiciones del entorno pueden hacer inviable el procesamiento inmediato de las solicitudes (como ser, por ejemplo, instancias de componentes de procesamiento accedidas constantemente que abarcan el poder de cómputo disponible o un incremento inviable de los costos asociados a la contratación de nuevos recursos que den lugar a la ejecución de nuevas instancias). En estos casos, las solicitudes deben ser demoradas hasta que las condiciones del entorno posibiliten su procesamiento. Sólo se realizará un procesamiento en condiciones no óptimas cuando dichas solicitudes no puedan ser retrasadas por más tiempo.

El componente CPB acepta solicitudes de procesamiento asincrónicas en todo momento pero almacena su información hasta que se den las condiciones óptimas para su procesamiento. En base a la cantidad de solicitudes almacenadas, las

condiciones del entorno de ejecución y un conjunto de reglas personalizadas, se instancian los componentes CPB para manejar las solicitudes requeridas.

Componente de Interfaz de Usuario (User Interface Component)

Las instancias de los componentes asociados a la interfaz gráfica provista por la aplicación web deben agregarse/removerse fácilmente a fin de no afectar la experiencia de los usuarios. Por este motivo, las dependencias de este tipo de elementos sobre otros tipos de componentes deben ser reducidas al mínimo (a fin de no generar una mayor carga de trabajo cada vez que se agrega/quita un componente gráfico).

En este caso, el componente CIU sirve como puente entre el acceso sincrónico de los usuarios humanos y las comunicaciones asincrónicas utilizadas por otros componentes de aplicación (las interacciones internas son asincrónicas para asegurar bajo acoplamiento), dando lugar al manejo de la interfaz de usuario requerida en base a información de estado externa. Es decir, este tipo de componente funcional debe ser utilizado como parte de *componentes de aplicación no definidos sin estado*, en los cuales la información es obtenida de medios externos (generalmente de las solicitudes iniciadas como parte de la propia interfaz en el dispositivo del usuario o, menos frecuentemente, de un almacenamiento externo).

Teniendo en cuenta que usualmente las interfaces web brindan la posibilidad de personalizar su apariencia (a fin de ser utilizadas por diferentes clientes), el componente CIU garantiza un acceso interactivo y, al mismo tiempo, la posibilidad de configurar el elemento de forma desacoplada a las aplicaciones restantes.

Imagen Multicomponente (Multicomponent Image)

Una aplicación web distribuida despliega sus componentes de aplicación a lo largo de múltiples servidores virtuales provistos por una infraestructura elástica. Los componentes de aplicación individuales pueden, sin embargo, no utilizar toda la capacidad contratada (por ejemplo, si solo se aloja un componente por servidor). Por lo

tanto, si no se monitorea el mapeo de los componentes a los servidores es posible generar una baja utilización de los recursos contratados.

El componente IM busca que múltiples componentes de aplicación sean alojados en un mismo servidor virtual a fin de asegurar que la ejecución de los recursos restantes contratados pueda ser utilizada para dar soporte a otro tipo de tareas (sin necesidad de realizar operaciones de entrega/retiro de recursos). De esta manera, los componentes de aplicación son asignados a los servidores virtuales disponibles (aun cuando no se encuentren activos en todo momento) con el objetivo de reducir la cantidad de operaciones de asignación/remoción vinculadas a tales componentes (ya que el servidor provee de funcionalidad a múltiples componentes de aplicación flexibilizando las capacidades de las aplicaciones).

Procesador de Mensajes basado en Tiempo (Timeout based Message Processor)

Un middleware orientado a mensajes puede utilizar una entrega basada en tiempo para asegurar que los mensajes han sido recibidos exitosamente por al menos un cliente. Sin embargo, en el caso de las aplicaciones web se debe asegurar adicionalmente que el mensaje ha sido procesado de forma apropiada luego de su recepción.

En lugar de enviar una confirmación luego de recibir el mensaje, el componente PMT envía una confirmación luego de que efectivamente ha procesado de forma correcta dicho mensaje. Así, los clientes acusan la recepción y el procesamiento de los mensajes asegurando que han sido manejados correctamente por la aplicación.

Si el mensaje no es confirmado luego de un tiempo predefinido, su procesamiento se asigna a otro cliente. De esta manera, se busca que la aplicación web procese (al menos una vez) todos los mensajes recibidos.

Procesador de Mensajes basado en Transacciones (Transaction based Message Processor)

Un middleware orientado a mensajes puede utilizar una entrega basada en transacciones a fin de asegurar que los mensajes han sido recibidos exitosamente por los clientes. Este tipo de entrega encapsula (en una única transacción) la lectura del mensaje y su remoción de la cola de mensajes pendientes de recepción. Sin embargo, no asegura nada acerca de su procesamiento.

El componente PMTR extiende el contexto transaccional incorporando el procesamiento del mensaje como parte de la recepción (por lo que interactúa con las ofertas de almacenamiento, lectura, procesamiento y escritura de los datos). Bajo esta perspectiva, los componentes que reciben mensajes o leen datos a fin de procesarlos en base a un contexto transaccional aseguran, respectivamente, que todos los mensajes recibidos son procesados y que todos los datos alterados son consistentes luego del procesamiento.

De esta manera, se busca asegurar que los componentes de aplicación procesan exitosamente todos los mensajes que reciben y que la información alterada por medio de las operaciones de procesamiento se mantiene consistente.

Componente de Acceso a Datos (Data Access Component)

Las operaciones requeridas para acceder a los datos de una aplicación web (como ser, entre otros, el pedido de autorización, la gestión de la cola de espera y la administración de posibles fallos en las consultas) genera una estrecha dependencia funcional entre los componentes de aplicación que requieren la información y las ofertas de almacenamiento utilizadas para dar soporte a los datos. Si se maneja esta dependencia como parte de las responsabilidades asignadas a los distintos componentes de aplicación, se complejiza la implementación de estos componentes por la introducción de nuevas responsabilidades ajenas a sus objetivos.

El componente CAD integra el acceso a diferentes fuentes de datos (posiblemente de distintos proveedores) con el objetivo de: *i)* proveer a los componentes de aplicación un acceso unificado a la información, y *ii)* coordinar las operaciones de manipulación de los datos almacenados. En este contexto, en el caso de que una oferta de almacenamiento sea remplazada o que su interfaz de acceso sea modificada, el componente CAD es el único que debe ser ajustado.

De esta manera, este tipo de componente funcional aísla la complejidad del acceso a los datos dando lugar a un nivel adicional de consistencia de la información (ya que maneja la complejidad del almacenamiento en base a protocolos de acceso que permiten configurar las estructuras de datos).

Abstracción de Datos (Data Abstractor)

Cuando una aplicación web distribuida diseñada en términos de consistencia eventual es desplegada en ofertas de almacenamiento, la consistencia de los datos puede lograrse utilizando un componente CAD. Sin embargo, este componente puede anular los beneficios (referidos al rendimiento y disponibilidad) introducidos por la consistencia eventual ya que garantiza una misma representación en todas las ocasiones.

En este contexto, el componente AD ajusta el estilo de la representación de los datos a fin de permitir que los datos recuperados sean eventualmente consistentes. La representación de los datos siempre refleja que el estado consistente es desconocido, haciendo uso de aproximación de valores o abstracción en valores más generales (como, por ejemplo, barras de progreso, gráficos de estado o transferencia y tendencias de cambio –aumentos/ decrementos-).

De esta manera, los datos que provee el componente (tanto a los usuarios como a otros componentes de aplicación o funcionales) son abstraídos (por medio de la aplicación de abstracciones y aproximaciones) para dar lugar a la consistencia eventual

de la información almacenada (requerida en el diseño de la aplicación). Bajo este enfoque de trabajo, se busca determinar un formato de presentación de la información en el cual se oculten posibles inconsistencias de los datos (permitiendo que la información recuperada por los componentes finales sea eventualmente consistente).

Procesador Idempotente (Idempotent Processor)

En una aplicación web distribuida existen dos posibles problemas derivados del procesamiento de datos inconsistentes y/o mensajes duplicados, a saber:

- Cuando las ofertas de almacenamiento se basan en consistencia eventual, los componentes de aplicación pueden leer información obsoleta que ya ha sido procesada.
- Cuando los componentes de aplicación intercambian información de forma asincrónica por medio de un middleware basado en mensajes que asegura al menos una entrega.

En el primer caso el componente de aplicación debería ser capaz de elegir si desea o no procesar nuevamente estos datos, mientras que en el segundo caso un procesamiento duplicado puede dar lugar a errores en la funcionalidad de aplicación asociada.

El componente PI tiene como objetivo asegurar que tanto los mensajes duplicados como los datos inconsistentes no afectaran de forma negativa el normal funcionamiento de los componentes de aplicación. Para esto, identifica y rastrea mensajes y/o datos duplicados o utiliza semánticas idempotentes como parte de las funciones de aplicación a fin de evitar que el componente asociado sea ejecutado (erróneamente) múltiples veces con un mismo resultado.

En base al tipo de inconsistencia identificada, las funciones de aplicación aíslan los mensajes y/o datos problemáticos o, directamente, son diseñadas para ser inmunes a estas condiciones. De esta manera, se busca que un componente de aplicación haga

frente a mensajes duplicados y datos inconsistentes sin dar lugar a la ejecución de funciones duplicadas.

8.3 Vínculos Arquitectónicos

Se entiende por vínculo a una conexión específica que ha sido establecida entre dos o más componentes arquitectónicos con el objetivo de indicar algún tipo de dependencia o relación entre los mismos.

Del estudio de los patrones de diseño analizados se evidencian tres formatos diferentes en los vínculos arquitectónicos, a saber:

i) *Vínculos de aplicación*: Conexiones establecidas entre componentes de alto nivel (componentes de aplicación definidos, componentes de aplicación no definidos y componentes de administración) con el objetivo de indicar una interacción funcional.
ii) *Vínculos de flujo*: Conexiones entre componentes funcionales a fin de detallar una función compleja en base a una secuencia de control de funciones elementales.
iii) *Vínculos especiales*: Conexiones establecidas entre componentes de administración y componentes de aplicación específicos a fin de gestionar la cantidad de réplicas de componentes y la asignación de recursos de tecnología de la información.

En base a estos formatos, en la Figura 8.2 se esquematizan las clases de vínculos arquitectónicos identificados. La primera división contempla el nivel de aplicación afectado por la conexión establecida, dando lugar a vínculos externos (entre componentes de alto nivel) e internos (entre componentes funcionales). A su vez, las conexiones externas se clasifican según su tipo como comunes o especiales. En el caso de los vínculos externos comunes, se identifican dos clases según su sentido: influencias (unidireccional) e interacciones (bidireccional).

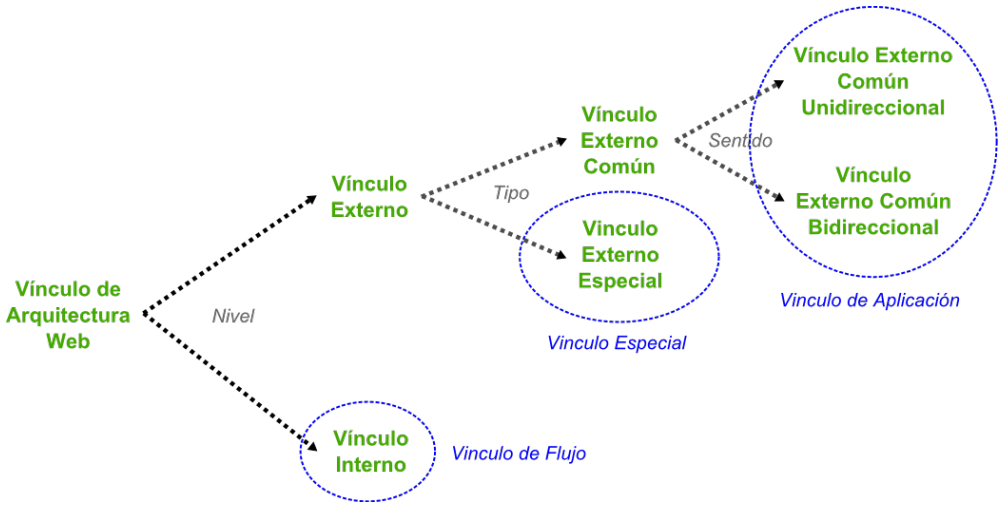


Figura 8.2. Clases de vínculos definidos para arquitecturas de software web.

8.4 Metamodelo de Diseño para Arquitecturas Web

Tal como se ha enunciado en los capítulos previos, durante las primeras etapas del desarrollo de una aplicación web, el arquitecto de software debe trabajar en el diseño de una arquitectura que, a futuro, pueda desplegarse sobre la infraestructura subyacente. En este contexto, es necesario diseñar una arquitectura de aplicación que satisfaga los requerimientos (funcionales y no funcionales) pero que, al mismo tiempo en el entorno de CC, pueda ser vinculada a las ofertas de infraestructura disponibles (a fin de ajustar el diseño resultante a las necesidades específicas). Para estos fines, en el apartado *"Proceso de Diseño: Pasos, Pautas y Lineamientos Generales"* se ha definido un esquema de trabajo que contribuye a guiar el proceso de construcción asociado a una arquitectura web.

Se define como *descripción arquitectónica* a la representación de un sistema que puede ser utilizada para su diseño, el análisis de su diseño o para la instanciación del sistema (Albin 2003). Usualmente, el conjunto de elementos que se utiliza para describir una arquitectura incluye componentes, conectores y restricciones arquitectónicas. Tomando en consideración los componentes y vínculos descriptos en las secciones

precedentes junto con los patrones de diseño que les dieron origen, en (Blas, Gonnet y Leone 2015) se propone un metamodelo que facilita la instanciación de arquitecturas web basadas en estos elementos.

Este metamodelo incluye dos especificaciones complementarias, a saber: *i)* una descripción UML (que define el conjunto de clases, relaciones y atributos necesarios para modelar las arquitecturas de software bajo estudio), y *ii)* un conjunto de restricciones OCL (que garantizan la consistencia de los modelos arquitectónicos instanciados a partir de los objetos UML).

8.4.1 Descripción UML

La Figura 8.3 presenta el modelo de clases diseñado como parte del metamodelo. Como puede observarse, se divide en cuatro grupos: *Descripción de Aplicación en la Nube* (*Cloud Application Description*), *Descripción de Descomposición de Capas* (*Layer Decomposition Description*), *Descripción de Componentes Arquitectónicos* (*Architectural Components Description*) y *Descripción de Estado de Componentes* (*Components State Description*).

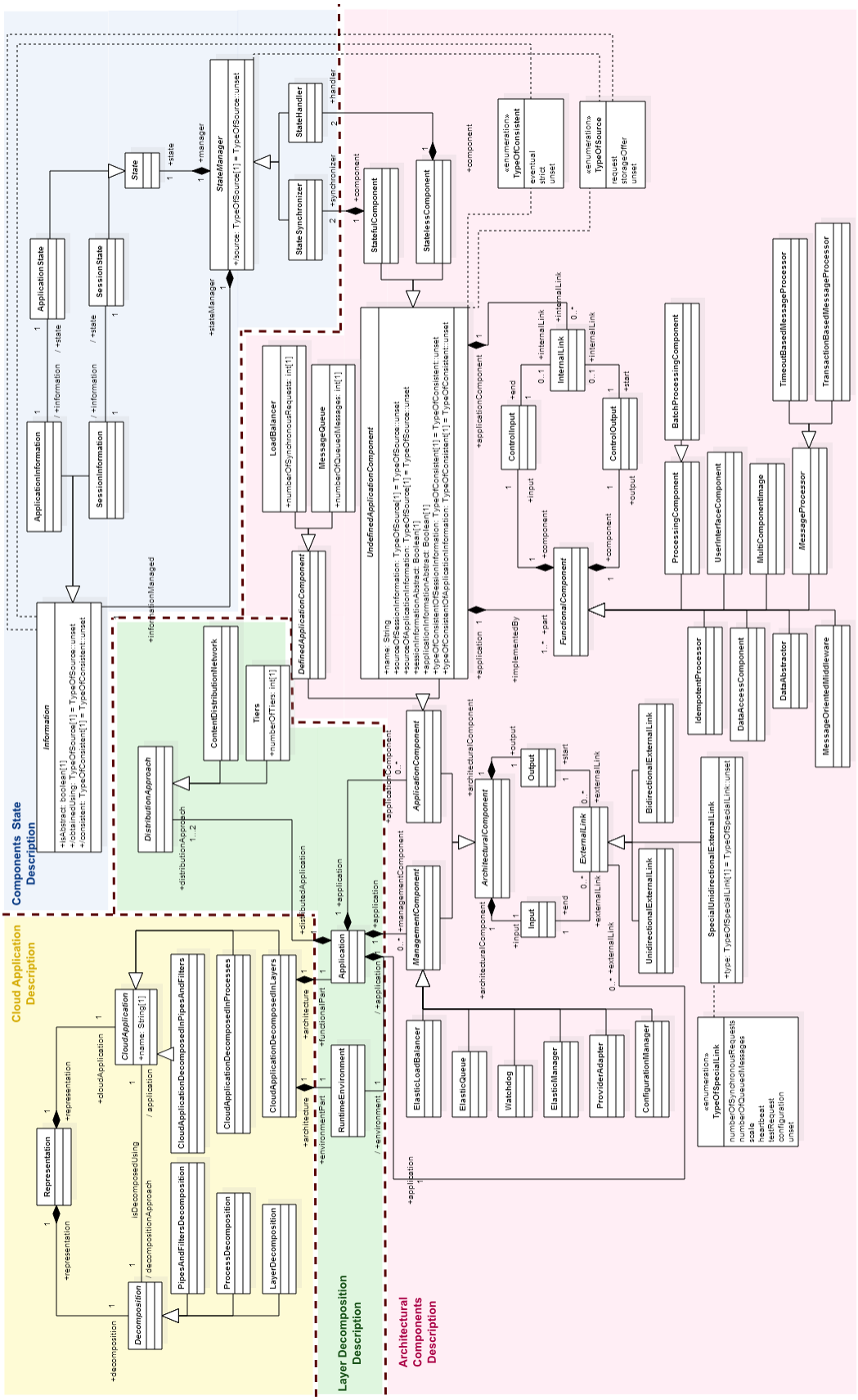


Figura 8.3. Metamodelo para la representación de arquitecturas de software web.

Metamodelo (Parte 1): Aplicación en la Nube

El conjunto de elementos agrupados en la sección *Cloud Application Description* (Figura 8.4) permite especificar la arquitectura del entorno de CC sobre el cual se despliega la aplicación web bajo diseño.

Esta especificación se logra instanciando la clase *Representation*. Una instancia de *Representation* debe contener una aplicación cloud (concepto *CloudApplication*) y un enfoque de descomposición asociado a la arquitectura de alto nivel del entorno (clase *Decomposition*). Ambos elementos modelan aspectos genéricos del CC, por lo que es necesario que sus instancias se relacionen con aspectos específicos de un entorno particular. Por este motivo, las instancias que se utilicen para describir el concepto *Representation* deben corresponderse (según la estrategia de descomposición utilizada) con alguna de las clases que especializan los términos abstractos definidos.

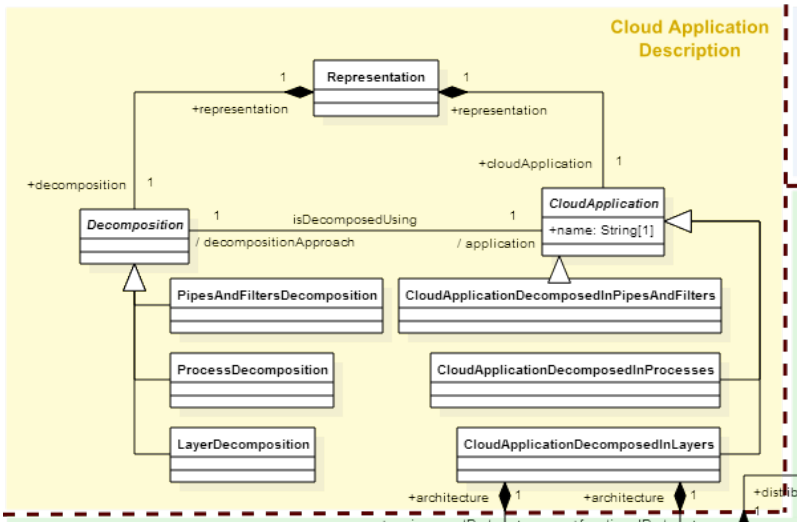


Figura 8.4. Metamodelo (parte 1) - Aplicación en la nube.

Tal como se ha explicitado con anterioridad, el enfoque de alto nivel adoptado en esta tesis para el estudio de las aplicaciones web se corresponde con una descomposición basada en capas (apartado *"Arquitecturas de Capas para Computación en la Nube"*). Luego, el conjunto de elementos modelado para la construcción de la

arquitectura del entorno de CC sobre el cual se diseñan las aplicaciones web, se desprende relaciones de composición establecidas a partir de la clase *CloudApplicationDecomposedInLayers*.

Metamodelo (Parte 2): Descomposición en Capas

En la sección *Layer Decomposition Description* se agrupan los elementos que definen una descomposición arquitectónica de alto nivel basada en capas (Figura 8.5).

Como se ha mencionado en el apartado previo, una instancia de la clase *CloudApplicationDecomposedInLayers* se define en función de dos elementos estructurales: *RuntimeEnvironment* y *Application*. En este contexto, la clase *RuntimeEnvironment* representa el conjunto de capas inferiores que componen el entorno de CC (es decir, la infraestructura sobre la cual se ejecuta la aplicación web), mientras que la clase *Application* modela la capa sobre la cual se diseña el software web. Teniendo en cuenta que el objetivo del metamodelo es especificar el conjunto de conceptos y relaciones requeridos para el diseño arquitectónico de aplicaciones web, no se incorpora mayor nivel de detalle para ampliar la descripción de la clase *RuntimeEnvironment*.

En relación a la descripción de la capa de aplicación, la instanciación de la clase *Application* implica la especificación de los componentes que forman parte de la arquitectura (detallados en el grupo *Architectural Components Description*) y la definición del enfoque de distribución que se utilizara como base para la estructuración del diseño de los componentes (clase *DistributionApproach*). De acuerdo a lo indicado en el apartado *"Patrón de Diseño de la Aplicación"*, dicho enfoque puede corresponder a una red de distribución de contenido (clase *ContentDistributionNetwork*) o a una división basada en bandas (clase *Tiers*).

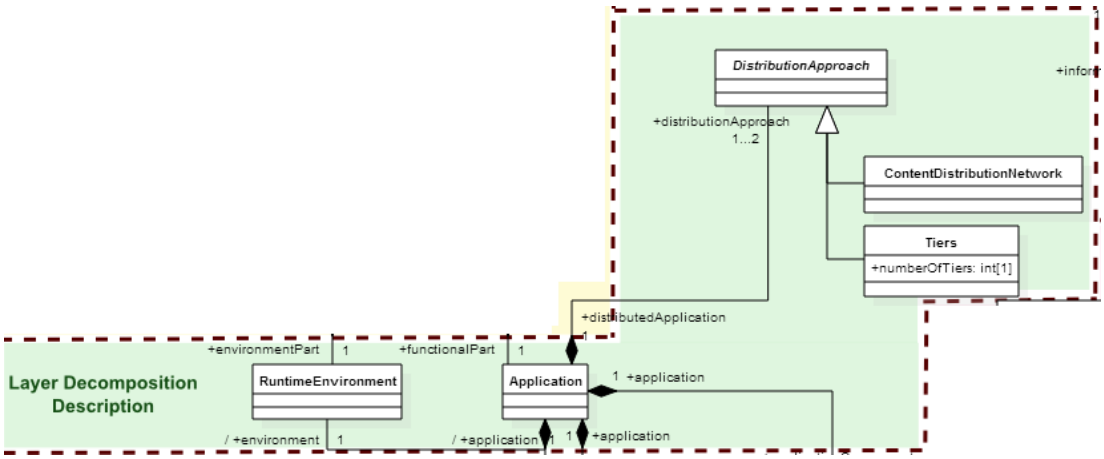


Figura 8.5. Metamodelo (parte 2) – Descomposición en capas.

Metamodelo (Parte 3): Componentes Arquitectónicos

Una vez definida la estructura general del entorno de CC que da soporte a la aplicación, deben utilizarse los componentes agrupados en la sección *Architectural Components Description* para especificar el diseño de la arquitectura del producto de software asociado a la aplicación web (Figura 8.6).

El grupo de elementos modelado en esta sección se divide en función de tres tipos de clases: clases que modelan componentes de alto nivel, clases que modelan componentes de bajo nivel y clases que modelan conectores. La jerarquía de conceptos que se desprende de la clase *ArchitecturalComponent* modela el conjunto de componentes de alto nivel a ser utilizado como base para el diseño de la arquitectura. Esta jerarquía incluye los componentes de aplicación (concepto *ApplicationComponent*) y los componentes de administración (concepto *ManagementComponent*). Por su parte, los conceptos modelados a partir de la clase *FunctionalComponent* definen los componentes de bajo nivel a ser utilizados como base para definir el comportamiento complejo de los componentes de alto nivel. Finalmente, los conceptos *ExternalLink* e *InternalLink* (junto con las clases que los especializan) definen el conjunto de relaciones disponibles para modelar vínculos entre los distintos tipos de componentes.

En términos generales, un *ArchitecturalComponent* tiene una entrada (clase *Input*) y una salida (clase *Output*) sobre las cuales se indican las conexiones entre componentes de alto nivel. Estas conexiones se especifican haciendo uso de las clases que heredan del concepto abstracto *ExternalLink* (conceptos *UnidirectionalExternalLink*, *BidirectionalExternalLink* y *SpecialUnidirectionalExternalLink*).

Los componentes que gestionan el desempeño de los componentes de aplicación (descriptos en el apartado *"Componentes de Administración"*) se encuentran modelados individualmente como especializaciones de la clase *ManagementComponent*. La misma estrategia ha sido utilizada para modelar los componentes *"Balanceador de Carga"* (clase *LoadBalancer*) y *"Cola de Mensajes"* (clase *MessageQueue*) que se derivan del concepto *DefinedApplicationComponent* (el cual modela de forma genérica un componente de aplicación definido).

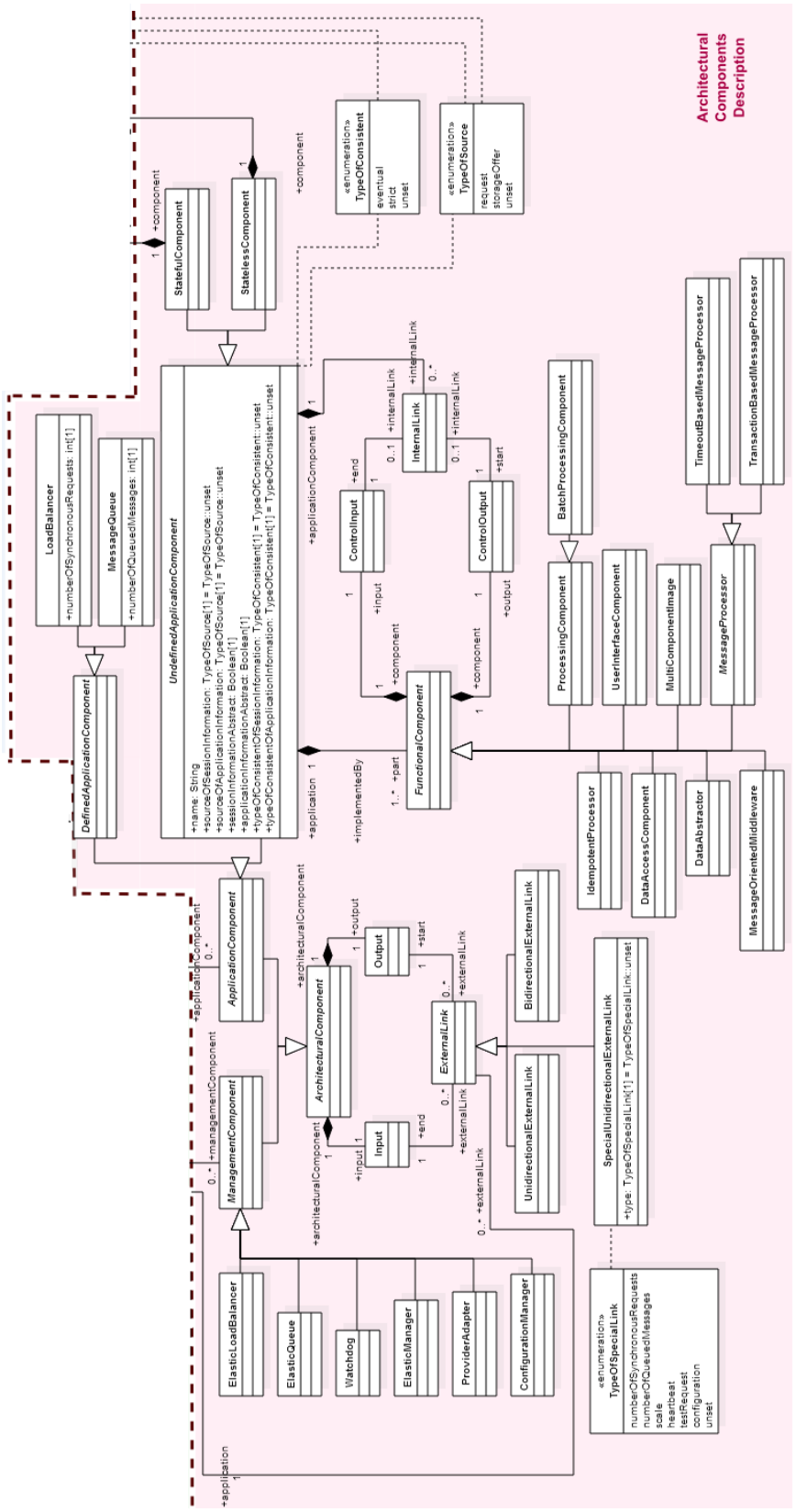


Figura 8.6. Metamodelo (parte 3) – Componentes arquitectónicos.

Los componentes de aplicación no definidos (es decir, aquellos que el arquitecto de software debe definir explícitamente a fin de dar respuesta a los requerimientos funcionales específicos de la aplicación) deben ser modelados haciendo uso de las especializaciones de la clase abstracta *UndefinedApplicationComponent*. El concepto StatelessComponent refiere a un componente de aplicación cuyo estado se maneja de forma externa (es decir, un *"Componente de Aplicación sin Estado"*). Por otro lado, el concepto *StatefulComponent* refiere a componentes de aplicación en los cuales todas las instancias deben sincronizar su estado para proveer un comportamiento uniforme (esto es, un *"Componente de Aplicación con Estado"*).

La especificación de una instancia de *UndefinedApplicationComponent* debe realizarse en función de los conceptos derivados de la clase *FunctionalComponent*. Cada uno de estos conceptos modela una responsabilidad funcional específica de acuerdo a los elementos definidos en el apartado *"Componentes Funcionales"*. Su combinación como parte de un componente de aplicación no definido permite especificar nuevas funcionalidades haciendo uso de funciones básicas. Al igual que en el caso de los componentes de alto nivel, los componentes funcionales disponen de una entrada (clase *ControlInput*) y una salida (clase *ControlOutput*) sobre las cuales se detallan los vínculos de interacción entre componentes. En este caso, tales conexiones quedan definidas por medio de la instanciación de la clase *InternalLink*.

Metamodelo (Parte 4): Estado de Componentes

La sección *Components State Description* contiene el conjunto de elementos que debe ser utilizado para especificar el manejo del estado en los componentes de aplicación definidos por el arquitecto (Figura 8.7).

El concepto *StateManager* modela un elemento genérico de control que se encarga de gestionar los estados específicos (interno o externo) según el tipo de componente al cual se encuentre asociado, a saber:

- En el caso de un *StatelessComponent* se utiliza un *StateHandler*.
- En el caso de un *StatefulComponent* se utiliza un *StateSynchronizer*.

Como se ha indicado en el apartado *"Definir los Mecanismos de Manejo del Estado"*, cada componente de aplicación tiene dos posibles estados a ser manejados: estado de sesión y estado de aplicación. En este contexto, el estado a ser gestionado por una instancia de *StateManager* debe vincularse a una instancia de la clase *SessionState* o *ApplicationState* a fin de modelar esta dependencia. A su vez, la instancia de estado debe asociarse al tipo de información a ser manipulada (la cual queda representada por las clases *SessionInformation* y *ApplicationInformation*).

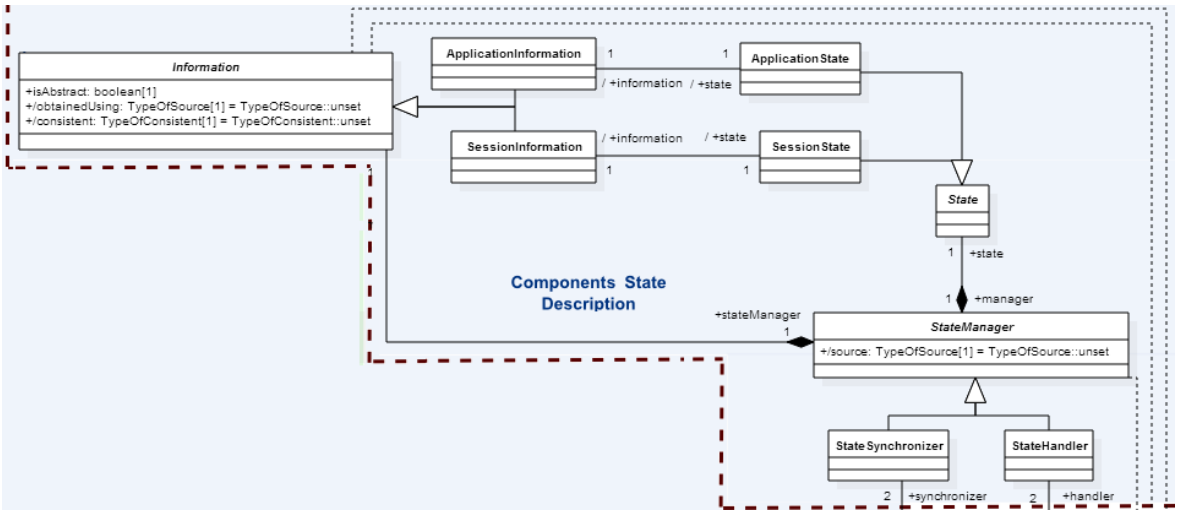


Figura 8.7. Metamodelo (parte 4) – Estado de componentes.

8.4.2 Restricciones OCL

Debido a que el conjunto de componentes y conectores diseñado en la descripción UML depende de los patrones de diseño arquitectónicos estudiados, no todos los diseños libremente construidos son válidos. Por este motivo, a fin de verificar la consistencia de los diseños arquitectónicos de las aplicaciones web, se establecieron restricciones OCL como complemento de los elementos definidos.

Estas restricciones ayudan a la validación de los diseños instanciados utilizando las clases propuestas, limitando su composición y garantizando la correcta aplicación de los patrones de diseño analizados. De esta manera, el metamodelo propuesto permite instanciar arquitecturas web válidas en base a los elementos y las reglas impuestas en cada uno de los componentes que dan lugar a los patrones.

Las restricciones OCL (formuladas como invariantes) que permiten validar la correctitud de los modelos instanciados a partir del metamodelo descripto se detallan en el *Apéndice B*.

8.5 Ejemplo de Instanciación del Metamodelo

A fin de ejemplificar la forma en la cual los elementos definidos en el metamodelo (clases y relaciones) pueden ser utilizados para instanciar un patrón especifico, se toma como ejemplo el patrón de diseño asociado al manejo elástico de componentes basado en *Elastic Load Balancer* (Figura 8.8).

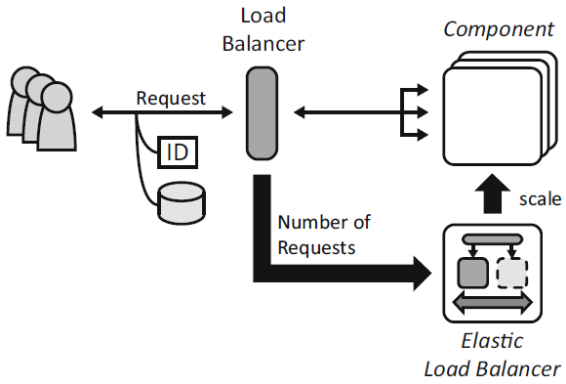


Figura 8.8. Patrón basado en el componente Elastic Load Balancer[2].

[2] Tomado de (Fehling y colab. 2014).

Como puede observarse, el patrón consta de seis elementos arquitectónicos: *Load Balancer*, *Elastic Load Balancer*, *Component*, una interacción entre *Load Balancer* y *Component*, un vínculo *number of requests* y un vínculo *scale*. Cada uno de estos elementos corresponde de forma directa a una clase del metamodelo de acuerdo al mapeo presentado en la Tabla 8.5. En este caso, el elemento *Component* es mapeado un componente de aplicación no definido ya que se modela un patrón genérico (sin funcionalidades de dominio).

Además, a fin de verificar la correcta conformación del patrón resultante a partir de las clases detalladas, se aplican el conjunto de invariantes OCL con identificador 18 a 24 (descriptas en el *Apéndice B*).

PATRÓN	**CLASE DEL METAMODELO**
Load Balancer	*LoadBalancer*
Elastic Load Balancer	*ElasticLoadBalancer*
Component	*UndefinedApplicationComponent*
Interacción *Load Balancer* y *Component*	*BidirectionalExternalLink*
Vínculo *number of requests*	*SpecialUnidirectionalExternalLink*
	Observaciones: Atributo *type* en *TypeOfSpecialLink::numberOfSynchronousRequests*
Vínculo *scale*	*SpecialUnidirectionalExternalLink*
	Observaciones: Atributo *type* en *TypeOfSpecialLink::scale*.

Tabla 8.5. Elementos del metamodelo que describen el patrón Elastic Load Balancer.

8.6 Herramienta de Modelado Complementaria

Aunque el principal objetivo de cualquier descripción arquitectónica es brindar un mecanismo de soporte para el proceso de diseño, también es deseable que este mecanismo pueda llevarse a un formato computacional que permita su análisis y posterior instanciación (Albin 2003).

En este contexto, siguiendo como base el metamodelo descripto en los apartados precedentes, se decidió implementar una herramienta de software para la construcción de modelos que sirva como soporte en la tarea de diseño ayudando al arquitecto de software a especificar aplicaciones web. Teniendo en cuenta que la plataforma Eclipse[3] es uno de los entornos de desarrollo más difundidos para la construcción de proyectos de software, la herramienta de modelado propuesta fue implementada como un *plug-in* para dicho entorno. La Figura 8.9 visualiza la arquitectura diseñada para su construcción.

Como puede observarse, las herramientas provistas como parte del proyecto *Modeling*[4] fueron claves en el diseño de la arquitectura. Este proyecto promociona el desarrollo de tecnologías asociadas a herramientas y estándares de implementación para el modelado de sistemas. Incluye un conjunto de plug-ins destinados a la implementación de herramientas de desarrollo dirigidas por modelos, entre los que se destacan *Eclipse Modeling Framework* (Steinberg y colab. 2008) y *Graphical Modeling Framework* (Gronback 2009). *Eclipse Modeling Framework* (EMF) se utiliza para la especificación de los modelos *Ecore* que describen los elementos componentes que forman la base de la herramienta de modelado bajo desarrollo. En este contexto, *Ecore* es un lenguaje para la especificación de metamodelos que pueden ser manipulados por la plataforma Eclipse mediante el uso de paquetes, clases y relaciones especiales. Por su parte, *Graphical Modeling Framework* (GMF) se emplea para la especificación de la notación gráfica que acompaña la descripción de los distintos elementos componentes. Complementariamente, el proyecto *Sirius*[5] aprovecha la tecnología provista en el marco del proyecto *Modeling* con el objetivo de facilitar la creación de ambientes de trabajo dirigidos por modelos gráficos.

[3] Disponible en *http://www.eclipse.org/*
[4] Disponible en *https://eclipse.org/modeling/*
[5] Disponible en *https://eclipse.org/sirius/*

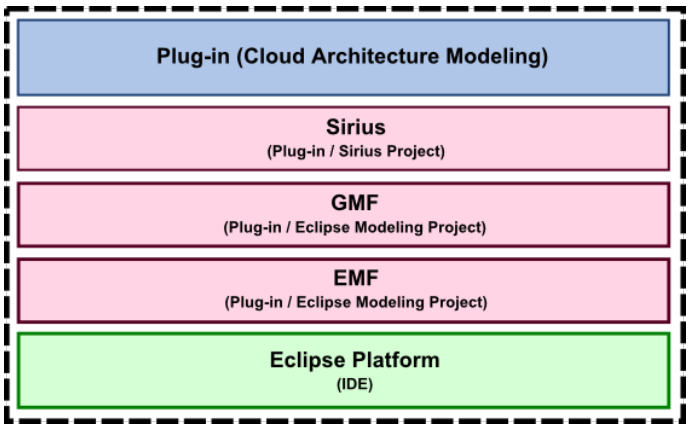


Figura 8.9. Arquitectura del plug-in Eclipse (basado en el proyecto Modeling).

En el caso de la herramienta de modelado de arquitecturas web, el diagrama de clases presentado en la Figura 8.3 como descripción del metamodelo fue implementado como modelo Ecore (haciendo uso de la herramienta EMF). Las restricciones OCL que complementan este metamodelo (detalladas en el *Apéndice B*) se incorporaron como parte de esta implementación a fin de verificar los modelos generados a partir de su instanciación.

Teniendo en cuenta que las clases del metamodelo se corresponden con un conjunto de elementos claramente definidos, las descripciones gráficas utilizadas para los distintos componentes arquitectónicos son consistentes con los íconos propuestos en la bibliografía utilizada. Estas descripciones se especificaron utilizando Sirius y, posteriormente, se mapearon a los elementos conceptuales definidos en el modelo Ecore.

La Figura 8.10 visualiza la pantalla principal del plug-in implementado (disponible en *PluginArchMdl*). Como puede observarse, se detallan cuatro grandes áreas de trabajo:

- *Parte 1*: Explorador de proyectos en el cual se detalla el proyecto de software *Test* sobre el cual se diseña la arquitectura web.
- *Parte 2*: Área de trabajo sobre la cual el arquitecto debe generar su diseño.

- *Parte 3*: Paleta de herramientas de diseño que contiene el conjunto de componentes y relaciones arquitectónicas disponibles para la creación del diseño.
- *Parte 4*: Tabla de propiedades en la cual se despliegan los atributos requeridos en relación a los distintos elementos contenidos en el diagrama.

Luego, en relación a un proyecto de software específico (Parte 1), el arquitecto de software construye sus diseños web (en la Parte 2) realizando un *drag and drop* de los elementos arquitectónicos disponibles (de la Parte 3) y configurando su definición de acuerdo a las propiedades requeridas en cada caso (Parte 4).

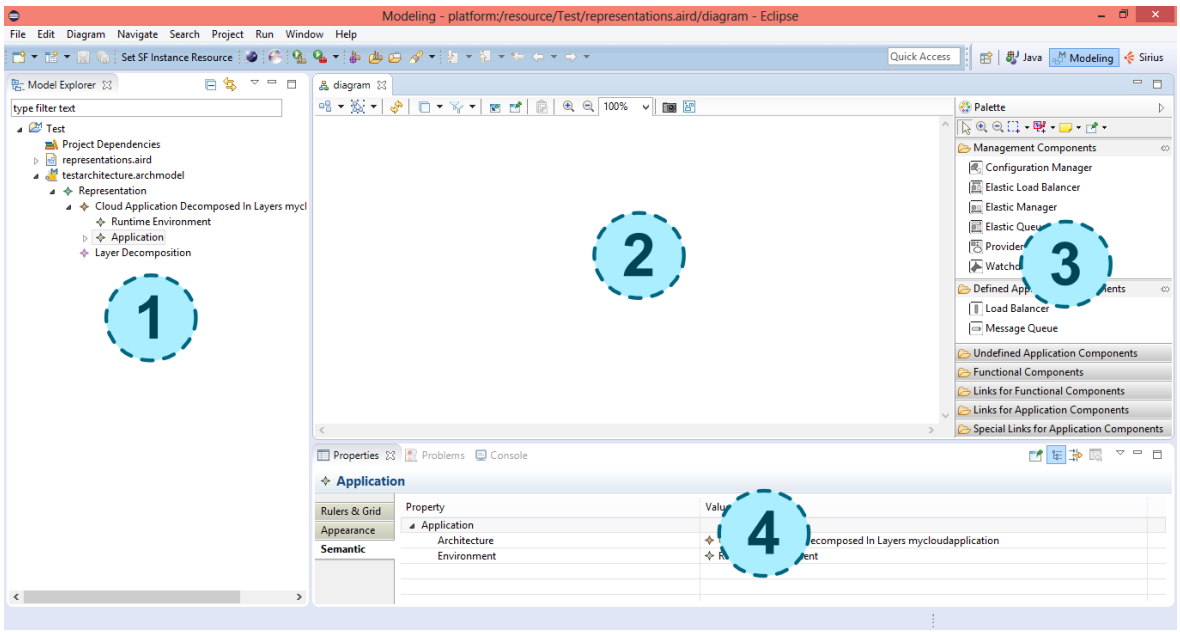


Figura 8.10. Pantalla principal del plug-in que permite diseñar arquitecturas web.

La verificación de los diseños generados se realiza ejecutando la opción *OCL → Validate* sobre el diseño arquitectónico final (tal como se visualiza en la Figura 8.11). Esta función realiza la comprobación de las invariantes OCL (como parte del proceso de verificación del modelo) sobre el conjunto de instancias creadas y configuradas de acuerdo a las decisiones arquitectónicas tomadas por el arquitecto. Ante una eventual violación de una o más restricciones, se informa al arquitecto la situación indicando los

problemas detectados (con el mayor nivel de detalle posible) en relación a los componentes definidos. Por el contrario, si se cumplen todas las restricciones, se comunica al arquitecto que el diseño realizado no viola las restricciones definidas (Figura 8.12).

Además de las funcionalidades de modelado gráfico, la herramienta incluye un asistente de creación que ayuda al arquitecto en la especificación de características propias del entorno de CC sobre el cual se despliega la aplicación web diseñada. Este asistente permite contextualizar el diseño arquitectónico como parte de un proyecto de software asociado a una estructura de CC (la cual queda definida haciendo uso de los elementos detallados en los apartados *"Metamodelo (Parte 1): Aplicación en la Nube"* y *"Metamodelo (Parte 2): Descomposición en Capas"*).

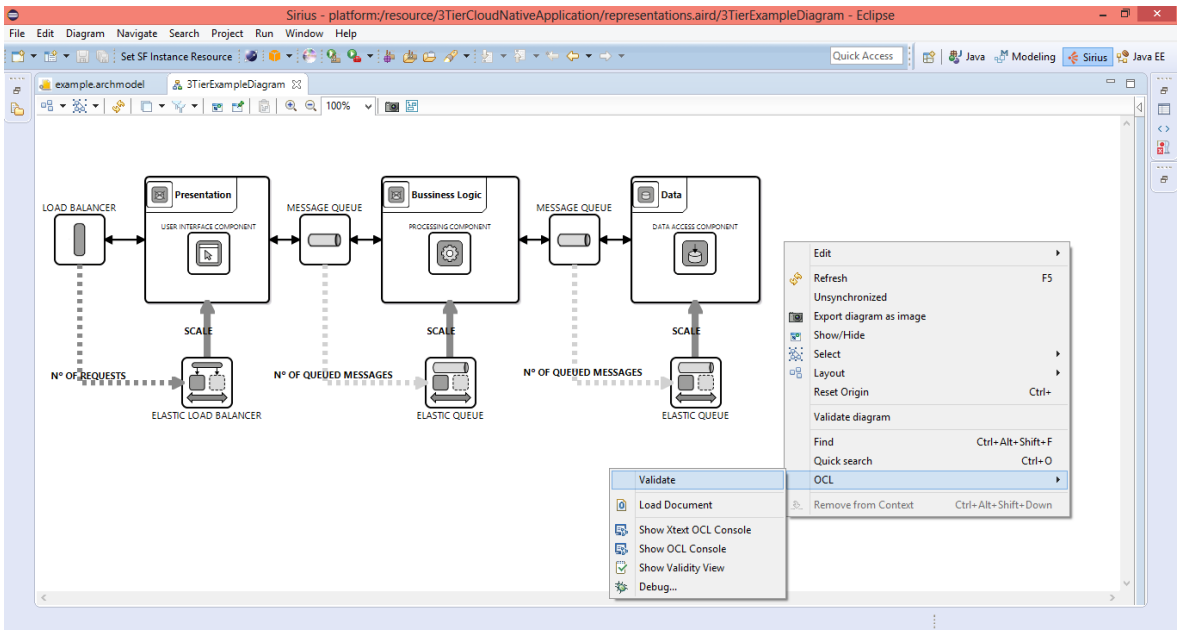


Figura 8.11. Verificación del diseño de una arquitectura web (parte 1).

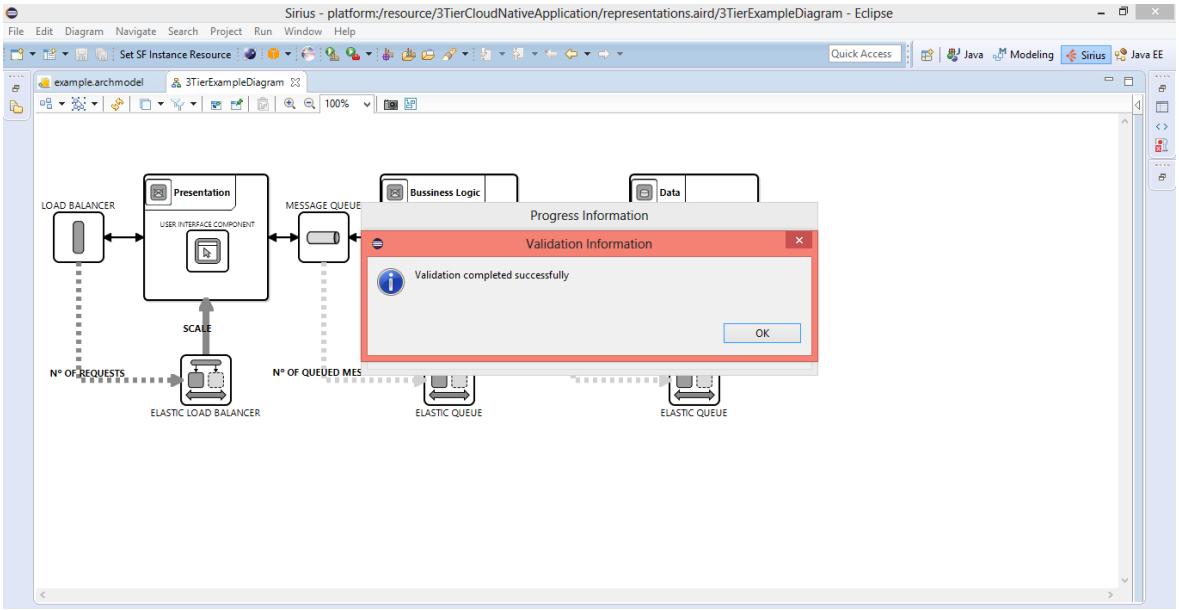


Figura 8.12. Verificación del diseño de una arquitectura web (parte 2).

En (Blas, Gonnet y Leone 2015) se ejemplifica la instanciación del metamodelo propuesto utilizando la herramienta de modelado desarrollada en base a dos casos de estudio: *i)* una aplicación genérica de tres bandas en la cual existen múltiples componentes arquitectónicos vinculados (de estructura similar al diseño que se visualiza en la Figura 8.11), y *ii)* la especificación de la arquitectura de software del caso *AWS Case Study: Unilever*[6].

8.7 Proceso de Diseño basado en los Patrones Arquitectónicos

Haciendo uso de los elementos definidos en el metamodelo desarrollado, en (Blas, Gonnet y Leone 2017c) se especializa el proceso presentado en el apartado *"Proceso de Diseño: Pasos, Pautas y Lineamientos Generales"* con el objetivo de guiar al arquitecto durante la tarea de construcción de arquitecturas de aplicaciones web

[6] Disponible en *https://aws.amazon.com/es/solutions/case-studies/unilever/*

basadas en los patrones de diseño analizados. Cada uno de los pasos identificados como parte de la actividad propuesta, involucra de forma explícita un conjunto de elementos definidos en el metamodelo.

La violación de alguna de estas restricciones OCL impuestas sobre las instancias del metamodelo implica que el diseño formulado no se adecúa a los patrones de diseño propuestos, por lo que el arquitecto debe modificarlo a fin de garantizar su consistencia. El impacto de este tipo de cambios implica volver a los pasos previos de la actividad de diseño a fin de incorporar, modificar y/o eliminar componentes y/o vínculos en virtud de corregir los problemas detectados en el diseño actual.

De esta manera, si el arquitecto de software sigue el conjunto de pasos propuesto al mismo tiempo que utiliza la herramienta de modelado implementada, obtiene un diseño arquitectónico asociado a una aplicación web en el que tanto sus componentes como sus conexiones están verificados conforme las restricciones de diseño planteadas en los patrones estudiados.

Conclusiones

En el contexto de computación en la nube (al igual que en la ingeniería de software tradicional), el uso de patrones de diseño se ha transformado en un constructor de utilidad para los arquitectos encargados de la elaboración del diseño estructural. Sin embargo, a raíz de la rapidez con la cual han surgido estos entornos, no todos los patrones de diseño propuestos en la literatura definen de forma adecuada un conjunto de componentes y relaciones arquitectónicas que posibiliten una clara separación entre el nivel de aplicaciones web y el nivel de infraestructura.

En virtud de proveer una solución a este problema, en este capítulo se ha especificado un metamodelo de componentes y conectores que ayuda a modelar arquitecturas de servicios de software para aplicaciones web. El modelo diseñado toma como base un conjunto de componentes y vínculos arquitectónicos que han sido obtenidos a partir del estudio de patrones de diseño específicos. Su implementación se basa en una descripción UML complementada con un conjunto de invariantes OCL. Además, se ha presentado una herramienta de software que brinda soporte a la tarea de diseño arquitectónico haciendo uso del metamodelo propuesto, la cual posibilita la evaluación de la arquitectura resultante a fin de verificar su estructura conforme los patrones de diseño incluidos.

De esta manera, se sientan las bases de diseño que dan lugar a la construcción de modelos de simulación que permiten analizar las características propias de las aplicaciones web y los aspectos de calidad asociados a las representaciones generadas para este tipo de arquitecturas.

Parte IV

Modelos de Simulación para Aplicaciones Web

Capítulo 9. Formalismos de Simulación basados en Eventos Discretos

El formalismo DEVS proporciona una base común para el modelado y simulación de eventos discretos. A lo largo de los años, se han desarrollado múltiples extensiones de este formalismo en virtud de resolver diferentes tipos de situaciones. Sin embargo, cuando la decisión de aceptar o rechazar un evento de entrada (generado específicamente por medio de un esquema de distribución de eventos de salida) se relaciona con las capacidades propias del modelo de simulación, la aplicación de las extensiones existentes genera soluciones de modelado complejas. En este capítulo se presenta una nueva extensión denominada RDEVS en la que se utiliza información de ruteo para gestionar un conjunto de eventos dirigidos. Este formalismo ha sido diseñado como una subclase de DEVS que provee una adaptación del formalismo original para describir procesos de ruteo en modelos de eventos discretos asociados a cualquier dominio. Como parte de la presentación del formalismo, se detalla su especificación formal (utilizando teoría de conjuntos) junto con la prueba de clausura bajo acoplamiento (la cual garantiza que los modelos propuestos pueden ser construidos de forma jerárquica). Se incluye además una breve descripción del framework desarrollado como herramienta para la construcción y simulación de modelos RDEVS en Java.

9.1 Discrete Event System Specification (DEVS)

El formalismo de Especificación de Sistemas de Eventos Discretos (DEVS por sus siglas en inglés, Discrete Event System Specification) es un formalismo modular y jerárquico basado en la teoría de sistemas que proporciona una metodología general para la construcción de modelos de simulación de eventos discretos que pueden ser reutilizados (Zeigler, Praehofer y Kim 2000).

Desde su introducción a finales de los 70 (Zeigler 1976), este formalismo se ha utilizado como base para la simulación de múltiples sistemas vinculados a diversos dominios. Algunas de las aplicaciones más relevantes que pueden encontrarse en la literatura refieren al dominio de redes sociales (Bouanan y colab. 2016), aplicaciones móviles (Kim y colab. 2015), gestión de cadenas de suministro (Gholami y colab. 2014), sistemas biológicos (Uhrmacher y Kuttler 2006; Uhrmacher y colab. 2007), dispositivos y estrategias militares (Wainer y Madhoun 2005; Moallemi y colab. 2010) y sistemas de comercio electrónico (Chezzi, Tymoschuk y Lerman 2013).

9.1.1 Especificación de Modelos: DEVS Atómicos y DEVS Acoplados

El formalismo DEVS describe el comportamiento de un sistema en base a dos niveles de descomposición: descripción de comportamiento (*modelo atómico*) y descripción de estructura (*modelo acoplado*).

En el nivel más bajo, un *modelo DEVS atómico* describe el comportamiento autónomo de un sistema de eventos discretos como una secuencia de transiciones deterministas entre estados secuenciales junto con la forma en la cual esta secuencia: *i)* reacciona ante eventos de entrada, y *ii)* genera eventos de salida. Por su parte en el nivel superior, un *modelo DEVS acoplado* describe un sistema como una red de componentes ensamblados que pueden quedar definidos, a su vez, como *modelos atómicos* o *modelos acoplados*.

Para mayores detalles acerca del formalismo original y sus propiedades, se sugiere recurrir a (Zeigler, Praehofer y Kim 2000).

Modelo Atómico (Atomic DEVS)

Un modelo atómico DEVS (Ecuación 9.1) queda definido como

$$M = (X , Y , S , \delta_{int} , \delta_{ext} , \lambda , ta) \qquad (9.1)$$

donde:

X es el conjunto de valores de entrada.

Y es el conjunto de valores de salida.

S es el conjunto de valores de los estados.

$\delta_{int}: S \rightarrow S$ es la función de transición interna.

$\delta_{ext}: Q \times X \rightarrow S$ es la función de transición externa, en la cual:

$Q = \{ (s,e) \mid s \in S, 0 \leq e \leq ta(s) \}$ es el conjunto de estados totales.

e es el tiempo transcurrido desde la última transición.

$\lambda: S \rightarrow Y$ es la función de salida.

$ta: S \rightarrow \mathbb{R}^{+}_{0,\infty}$ es la función de avance de tiempo.

Modelo Acoplado (Coupled DEVS)

Un modelo acoplado DEVS (Ecuación 9.2) queda definido como

$$N = (X , Y , D , \{ M_d \} , \{ I_d \} , \{ Z_{i,d} \} , Select) \qquad (9.2)$$

donde:

X es el conjunto de valores de entrada.

Y es el conjunto de valores de salida.

D es el conjunto de referencias a componentes.

Para cada $d \in D$, se tiene que:

$M_d = (X_d , Y_d , S_d , \delta_{int,d} , \delta_{ext,d} , \lambda_d , ta_d)$ es un modelo DEVS que compone la estructura acoplada.

I_d es el conjunto de influyentes sobre d en el cual $d \subseteq D \cup \{ N \}$, $d \notin I_d$.

Para cada $i \in I_d$, $Z_{i,d}$ es la función de traducción de salidas, en la cual:

$Z_{i,d}: X \rightarrow X_d$, si $i = N$.

$Z_{i,d}: Y_i \rightarrow Y$, si $d = N$.

$Z_{i,d}: Y_i \rightarrow X_d$, si $i \neq N$ y $d \neq N$.

Select: $2^D \rightarrow D$ es la función desempate para el manejo de eventos simultáneos.

9.1.2 Interpretación de los Modelos

Sea M un modelo DEVS atómico que representa un sistema cuyo estado inicial es s. Si no se da ningún evento externo que altere su estado, el modelo permanece en el estado s durante una cantidad de instantes de tiempo definida por $ta(s)$. Cuando el tiempo transcurrido en el estado s (el cual queda indicado como e) es igual a la cantidad de instantes de tiempo que el modelo debe mantenerse en dicho estado (es decir, $e = ta(s)$), el sistema produce un evento de salida con valor $\lambda(s)$ y, posteriormente, cambia de estado (de s a s') por medio de la ejecución de la función $\delta_{int}(s)$. En este caso, el sistema ha efectuado una *transición interna*. Por el contrario, si antes de que se alcance

la condición de transición interna se presenta un evento externo con valor *x*, el modelo debe atenderlo. Dado que este evento ocurre cuando el sistema aún se encuentra en estado *s* habiendo transcurrido *e* instantes de tiempo en dicho estado (donde $e \leq ta(s)$), el estado total del sistema queda definido como (s,e). En este caso, el sistema no produce ningún evento de salida pero modifica su estado interno de acuerdo a la función $\delta_{ext}(s,e,x)$.

Por otra parte, sea *N* un modelo DEVS acoplado formado por elementos identificados por *d* ($d \in D$) los cuales han sido denominados M_d. Cada M_d definido se corresponde con un modelo DEVS atómico que funciona internamente de forma independiente de acuerdo al mecanismo descripto con anterioridad. En este contexto, la estructura interna del modelo *N* queda definida por medio de la vinculación de:

- Las salidas con las entradas de los diferentes M_d.
- Las entradas/salidas del propio modelo *N* con las entradas/salidas (respectivamente) de los modelos M_d.

Dado que el tipo de eventos que puede recibir un componente (es decir, los eventos de entrada) pueden no coincidir con el tipo de eventos que generan los componentes vinculados (es decir, los eventos de salida), la adaptación de la conexión entre los modelos *i* y *d* se realiza por medio del uso de la función de traducción $Z_{i.d}$. Bajo este esquema, el modelo *i* pertenece al conjunto de modelos influyentes de *d* (el cual queda definido como I_d).

Luego, un modelo M_d que compone a *N* recibe eventos de entrada (en el contexto de DEVS atómicos, eventos externos) que provienen de las salidas de otros modelos atómicos o de la propia entrada del sistema global.

Estado Transitorio y Estado Pasivo en Modelos Atómicos

Tal como se ha enunciado, la evaluación de la función $ta(s)$ corresponde al tiempo de vida en el estado *s*. Dicho tiempo queda definido como la cantidad de instantes de

tiempo que el sistema debe permanecer en *s*, siendo el resultado un valor real que pertenece al intervalo [0;∞).

En el caso que *ta(s) = 0*, el estado *s* se denomina *estado transitorio*. Una vez que el sistema ingrese al estado *s*, inmediatamente se requerirá la ejecución de una nueva transición que lo saque del mismo.

Por el contrario si *ta(s)* = ∞, una vez que el sistema ingrese al estado *s* permanecerá en dicho estado para siempre (a menos que se presente un evento externo que lo obligue a cambiar de estado). En este caso, el estado *s* se denomina *estado pasivo*.

9.2 Extensiones del Formalismo DEVS

A lo largo de los años, DEVS ha encontrado una creciente aceptación en la comunidad de investigación de la simulación basada en modelos, convirtiéndose en uno de los paradigmas preferidos para llevar a cabo estrategias de modelado y simulación (Wainer y Mosterman 2010).

En este contexto, varios autores han mejorado la especificación del formalismo para obtener nuevas variantes, extensiones y abstracciones útiles para resolver diferentes tipos de situaciones. Desde este punto de vista, las extensiones del formalismo DEVS amplían las clases de modelos de sistemas que pueden ser representados haciendo uso del formalismo original (Zeigler, Praehofer y Kim 2000). Algunas de las extensiones DEVS más utilizadas son: *Cell-DEVS* (Wainer 2004), *Dynamic Structure DEVS* (Barros 1997), *Fuzzy-DEVS* (Kwon y colab. 1996), *Min-Max-DEVS* (Hamri, Giambiasi y Frydman 2006), *Multi-Level DEVS* (Steiniger y Uhrmacher 2016), *Parallel DEVS* (Chow y Zeigler 1994), *Real Time DEVS* (Hong y colab. 1997) y *Vectorial DEVS* (Bergero y Kofman 2014).

Frecuentemente las extensiones de DEVS requieren nuevos algoritmos de simulación a fin de mejorar la ejecución del proceso de simulación haciendo uso de las características propias del nuevo enfoque propuesto. En estos casos, los autores y sus colaboradores diseñan algoritmos de simulación específicos que dan lugar a nuevos simuladores DEVS que complementan las extensiones desarrolladas (Kim, Kim y Park 1998; Muzy y Nutaro 2005; Shang y Wainer 2006; Liu 2010; Liu y Wainer 2010).

Luego, es evidente que el entorno que rodea al formalismo DEVS ha crecido junto con la evolución de los problemas a ser resueltos con técnicas de simulación basadas en eventos discretos. A medida que aumenta la complejidad de los problemas, se requieren nuevos mecanismos de soporte que contribuyan a su resolución.

(EXTENSIONES DE DEVS) *Las extensiones del formalismo DEVS amplían las clases de modelos de sistemas que pueden ser representados haciendo uso del formalismo original (Zeigler, Praehofer y Kim 2000).*

9.2.1 Fundamentos de RDEVS

El formalismo DEVS Enrutados (RDEVS por sus siglas en inglés, Routed DEVS) fue diseñado como una adaptación del formalismo DEVS que busca dar solución a la identificación de eventos como parte del procesamiento de los modelos de simulación.

Frecuentemente, como parte del comportamiento asociado a modelos DEVS, se requiere de la identificación del modelo de origen o destino de los eventos con el objetivo de decidir su ruta (es decir, su procesamiento) dentro del modelo de simulación. Este problema se denomina *problema de ruteo de eventos* (o, simplemente, *problema de ruteo*) porque afecta el diseño del modelo de simulación pero no se encuentra directamente relacionado al funcionamiento de los componentes a modelar.

Teniendo en cuenta que los acoplamientos se definen en tiempo de diseño y considerando que su función es transferir eventos de salida/entrada como mecanismo de conexión entre modelos, usualmente se agregan otros mecanismos para redirigir los eventos hacia su destino final. Bajo esta perspectiva, la resolución del problema implica que quien diseña el modelo de simulación se encuentra forzado a incorporar nuevos elementos (que se suman a la complejidad de los elementos que ya conoce -dado que es un experto del dominio de trabajo-) en virtud de resolver un problema externo. En estos casos, se suele incluir un modelo especial que actúa como *switch* a fin de decidir qué componente interno debe utilizarse para procesar el evento. Este componente se antepone al flujo de entradas a fin de capturar y redistribuir los eventos de acuerdo a un criterio predefinido como parte de su comportamiento.

Este tipo de soluciones conlleva a un aumento en la cantidad de modelos de simulación involucrados como parte del diseño final, dando lugar a un incremento del grado de composición asociado al modelo resultante (lo que de cierta manera, impacta en su complejidad). Además, el impacto de un cambio en alguna de las rutas predefinidas (es decir, la adición, modificación o eliminación de un camino) implica un ajuste del comportamiento de todos los modelos involucrados en el proceso de ruteo. Luego, la capacidad de modificar o adaptar el modelo a nuevas situaciones (en términos de ingeniería de software, la “modificabilidad” del modelo de simulación) se ve afectada negativamente, ya que el impacto de un cambio afecta a múltiples componentes.

En este contexto, el formalismo RDEVS actúa como una capa sobre el formalismo DEVS que provee funcionalidad de ruteo sin necesidad de que el diseñador (usuario) recurra a DEVS para lograr este tipo de funciones (Zeigler, Muzy y Kofman 2018). El núcleo de RDEVS consiste en la abstracción y organización del flujo de eventos de forma independiente al comportamiento de los componentes. De acuerdo con esta perspectiva, el conjunto de componentes requeridos como parte de la simulación puede ser modelado en DEVS y, posteriormente, se puede aplicar el formalismo RDEVS sobre estos modelos a fin de diseñar y simular el proceso de ruteo.

La extensión propuesta define un conjunto de modelos estructurados para procesar eventos dirigidos dentro de una red de componentes. Cada componente queda definido en base a dos elementos, a saber: *i)* una descripción de comportamiento, y *ii)* la información de ruteo específica asociada al nodo (la cual permite identificar el origen y destino de los distintos eventos vinculados al componente). Haciendo uso de la información de ruteo, los modelos RDEVS determinan la aceptación o el rechazo de los eventos de entrada en una etapa previa a ejecutar sus instrucciones. Además, tienen la capacidad de enviar eventos de salida a un grupo específico de receptores (lo que permite garantizar el procesamiento del evento por parte de uno o más destinatarios).

El Caso de las Arquitecturas de Software y las Solicitudes de Usuario

Según (Bass, Clements y Kazman 2012), las arquitecturas de software (desde un punto de vista estructural) ilustran un conjunto de componentes vinculados por conexiones (ver *"Definición de Arquitectura de Software"*). Tales conexiones se basan en las relaciones existentes entre los distintos elementos de software, las cuales son definidas con el objetivo de dar respuesta al conjunto de funcionalidades requeridas para resolver las solicitudes de los usuarios.

Luego, las conexiones definidas como parte de una arquitectura de software se derivan de la interacción que tiene lugar entre componentes cuando el software necesita resolver un tipo de solicitud específica. Por lo tanto, la especificación de las conexiones se define a nivel arquitectónico (es decir, en tiempo de diseño como dependencias entre componentes) pero su ejecución se vincula con el tipo de solicitud involucrado al momento de utilizar el componente de software (es decir, en tiempo de ejecución según el contexto actual del entorno y la respuesta que debe darse al usuario en función de su petición).

Cuando se utiliza el diseño arquitectónico como base para la construcción de un modelo de simulación, el modelo final debe reflejar todas las conexiones definidas

entre componentes y, además, debe administrar la forma en la cual se distribuyen los distintos tipos de solicitudes (es decir, debe gestionar la complejidad del flujos de solicitudes). Si las solicitudes son modeladas como eventos, la formulación de este problema puede definirse como un *problema de ruteo de eventos*.

En (Blas, Gonnet y Leone 2016b) se presenta una primera aproximación a la resolución de este problema haciendo uso del formalismo DEVS. Un análisis rápido de los modelos de simulación propuestos refleja de inmediato la complejidad de su estructura y, aunque la solución constituye un buen enfoque, se observa que no es escalable. Cuando el número de componentes arquitectónicos crece y/o se incorporan nuevos tipos de solicitudes, la configuración de los modelos involucrados en la transformación *arquitectura-modelo de simulación* se convierte en un problema (ya que el diseñador debe conocer los flujos de las solicitudes en etapa de diseño).

Ante este tipo de problemas, los modelos RDEVS son capaces de administrar el flujo de eventos utilizando su información de estado (a fin de simplificar la construcción de los modelos de simulación).

Tómese como ejemplo las representaciones arquitectónicas de aplicaciones de software web descriptas en el capítulo precedente. Si los eventos de entrada representan distintos tipos de solicitudes de usuario que navegan dentro de los componentes arquitectónicos en búsqueda de su resolución, el modelo RDEVS equivalente a una representación de este tipo administrará el flujo de solicitudes utilizando el conjunto de conexiones especificadas a nivel arquitectónico pero, a su vez, cada componente será capaz de decidir aceptar/rechazar las solicitudes según el estado actual de resolución de las mismas. Luego, al utilizar el formalismo RDEVS, los modelos de simulación diseñados tendrán el conocimiento requerido para determinar el tratamiento de las solicitudes (sin afectar la definición de su comportamiento).

9.3 Routed DEVS (RDEVS)

Tal como se ha definido con anterioridad, la premisa básica de RDEVS es que un modelo de simulación debe procesar únicamente el conjunto de eventos de entrada que le ha sido enviado específicamente por parte de un conjunto de remitentes previamente autorizados. Para garantizar esta premisa, el formalismo requiere que cada modelo de simulación que forma parte del proceso de ruteo se encuentre asociado a un identificador único que define su existencia como entidad componente de dicho esquema.

En este contexto, un modelo de simulación RDEVS acepta eventos de entrada, si y solo si, se cumple que:

i) el identificador del remitente del evento corresponde a un modelo autorizado, y
ii) el identificador del propio modelo se encuentra incluido en el conjunto de receptores habilitados para el procesamiento del evento.

Luego, haciendo uso de la información de ruteo propia de cada modelo, estas entidades tienen la capacidad de decidir si procesan o rechazan un evento de entrada y, al mismo tiempo, definir el conjunto de modelos destino que debe procesar sus eventos de salida. En este sentido, todos los eventos manipulados como parte del comportamiento de los modelos RDEVS, incluyen en su definición la información requerida para identificar tanto al modelo remitente como así también a todos los modelos destinatarios.

9.3.1 Conceptualización de Modelos: Componentes, Entidades y Procesos

El formalismo RDEVS se basa en tres tipos de modelos, los cuales han sido definidos como *modelo esencial* (*essential model*), *modelo de ruteo* (*routing model*) y *modelo de red* (*network model*).

Cada modelo representa un nivel de abstracción utilizado para definir el conjunto de elementos que forma parte de la esquematización propuesta para el manejo de la estrategia de ruteo. Estos niveles refieren a *componentes* (*components*), *entidades* (*entities*) y *procesos* (*processes*). Bajo este esquema, un *modelo de red* representa un *proceso* de ruteo que ha sido diseñado a fin de lograr un objetivo específico. Cada *proceso* queda compuesto por un conjunto de *entidades* definidas de acuerdo a *modelos de ruteo*. A su vez, cada *entidad* representa un elemento utilizado para indicar las interacciones de ruteo requeridas entre diferentes *componentes*. El conjunto de *componentes* disponibles para construir el *proceso* de ruteo se define haciendo uso de *modelos esenciales*.

La Figura 9.1 esquematiza las dependencias entre los diferentes tipos de modelos junto con su relación de acuerdo a los elementos que definen la estructura de ruteo. Muestra la representación (en términos de modelos de simulación) de dos procesos (identificados como *M1* y *M2*) que han sido construidos haciendo uso del mismo conjunto de elementos básicos.

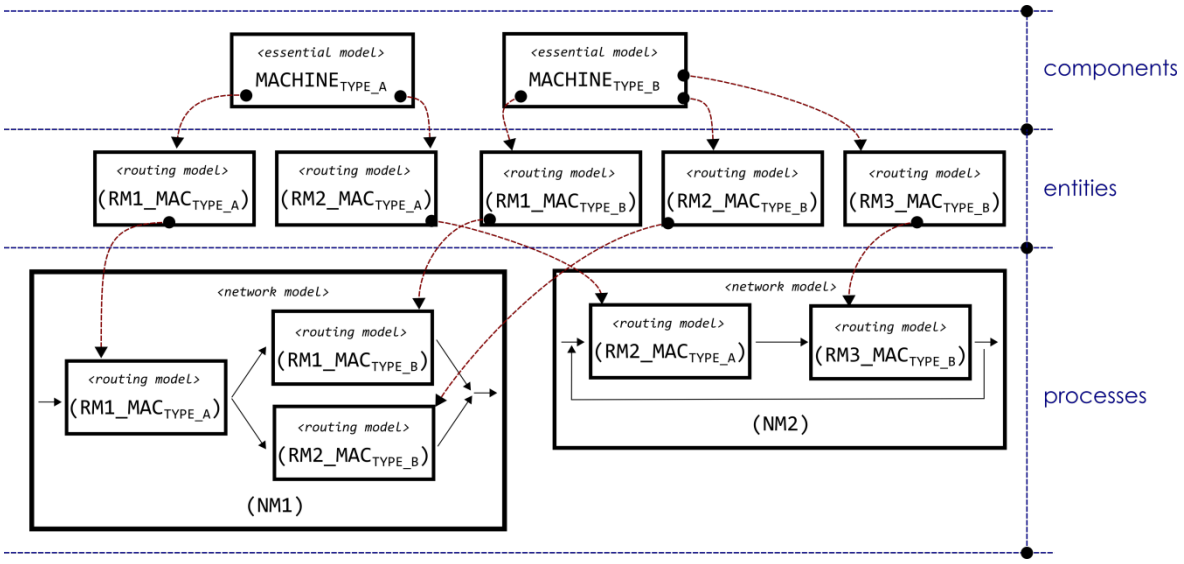


Figura 9.1. Dependencias de modelos para la definición de estructuras de ruteo.

Los *componentes* disponibles para construir los modelos de simulación asociados a los casos *M1* y *M2* se corresponden con *modelos esenciales* que representan el comportamiento de dos máquinas diferentes (denominadas de forma genérica como $MACHINE_{TYPE\_A}$ y $MACHINE_{TYPE\_B}$). Para cada *modelo esencial*, se define un conjunto de *entidades* de acuerdo a lo especificado en la Tabla 9.1. Cada *entidad* se encuentra estructurada como un *modelo de ruteo* que contiene el comportamiento de una de las dos máquinas (es decir, un *modelo esencial*) junto con la información requerida para abordar el *proceso* de ruteo sobre el *componente* previamente definido. Luego, utilizando las *entidades* diseñadas, se modelan los escenarios *M1* y *M2*. Cada escenario representa un *proceso* de ruteo estructurado como un *modelo de red* (*NM1* y *NM2*) que vincula un conjunto de *entidades* (previamente definidas en términos de *modelos de ruteo*).

MODELO ESENCIAL (COMPONENTE)	ENTIDADES (MODELO DE RUTEO)
$MACHINE_{TYPE\_A}$	$RM1\_MAC_{TYPE\_A}$
	$RM2\_MAC_{TYPE\_A}$
$MACHINE_{TYPE\_B}$	$RM1\_MAC_{TYPE\_B}$
	$RM2\_MAC_{TYPE\_B}$
	$RM3\_MAC_{TYPE\_B}$

Tabla 9.1. Entidades de ruteo asociadas a componentes básicos.

Tal como se ilustra en el ejemplo precedente, en ambos casos se utilizan las mismas máquinas (*componentes*) como base de los nodos (*entidades*) que componen el escenario (*proceso*). Esto permite abstraer al diseñador de los detalles de ruteo, brindándole la posibilidad de enfocar su atención en el diseño de las máquinas relevantes para el objetivo de simulación. Además, facilita la ejecución de múltiples escenarios asociados al mismo conjunto de máquinas (facilitando la experimentación de diferentes situaciones sin alterar el comportamiento de los modelos básicos desarrollados).

Luego, los lineamientos propuestos como base del enfoque de los modelos RDEVS proponen integrar la definición de los componentes en la especificación de las entidades a fin de facilitar la tarea de modelado permitiendo:

- Modelar solo el comportamiento de los elementos del dominio como elementos componentes básicos (sin gestionar implementaciones para el proceso de ruteo).
- Definir diferentes escenarios utilizando un mismo conjunto de componentes (único) sin cambiar la especificación de los modelos que representan los distintos elementos del dominio.

9.3.2 Formalización de Modelos: RDEVS Esenciales, RDEVS de Ruteo y RDEVS de Red [1]

Modelo Esencial (RDEVS Essential Model)

Este modelo especifica la descripción de comportamiento de un modelo de simulación interno que debe ser utilizado como parte componente del proceso de ruteo. Cada modelo esencial debe ser diseñado como un elemento básico que, eventualmente, puede ser replicado en múltiples entidades de un mismo proceso de ruteo.

Un modelo esencial RDEVS equivale a un modelo atómico DEVS. Formalmente, queda definido por la estructura

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau > \qquad (9.3)$$

donde:

$X \equiv$ conjunto de eventos de entrada,

$Y \equiv$ conjunto de eventos de salida,

[1] Estos modelos han sido propuestos en (Blas, Gonnet y Leone 2017a).

S ≡ conjunto de estados secuenciales,

$\delta_{int}: S \rightarrow S$ ≡ función de transición interna,

$\delta_{ext}: Q \times X \rightarrow S$≡ función de transición externa, en la cual:

$Q = \{ (s,e) \mid s \in S, 0 \leq e \leq \tau(s) \}$ ≡ conjunto de estados totales,

e ≡ tiempo transcurrido desde la última transición,

$\lambda: S \rightarrow Y \cup \emptyset$ ≡ función de salida,

$\tau: S \rightarrow \mathbb{R}^{+}_{0,\infty}$ ≡ función de avance de tiempo.

Modelo de Ruteo (RDEVS Routing Model)

Este modelo define una entidad de simulación básica en la cual tiene lugar el proceso de ruteo. Para esto, un modelo de ruteo utiliza la especificación de un *componente* (esto es, un modelo esencial) como base de comportamiento para su descripción operacional (a fin de especificar su propio comportamiento). Luego, la estructura de un modelo de ruteo encapsula en su definición la estructura de un modelo esencial (Ecuación 9.3) junto con los elementos requeridos para describir la información de ruteo que será utilizada para aceptar/rechazar eventos de entrada y dirigir eventos de salida.

En este sentido, la especificación del modelo de ruteo detalla una entidad que vincula la descripción de un modelo esencial con los datos necesarios para tratar los eventos de entrada/salida. Entonces, se pueden definir múltiples modelos de ruteo basados en un mismo modelo esencial con diferente información de ruteo y, viceversa, la misma información de ruteo puede ser aplicada a múltiples modelos de ruteo basados en modelos esenciales con distinto comportamiento. Esta flexibilidad permite crear diferentes configuraciones con el mismo conjunto de componentes básicos.

Formalmente, un modelo de ruteo RDEVS queda definido por la estructura

$$R = < \omega, E, M > \quad (9.4)$$

donde:

$\omega = (u, W, \delta_r)$ ≡ información de ruteo, en la cual:

$u \in N_0$ ≡ identificador del modelo,

$W = \{ w_1, w_2,..., w_p \mid w_1, w_2,..., w_p \in N_0 \}$ ≡ conjunto de identificadores de modelos de ruteo que representan el conjunto de entidades remitentes habilitadas (esto es, modelos de ruteo desde los cuales tiene permitido recibir eventos de entrada),

$\delta_r: S_M \rightarrow T$ ≡ función de ruteo utilizada para dirigir eventos de salida, en la cual:

S_M ≡ estado del modelo M,

$T = \{ t_1, t_2,..., t_k \mid t_1, t_2,..., t_k \in N_0 \}$ ≡ conjunto de identificadores de modelos de ruteo que representan el conjunto de posibles entidades destinatarias,

$E = < X_E, S_E, Y_E, \delta_{int,E}, \delta_{ext,E}, \lambda_E, \tau_E >$ ≡ modelo esencial embebido en R,

$M = <X_M, S_M, Y_M, \delta_{int,M}, \delta_{ext,M}, \lambda_M, \tau_M>$ ≡ modelo DEVS atómico que especifica el comportamiento de una entidad de ruteo, en el cual:

$X_M = \{ (x, h, T) \mid x \in X_E, h \in N_0, T = \{ t_1, t_2,..., t_k \mid t_1, t_2,..., t_k \in N_0 \} \}$ ≡ conjunto de eventos de entrada con identificación, en el cual:

x ≡ evento de entrada definido en E,

h ≡ identificador del modelo remitente,

T ≡ conjunto de identificadores de los modelos destinatarios,

$S_M = S_E \equiv$ conjunto de estados secuenciales,

$Y_M = \{ (y, h, T) \mid y \in Y_E, h \in N_0, T = \{ t_1, t_2,..., t_k \mid t_1, t_2,..., t_k \in N_0 \} \} \equiv$ conjunto de eventos de salida con identificación, en el cual:

$y \equiv$ evento de salida definido en E,

$h \equiv$ identificador del modelo remitente (esto es, el valor de u),

$T \equiv$ conjunto de identificadores de los modelos destinatarios,

$\delta_{int,M}: S_M \rightarrow S_M = \delta_{int,E} \equiv$ función de transición interna,

$\delta_{ext,M}: Q_M \times X_M \rightarrow S_M \equiv$ función de transición externa, en la cual:

$Q_M = \{ (s,e) \mid s \in S_M, 0 \leq e \leq \tau_M(s) \} \equiv$ conjunto de estados totales,

$e \equiv$ tiempo transcurrido desde la última transición,

cuya definición solamente acepta eventos de entrada cuando se satisface una de las siguientes condiciones:

- *Condición #1*: El evento ha sido enviado a R desde un modelo habilitado (es decir, u se encuentra incluido en el conjunto de destinatarios T y el identificador de remitente h se encuentra incluido en W).
- *Condición #2*: El evento proviene de una fuente externa (esto ocurre cuando $h=0$ y u se encuentra incluido en el conjunto de destinatarios T).
- *Condición #3*: El modelo se encuentra forzado a aceptar todos los eventos de entrada, por lo que su información de ruteo es $u = 0 \wedge W = \emptyset$.

Luego, esta función queda definida como:

$$\delta_{ext,M}(s,e,x) = \begin{cases} \delta_{ext,E}(s,e\_c+e,x) & \text{si se cumple ①} \\ s & \text{en otro caso ②} \end{cases}$$

donde:

① condición $(u \in T \wedge h \in W) \vee (h = 0 \wedge u \in T) \vee (u = 0 \wedge W = \emptyset)$ con $x' = (x, h, T)$ y seteando $e\_c = 0$ luego de la ejecución,

② actualizando el tiempo transcurrido acumulado según $e\_c = e\_c + e$,

$\lambda_M: S_M \rightarrow Y_M \cup \emptyset \equiv$ función de salida que genera eventos identificados, la cual queda definida como:

$$\lambda_M(s) = (\lambda_E(s), u, \delta_r(s))$$

$\tau_M: S_M \rightarrow \mathbb{R}^+_{0,\infty} \equiv$ función de avance de tiempo.

A partir de esta definición, se observa que los eventos quedan identificados en términos de su remitente (es decir, del modelo que genera el evento como evento de salida). Aunque se utiliza un número natural para definir los identificadores de los modelos, este dato puede ser reemplazado por algún otro tipo de información que ayude a individualizar los modelos como parte del proceso de ruteo.

La especificación de las funciones $\delta_{ext,M}$, $\delta_{int,M}$ y λ_M muestra la forma en la cual se encapsulan los comportamientos del modelo esencial como parte del modelo de ruteo.

La función de transición externa ha sido diseñada como una función *if/else* que genera un cambio de estado únicamente cuando se acepta un evento de entrada (caso ①). Esta aceptación implica que el evento debe ser procesado en el modelo ya que, en principio, ha sido enviado hacia este destino para su tratamiento. Es decir, si un evento de entrada tiene la identificación requerida, el modelo de ruteo lo acepta y ejecuta una transición de estado equivalente a la definida en la función de transición externa detallada como parte del modelo esencial incrustado en él. De lo contrario, el modelo

de ruteo rechaza el evento y se mantiene en el estado que se encontraba antes de que éste arribe (actualizando el tiempo transcurrido desde la última transición ya que deben acumularse los instantes transcurridos en relación al modelo E).

La ejecución de las transiciones internas se rige por el modelo esencial ya que el comportamiento del modelo de ruteo debe ser el mismo que el descripto para el modelo esencial incluido en él. Es decir, en el modelo de ruteo solamente tienen lugar las transiciones internas que provienen del modelo esencial asociado. Por este motivo, la función de transición interna de M equivale a la función de transición interna definida en E. Sin embargo, en este punto es importante tener en cuenta que previo a realizar una transición interna debe enviarse un evento de salida.

Luego, considerando que la función de transición interna de M es igual a la definida en el modelo E, las salidas de ambos modelos deben producirse en el mismo instante de tiempo. Sin embargo, las salidas de M deben corresponder a eventos identificados (ya que pertenecen a una entidad de ruteo). Por este motivo, la función de salida de M utiliza la función de salida de E como parte del evento a ser transferido y, a su vez, incorpora la información de ruteo asociada al estado actual. Esta información queda definida utilizando el identificador del remitente (es decir, el identificador del modelo de ruteo) y la función de enrutamiento (que brinda el conjunto de destinatarios del evento).

Modelo de Red (RDEVS Network Model)

Este modelo describe una estructura de simulación que debe ser diseñada con un objetivo específico de interacción entre componentes. Esta interacción requiere la definición de un proceso que envíe y reciba eventos identificados (es decir, un proceso de ruteo).

Para construir este proceso se requieren múltiples entidades de ruteo. En este sentido, la definición de un modelo de red incluye un conjunto de modelos de ruteo

junto con los acoplamientos requeridos para identificar sus influencias. Estos acoplamientos quedan detallados como conexiones *all-to-all* con el objetivo de independizar la tarea de ruteo de los vínculos entre componentes. Además, un modelo de red incorpora en su definición dos nuevas funciones de traducción (especiales) que son utilizadas para conectar múltiples modelos de red que han sido diseñados como procesos de ruteo individuales. Estas funciones permiten equiparar eventos de diferentes tipos de modelos con eventos identificados. Luego, un modelo de red RDEVS se encuentra diseñado para interactuar con otros modelos del mismo tipo o, simplemente, con modelos DEVS. La combinación de diferentes tipos de modelos varía según el propósito de la simulación.

Formalmente, un modelo de red RDEVS queda definido por la estructura

$$N = < X, Y, D, \{ R_d \}, \{ I_d \}, \{ Z_{i,d} \}, T_{in}, T_{out}, Select > \quad (9.5)$$

donde:

$X \equiv$ conjunto de eventos de entrada,

$Y \equiv$ conjunto de eventos de salida,

$D \equiv$ conjunto de identificadores que representan las entidades a ser utilizadas como parte del proceso de ruteo (es decir, referencias a modelos de ruteo), en el cual $d \in N_0, \ \forall d \in D$,

Para cada $d \in D$, R_d es un modelo de ruteo definido como:

$$R_d =< \omega_d, E_d, M_d >$$

con $u_d = d$,

Para cada $d \in D \cup \{ N \}$, I_d es el conjunto de influyentes de d, el cual queda definido como:

$$I_d = \{ i \mid i \in D \wedge i \neq d \} \cup \{ N \}$$

con el objetivo de mantener todos los acoplamientos requeridos,

Para cada $i \in I_d$, $Z_{i,d}$ es la función de traducción entre eventos de salida del modelo i y eventos de entrada del modelo d, siendo:

$Z_{i,d} = T_{in}$ si $i = N$,

$Z_{i,d} = T_{out}$ si $d = N$,

$Z_{i,d}: Y_{M,i} \rightarrow X_{M,d}$ si $i \neq N \wedge d \neq N$,

$T_{in}: X \rightarrow \{ (x, h, T) \mid x \in X, h \in N_0, T = \{ t_1, t_2,..., t_k \mid t_1, t_2,..., t_k \in N_0 \} \}$ ≡ función de traducción de entrada que toma un evento de entrada (que proviene de un modelo externo a la red) y devuelve un evento de entrada identificado en el contexto del proceso de ruteo, en la cual:

x ≡ evento de entrada,

$h = 0$ ≡ identificador del modelo remitente (el valor 0 indica que el evento proviene de una fuente externa),

T ≡ conjunto de identificadores de destinatarios del evento de entrada,

$T_{out}: \{ (y, h, T) \mid y \in Y, h \in N_0, T = \{ (t_1, t_2,..., t_k) \mid t_1, t_2,..., t_k \in N_0 \} \} \rightarrow Y$ ≡ función de traducción de salida que toma un evento de salida identificado en el contexto del proceso de ruteo y devuelve un evento de salida que puede ser enviado hacia el exterior de la red, en la cual:

y ≡ evento de salida,

h ≡ identificador del modelo remitente,

$T = \emptyset$ ≡ conjunto de identificadores de destinatarios (el conjunto vacío indica que el evento es enviado hacia un destino externo),

Select: $2^D \rightarrow$ D ≡ función de desempate entre transiciones simultaneas.

Según la definición formal, el modelo de red solo comprende la estructura que brinda soporte al proceso de ruteo diseñado en su interior (haciendo uso de modelos de ruteo). Por este motivo, tanto los eventos que provienen de modelos externos (es decir, orígenes ajenos al proceso de ruteo modelado) como los eventos que son enviados hacia modelos externos (es decir, destinos ajenos al proceso de ruteo modelado) no son conceptualizados como eventos con identificación. En estos casos, todo evento que ingresa/egresa hacia un modelo de red es simplemente un evento DEVS.

Luego, todos los eventos de entrada externos que arriban a un modelo de red, son traducidos por medio de la aplicación de la función de traducción de entrada con el objetivo de convertirse en eventos identificados dentro del proceso de ruteo. Dado que no existe un remitente válido para dichos eventos, la función de traducción establece el valor de *h* en cero. Cualquier modelo de ruteo que reciba un evento con *h = 0* debe evaluar la pertenencia de su identificador al conjunto *T* con el objetivo de determinar si acepta o rechaza el evento (esto es, la *Condición #2* de la función de transición externa).

Para los eventos de salida se aplica una estrategia inversa. Dado que no existen destinatarios válidos dentro del modelo para dichos eventos (ya que el proceso de ruteo solo tiene lugar dentro del modelo de red) y ante el desconocimiento de la información asociada a posibles procesos de ruteo modelados por fuera de este ámbito, los destinos externos se fijan haciendo uso del conjunto vacío (*T* = ø). Teniendo en cuenta que los eventos de salida que abandonan la red deben corresponder a eventos DEVS (a fin de mantener el proceso de ruteo aislado de otros modelos), la función de traducción de salida reduce los eventos de salida identificados a eventos de salida regulares DEVS (por medio de la remoción de la información de ruteo).

9.3.3 Clausura Bajo Acoplamiento (Closure Under Coupling)

Un formalismo basado en sistemas es cerrado bajo acoplamiento si el resultado de cualquier configuración de red de sistemas especificada en el formalismo es, en sí misma, un sistema en el formalismo (Zeigler, Praehofer y Kim 2000).

La clausura bajo acoplamiento garantiza la composición jerárquica de modelos en un formalismo dado. Es decir, permite construir modelos de forma recursiva con cualquier nivel arbitrario de jerarquía en base a una estrategia modular. Además de garantizar la construcción jerárquica, el cumplimiento de esta propiedad proporciona seguridad de que el formalismo bajo consideración se encuentra bien definido, permitiendo verificar el correcto funcionamiento de la retroalimentación de los modelos acoplados (Zeigler 2018).

Una forma de demostrar que el formalismo de RDEVS es cerrado bajo acoplamiento consiste en la obtención de dos modelos RDEVS equivalentes, a saber:

i) el modelo de ruteo RDEVS que equivale al modelo de red RDEVS, y
ii) el modelo esencial RDEVS que equivale al modelo de ruteo RDEVS.

Ambos modelos (en conjunto) pueden ser utilizados para justificar la composición jerárquica, ya que los modelos propuestos en i) y ii) aseguran que los modelos de red y de ruteo pueden ser utilizados para estructurar jerárquicamente los modelos de ruteo y esenciales, respectivamente. Luego, por transitividad, un sistema que ha sido especificado utilizando un modelo de red RDEVS puede reducirse a un modelo de ruteo RDEVS de comportamiento equivalente y, a su vez, este último puede volver reducirse en un modelo esencial RDEVS equivalente. De esta manera, el modelo esencial resultante (que corresponde a un modelo equivalente del modelo de red original) puede utilizarse como componente de uno o más modelos de ruteo los cuales, a su vez, den lugar a nuevas entidades con el objetivo de especificar un nuevo modelo de red.

Las siguientes secciones detallan las demostraciones de i) y ii). Si bien los modelos equivalentes no son comúnmente utilizados en las demostraciones de clausura bajo acoplamiento, su aplicación en el formalismo RDEVS tiene como objetivo mostrar la forma en la cual la construcción jerárquica permanece dentro del formalismo.

(CLAUSURA BAJO ACOPLAMIENTO) *Un formalismo basado en sistemas es cerrado bajo acoplamiento si el resultado de cualquier configuración de red de sistemas especificada en el formalismo es, en sí misma, un sistema en el formalismo (Zeigler, Praehofer y Kim 2000).*

Demostración i): Modelo de Red a Modelo de Ruteo

El modelo de red

$$N = < X, Y, D, \{ R_d \}, \{ I_d \}, \{ Z_{i,d} \}, T_{in}, T_{out}, Select > \tag{9.6}$$

en el cual, cada $d \in D$ refiere a un modelo de ruteo R_d estructurado como

$$R_d = < \omega_d, E_d, M_d > \tag{9.7}$$

define un modelo de ruteo equivalente

$$R = < \omega, E, M> \tag{9.8}$$

en el cual

$\omega = (u, W, \delta_r) = (0, \emptyset, \delta_r) \mid \delta_r: S_M \rightarrow T \wedge T = \emptyset \equiv$ información de ruteo a ser utilizada en R. El modelo equivalente utiliza el valor cero como identificador y el conjunto vacío para representar los modelos de ruteo que constituyen entidades remitentes habilitadas. Esta configuración da lugar a la *Condición #3* de la función de transición externa, por lo que todos los eventos de entrada que arriben a *R* serán atendidos. Además, al indicar el resultado de la función de ruteo como el conjunto vacío, todos los eventos de salida que se produzcan en *R* serán eventos

de salida externos (es decir, no tendrán un destino explícito). Luego, el manejo de los eventos entrantes y salientes del modelo *R* se corresponde (de forma equivalente) con el proceso definido para los eventos de entrada y salida del modelo *N*.

$E = < X_E, S_E, Y_E, \delta_{int,E}, \delta_{ext,E}, \lambda_E, \tau_E >$ ≡ modelo esencial embebido en *R*, en el cual:

$X_E = X$ ≡ conjunto de eventos de entrada de *E*. Dado que *R* es un modelo equivalente de N y considerando que (por definición) un modelo de ruteo utiliza las entradas de *E* como parte de la especificación de sus propias entradas, el conjunto de entradas de *E* se corresponde con el conjunto de entradas definidas para *N*.

$S_E = \times_{i \in D} Q_i \mid Q_i = \{ (s_i, e_i) \mid s_i \in S_{M,i}, 0 \leq e_i \leq \tau_{M,i}(s_i), \forall i \in D$ ≡ conjunto de estados secuenciales de *E*, el cual queda definido como el producto de los conjuntos Q_i de cada modelo de ruteo que compone el modelo de red *N*. En este contexto, cada Q_i es definido como un par ordenado que contiene el estado y el tiempo transcurrido desde la última transición del modelo R_i.

$Y_E = Y$ ≡ conjunto de eventos de salida de *E*. Al igual que en el caso de los eventos de entrada, los eventos de salida se definen de acuerdo a los eventos detallados en *N*.

$\delta_{int,E}(s) = s'$ ≡ función de transición interna de *E* que modifica el estado del modelo partiendo de $s = (...,(s_j,e_j),...)$ y generando una transición hacia el estado $s' = (...,(s'_j,e'_j),...)$ donde $\{ s, s' \} \in S_E$. Teniendo en cuenta que el estado de *E* es definido como una combinación de los estados de los modelos de ruteo incluidos en *N*, una transición interna de *E* puede involucrar transiciones internas simultáneas de múltiples componentes. Considerando que los componentes inminentes (estos son, los componentes candidatos para ajustar su estado como consecuencia de una transición) son

recolectados de acuerdo al valor de tiempo σ en un conjunto definido como $IMM(s) = \{ i \in D \mid \sigma_i = \tau_{E,i}(s) \}$, un modelo i^* de este conjunto debe ser seleccionado para ejecutar su transición interna. Para esto, puede utilizarse la función de desempate. Luego, la transición interna inminente a ser ejecutada pertenece al modelo R_{i*} donde $i^* = Select(IMM(s))$. Sin embargo, la ejecución de esta transición conlleva la ejecución de todas las transiciones externas de los componentes influenciados por R_{i*}. Por lo tanto, la trasformación del estado $s = (...,(s_j,e_j),...)$ a $s' = (...,(s'_j,e'_j),...)$ queda especificada como

$$s_j = \begin{cases} \delta_{int,M,j}(s_j) & si\ j=i^* \\ \delta_{ext,M,j}(s_j,e_j+\tau_E(s),x_j) & si\ i^* \in I_j \wedge x_j \neq \emptyset\ con\ x_j=Z_{i^*,j}(\lambda_{M,i^*}(s_{i^*})) \\ s_j & en\ otro\ caso \end{cases}$$

$$e_j = \begin{cases} 0 & si\ \left(j=i^* \right) \vee (i^* \in I_j \wedge x_j \neq \emptyset) \\ e_j + \tau_E(s) & en\ otro\ caso. \end{cases}$$

$\delta_{ext,E}(s, e, x) = s' \equiv$ función de transición externa de E que modifica el conjunto de pares de estados que referencian a los modelos R_i que se encuentran vinculados a las entradas del modelo N. Dado que el estado de E es definido como S_E, los estados $\{ s, s' \} \in S_E$ con $s = (...,(s_i,e_i),...)$ y $s' = (...,(s'_i,e'_i),...)$. Considerando que los componentes afectados por la transición se coleccionan en el conjunto C definido como $C = \{ i \in D \mid N \in I_i \wedge x_i \neq \emptyset \}$, se tiene que

$$s_i^* = \delta_{ext,M,i}(s_i,e_i+e,x_i) \qquad con\ x_i = Z_{N,i}(x),\ \forall i \in C$$

por lo que la transformación del estado queda definida como

$$(s_i,e_i) = \begin{cases} (s_i^*,0) & si\ (N \in I_i \wedge x_i \neq \emptyset) \\ (s_d,\ e_d+e) & en\ otro\ caso. \end{cases}$$

$\lambda_E(s)$: $S_E \rightarrow Y_E \cup \emptyset$ ≡ función de salida de *E* que genera un evento de salida si y solo si el modelo que ejecuta la transición interna (esto es, el modelo i^*) se encuentra vinculado a las salidas de *N*. Luego, su definición queda formalizada como

$$\lambda_E(s) = \begin{cases} Z_{i^*,N}\left(\lambda_{M,i^*}(s_{i^*})\right) & si\ N \in I_{i^*} \\ \emptyset & en\ otro\ caso. \end{cases}$$

τ_E: $S_E \rightarrow \mathbb{R}^+_{0,\infty}$ ≡ función de avance de tiempo de *E* que selecciona el tiempo del evento más inminente de todos los componentes de *N*. Esto implica encontrar el menor tiempo remanente (hasta la próxima transición interna) de todos los modelos de ruteo incluidos en *N*, ya que este tiempo corresponde a la cantidad de instantes de tiempo que debe transcurrir hasta que se ejecute la siguiente transición. Dicho tiempo queda simbolizado con σ. Luego, la función se define como $\tau_E(s) = min\ \{ \sigma_i = \sigma_{M,i}(s_i) - e_i \mid i \in D \}$.

$M = < X_M, S_M, Y_M, \delta_{int,M}, \delta_{ext,M}, \lambda_M, \tau_M >$ ≡ modelo DEVS atómico que especifica el proceso de ruteo sobre *R*. De acuerdo a los lineamientos establecidos en la Ecuación 9.4, la formalización de *M* se define en base a la especificación de *E*. Luego, no deben incorporarse nuevas consideraciones al construir una equivalencia con N (ya que podría violarse la propia definición de modelo de ruteo). Dado que el modelo *E* asociado a *R* ya ha sido definido, la descripción de M debe seguir la formulación especificada en el tipo de modelo asociado.

Luego, se ha construido un modelo de ruteo R cuyo comportamiento equivale al modelo de red N.

Demostración ii): Modelo de Ruteo a Modelo Esencial

El modelo de ruteo

$$R = < \omega_R, E_R, M_R > \qquad (9.9)$$

en el cual

$$M_R = < X_{M,R,} S_{M,R}, Y_{M,R}, \delta_{int,M,R}, \delta_{ext,M,R}, \lambda_{M,R}, \tau_{M,R} > \quad (9.10)$$

define un modelo esencial equivalente

$$M = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau> \quad (9.11)$$

en el cual $X = X_{M,R}$, $S = S_{M,R}$, $Y = Y_{M,R}$, δ_{int} se deriva de la definición de $\delta_{int,M,R}$, δ_{ext} se deriva de la definición de $\delta_{ext.M,R}$, $\lambda = \lambda_{M,R}$ y $\tau = \tau_{M,R}$.

De acuerdo con esta prueba de equivalencia, cada elemento incluido en la definición del modelo M_R (que compone un modelo de ruteo R) puede ser asignado directamente a un elemento componente de un modelo esencial M. Estas asociaciones se fundamentan en la propia definición de modelo de ruteo.

La especificación de un modelo de ruteo incluye dos modelos atómicos DEVS definidos como E y M. Mientras que E determina el componente que se utiliza como parte de la entidad de ruteo (es decir, el modelo esencial que describe el comportamiento del modelo de ruteo), M define el modelo de simulación ejecutable sobre el cual se lleva a cabo el proceso de direccionamiento. Dado que la prueba de equivalencia intenta encontrar un componente que exhiba el mismo comportamiento que una entidad de ruteo y, considerando que tanto M como M_R se corresponden con modelos atómicos DEVS, se puede mapear directamente la descripción de la entidad de ruteo como parte de la especificación del componente. Toda la información necesaria para ejecutar el proceso de ruteo queda implícitamente incluida en M ya que la especificación de M_R utiliza el contenido de los elementos ω_R y E_R.

Siguiendo esta equivalencia, cada componente de M_R (esto es, la descripción del modelo de simulación que actúa como entidad de ruteo) permite definir un nuevo modelo esencial M que reduce el comportamiento de la entidad hacia la formulación de un componente.

9.4 RDEVS como Subclase de DEVS

De acuerdo con (Zeigler, Praehofer y Kim 2000), el núcleo de DEVS incluye un framework de Modelado y Simulación (M&S) estructurado en base a tres componentes independientes (Figura 9.2), a saber:

- *Modelo (Model)*: Describe la especificación del sistema bajo estudio de acuerdo a su estructura y comportamiento.
- *Simulador (Simulator)*: Refiere al sistema computacional que ejecuta el conjunto de instrucciones detalladas en el modelo.
- *Marco experimental (Experimental Frame)*: Representa las condiciones bajo las cuales se observa el sistema en estudio, construyendo un contexto para la experimentación y validación del modelo.

Teniendo en cuenta que estos componentes deben interactuar a fin de cumplir sus funciones, el framework incluye un conjunto de relaciones que modelan las interacciones válidas entre componentes. En este contexto, la *relación de modelado* (*modeling relation*) indica que un modelo dado constituye una representación válida de un sistema específico dentro de un marco experimental definido. Por su parte, la *relación de simulación* (*simulation relation*) especifica la constitución de una simulación correcta para un modelo dado haciendo uso de un simulador específico.

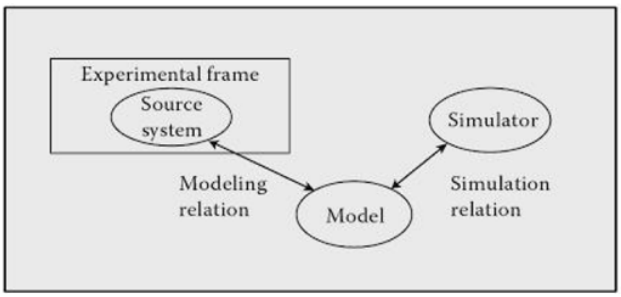


Figura 9.2. Framework de modelado y simulación[2].

[2] Tomado de (Zeigler, Praehofer y Kim 2000).

La cantidad de extensiones y simuladores desarrollados en los últimos años muestra la evolución del formalismo de DEVS como un campo interesante para resolver nuevos tipos de problemas relacionados con sistemas de eventos discretos. Siguiendo el enfoque propuesto en el framework de M&S, se han desarrollado múltiples variantes, extensiones y abstracciones utilizando el núcleo de los conceptos definidos como parte del formalismo original. En este contexto, las extensiones del formalismo DEVS sugieren direcciones en las cuales es posible mejorar: *i)* el modelado de la simulación en general, y *ii)* el formalismo de DEVS en particular.

En este sentido, en (Blas y colab. 2018) se ha presentado una clasificación de las extensiones basada en dos dimensiones fundamentales aplicadas sobre el framework de M&S: *tipos de sistemas* y *tipos de problemas*. En base a estas dimensiones se proponen dos tipos de extensiones básicas: *variantes DEVS* (subconjunto de extensiones de DEVS en las que el formalismo alternativo modela un nuevo tipo de sistema que, anteriormente, no podía modelarse con el formalismo original) y *subclases DEVS* (subconjunto de extensiones de DEVS en las que el formalismo alternativo mejora la solución de un problema de simulación aplicando modelos de DEVS de manera significativa).

En este último caso, a medida que aumenta la complejidad de los problemas se diseñan nuevos mecanismos para enfrentarlos. Este es el caso de RDEVS. Luego, como parte de la propuesta realizada en términos de la clasificación, se ha demostrado que RDEVS corresponde a una subclase DEVS. Para mayores detalles sobre esta comprobación, se sugiere recurrir a (Blas y colab. 2018).

9.5 Beneficios de la Combinación DEVS / RDEVS

Cuando el flujo de eventos es independiente del comportamiento de los componentes, el diseñador puede usar RDEVS para respaldar la tarea de modelado del ruteo de eventos mediante una separación de objetivos adecuada (es términos de la gestión del flujo de eventos y el comportamiento de los componentes). Teniendo en cuenta que los modelos esenciales representan el comportamiento de los componentes básicos y que los modelos de ruteo y de red configuran el contexto en el cual tiene lugar el proceso de direccionamiento, la propuesta detrás de RDEVS permite generar distintos niveles de abstracción para un mismo problema con el objetivo de facilitar la tarea de modelado. Dado que un modelo esencial puede ser utilizado como componente de múltiples modelos de ruteo, una misma especificación de comportamiento puede replicarse en varias instancias con diferentes destinos (es decir, distintas entidades). Si bien este problema puede resolverse utilizando el formalismo DEVS, los modelos finales suelen ser mucho más complejos ya que abarcan la resolución del flujo de eventos de forma conjunta a los objetivos de simulación.

En este contexto, es importante destacar que el uso de RDEVS no restringe su combinación con modelos DEVS. Tal como se ha indicado, el formalismo RDEVS corresponde a una subclase de DEVS cerrada bajo acoplamiento. Esto tiene dos implicancias fundamentales, a saber:

i) *Es posible combinar modelos RDEVS con modelos DEVS para la resolución de un único problema*: Un modelo esencial RDEVS es equivalente a un modelo atómico DEVS. Luego, dado que DEVS es cerrado bajo acoplamiento, un modelo DEVS puede ser utilizado como base para la construcción de los componentes a ser incluidos como parte de las entidades de ruteo. Además, los procesos de ruteo definidos en base a modelos de red RDEVS pueden ser combinados con modelos de simulación DEVS externos a fin de implementar

un proceso de direccionamiento sobre algunos de los componentes requeridos como parte de un modelo de mayores dimensiones.

ii) *Los modelos RDEVS pueden ejecutarse utilizando un algoritmo de simulación DEVS*: Teniendo en cuenta que RDEVS es una adaptación del formalismo de DEVS conceptualmente definida como una subclase DEVS, todo modelo de la subclase (RDEVS) debe pertenecer a la clase (DEVS). Dado que los modelos RDEVS son compatibles con los modelos DEVS, ambos modelos pueden ser ejecutados utilizando simuladores DEVS.

Luego, el uso de RDEVS como complemento de los modelos DEVS posibilita la construcción de modelos de simulación más poderosos que combinan los beneficios y ventajas de ambos formalismos.

9.6 Implementación de Modelos RDEVS en Java

Dado que el formalismo RDEVS se encuentra diseñado como una subclase de DEVS, las implementaciones disponibles del formalismo original pueden ser utilizadas como base para desarrollar implementaciones propias de RDEVS. En este contexto, con el objetivo de proporcionar un entorno de software que ayude al modelado y simulación en este nuevo formalismo, en (Blas, Gonnet y Leone 2017a) se propuso un framework de software basado en Java para RDEVS. Esta herramienta brinda el respaldo necesario para la implementación y simulación de estructuras de ruteo en modelos de sistemas de eventos discretos basado en el formalismo RDEVS.

El framework propuesto utiliza DEVSJAVA[3] como herramienta de software subyacente. Las clases Java implementadas como conceptos propios de RDEVS se vinculan con las clases propuestas en la herramienta original a fin de garantizar su compatibilidad con fines de simulación.

[3] Disponible en *http://acims.asu.edu/software/devsjava/*

La Figura 9.3 esquematiza un subconjunto simplificado de estas clases, poniendo énfasis en el conjunto de elementos y relaciones fundamentales que han sido definidos como parte de la herramienta de software. Las clases resaltadas en amarillo pertenecen al módulo DEVSJAVA.

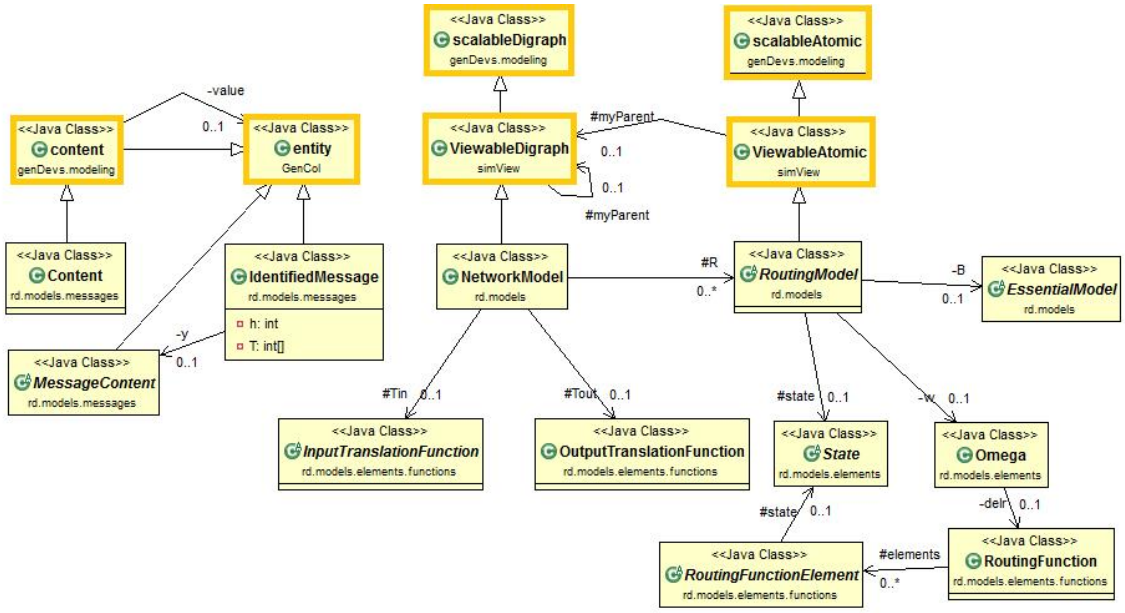


Figura 9.3. Subconjunto de clases definidas como parte del framework RDEVS.

Como puede observarse, cada tipo de modelo incluido en la definición del formalismo RDEVS queda definido haciendo uso de una clase específica, a saber: *i)* el modelo esencial RDEVS es definido en la clase *EssentialModel.java*, *ii)* el modelo de ruteo RDEVS es definido en la clase *RoutingModel.java*, y *iii)* el modelo de red RDEVS es definido en la clase *NetworkModel.java*. Los elementos asociados a la función de ruteo δ_r quedan definidos en las clases *RoutingFunction.java* y *RoutingFunctionElement.java*. Por su parte, la clase *IdentifiedMessaje.java* define la estructura de los eventos con identificación a ser gestionados por el simulador como parte del proceso de ruteo. Las funciones de traducción que componen la definición de un modelo de red corresponden a subclases de *TranslationFunction.java*. Tanto la clase

InputTranslationFunction.java como la clase *OutputTranslationFunction.java* administran eventos de entrada y salida (respectivamente) con el objetivo de ajustar eventos externos en base a los procesos de transformación definidos como parte de los modelos de red RDEVS.

Luego, el diseñador debe instanciar estas clases de acuerdo a las necesidades requeridas como parte de su modelo RDEVS a fin de implementar un modelo de simulación ejecutable en el entorno Java.

Además de la implementación a nivel de código Java, la herramienta de software diseñada tiene la posibilidad de ser complementada con representaciones gráficas de los modelos desarrollados. DEVS Suite[4] es un entorno gráfico que se utiliza para representar modelos de DEVS implementados con DEVSJAVA. Teniendo en cuenta que el proyecto RDEVS extiende un subconjunto de las clases definidas en el proyecto DEVSJAVA, esta herramienta gráfica también puede aplicarse para la representación visual de los modelos de simulación RDEVS.

La Figura 9.4 muestra una captura de pantalla de la implementación del modelo *NM1* (definido de forma conceptual en la Figura 9.1) haciendo uso de la herramienta de software desarrollada en base a una representación DEVS Suite (ventana principal). En el explorador de proyectos (panel izquierdo), se visualizan algunas de las clases Java diseñadas como parte del proyecto RDEVS (las cuales han sido diagramadas en la Figura 9.3).

[4] Disponible en *http://acims.asu.edu/software/devs-suite/*

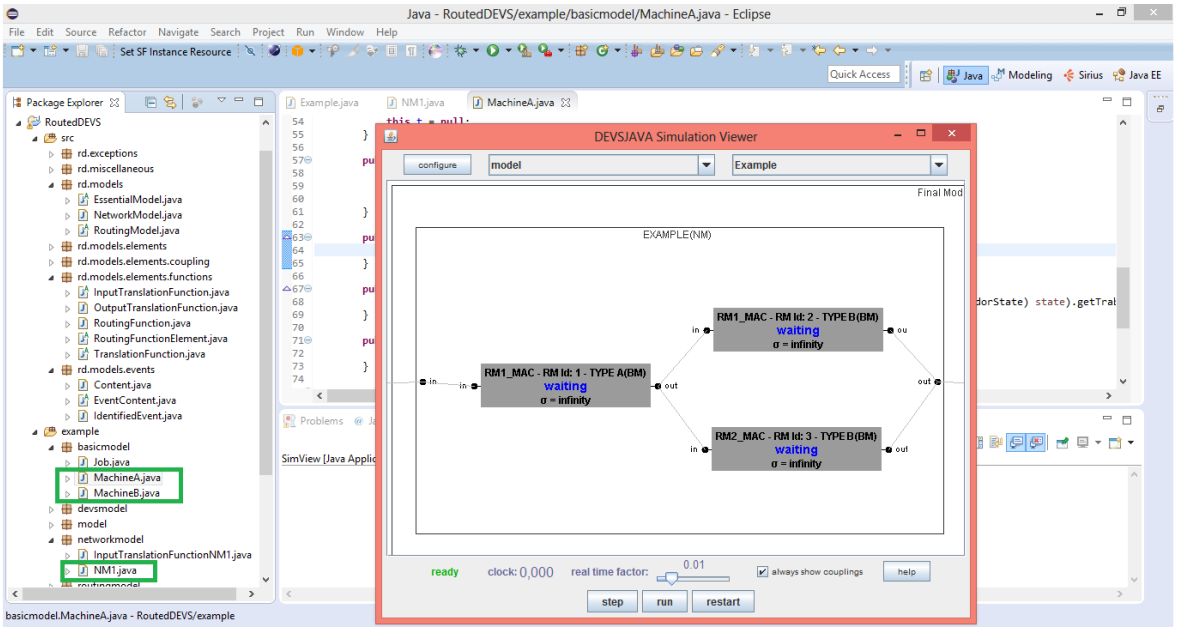


Figura 9.4. Ejemplo de modelo RDEVS implementado en Java.

Las clases resaltadas en verde refieren a los modelos RDEVS implementados con el objetivo de construir el modelo de red *NM1*. Específicamente, las clases *MachineA.java* y *MachineB.java* representan los componentes requeridos como parte de las entidades de ruteo. Ambas clases fueron definidas como instancias de *EssentialModel.java*. A su vez, la clase *NM1.java* fue definida como instancia de *NetworkModel.java.* Esta clase representa el proceso de ruteo final.

El proyecto Java asociado al framework de RDEVS se encuentra disponible en *Framework RDEVS*.

Conclusiones

DEVS es un formalismo modular y jerárquico que permite modelar y analizar sistemas de diversos tipos. La aplicación de este formalismo para la resolución de problemas de ruteo usualmente resulta en estructuras de modelado complejas que desvían la atención del diseñador hacia nuevos desafíos independientes del contexto de trabajo.

En este capítulo se ha presentado una adaptación del formalismo de DEVS denominada RDEVS, la cual ha sido definida como una subclase del formalismo original que permite mejorar el diseño de los modelos de simulación de eventos discretos que abordan problemas de ruteo. Se ha definido formalmente el conjunto de modelos básicos que conforman RDEVS, detallándose además la prueba de clausura bajo acoplamiento del formalismo diseñado. También se ha incluido una breve descripción de la herramienta de modelado y simulación basada en Java que ha sido desarrollada en virtud de dar soporte a los modelos RDEVS.

En este sentido, RDEVS define un conjunto de modelos de simulación que gestionan, de forma independiente al comportamiento de los componentes básicos, un conjunto de eventos identificados utilizando información de ruteo vinculada a entidades que componen un proceso de direccionamiento específico. Cada evento identificado incluye información asociada a su origen y destino/s, por lo que los modelos RDEVS tienen la capacidad de determinar si deben o no aceptar un evento antes de ejecutar sus instrucciones. Aunque se definen nuevos tipos de modelos para representar múltiples niveles de abstracción, el núcleo del formalismo sigue siendo DEVS. Esta dependencia permite ejecutar modelos RDEVS utilizando simuladores DEVS.

La estructura de comportamiento propuesta en el formalismo RDEVS sienta las bases requeridas para el diseño de los modelos de simulación de componentes arquitectónicos de aplicaciones web que se detalla en el capítulo siguiente. Sin embargo, dado que es un

formalismo genérico, también puede ser aplicado a otros contextos (o, incluso, a componentes de un contexto de mayor jerarquía modelado con DEVS) en los que se requiera resolver un problema de ruteo (como, por ejemplo, aplicaciones móviles, protocolos de red y sistemas-de-sistemas).

Capítulo 10. Modelos RDEVS para Componentes de Aplicaciones Web

La definición de modelos de simulación que brinden soporte a los conceptos involucrados en la representación definida para las arquitecturas de software web es uno de los principales objetivos de esta tesis. En el capítulo previo se ha presentado el formalismo RDEVS, el cual ha sido diseñado con el objetivo de gestionar procesos de ruteo de eventos como parte de modelos de simulación discretos. Además, como resultado del conjunto de capítulos desarrollados en las partes II y III, se ha definido un esquema de calidad útil para relevar características propias de las aplicaciones de software web junto con una estrategia de representación basada en componentes definidos según patrones de computación en la nube. En este capítulo se integran todos estos elementos como parte de la definición de modelos de simulación RDEVS aplicables para la evaluación de la calidad de productos de software basados en entornos web. Estos modelos constituyen la base del proceso de evaluación de la calidad planteado en relación a las arquitecturas de software.

10.1 Criterios de Simulación

No todos los atributos de calidad requeridos sobre un producto de software específico pueden ser evaluados a nivel de arquitectura. Esto se debe, fundamentalmente, a que la información necesaria para su estimación no se encuentra

siempre disponible durante la etapa de diseño (lo que conlleva a la necesidad de analizar la factibilidad de la utilización de estos atributos en el contexto del diseño arquitectónico).

Con el objetivo de encontrar un subconjunto de atributos que pueda ser analizado a partir de la arquitectura, pero que además pueden ser evaluado mediante la aplicación de técnicas de simulación, en (Blas, Gonnet y Leone 2014) se ha presentado un conjunto de criterios que permiten (en base a una clasificación dada por un modelo de calidad de producto) elaborar una taxonomía de propiedades de calidad relevantes. Tales criterios brindan un esquema de trabajo que sienta las bases requeridas para la definición de los objetivos de simulación a ser tratados como parte de los modelos a diseñar para las arquitecturas de software web.

El modelo de calidad tomado como punto de partida en dicha propuesta corresponde al descripto en el estándar ISO/IEC 25010 (el cual ha sido presentado en el apartado *"Modelo de Calidad de Producto del Estándar ISO/IEC 25010"*). Sin embargo, a fin de generar una clasificación útil en el contexto de aplicaciones web, en las siguientes subsecciones se presenta la aplicación de los criterios propuestos sobre el modelo de calidad SaaS diseñado conforme lo descripto en el apartado *"Modelo de Calidad para Servicios de Software"*. De esta manera, el conjunto de factores de calidad abstractos propuestos en el modelo original es reutilizado en el contexto de proyectos web por medio de la incorporación de características y subcaracterísticas complementarias (de interés para el producto de software bajo evaluación en relación a los entornos de CC).

Los criterios propuestos han sido definidos de acuerdo a tres lineamientos básicos que deben ser aplicados sobre las propiedades de más alto nivel del modelo de calidad elegido (en el caso del modelo SaaS, estas propiedades refieren a *características de calidad*). Esto se debe a que, restringiendo la calidad en el nivel de información más alto, se restringen todos los niveles inferiores asociados (por lo que no se requiere de

una evaluación exhaustiva de todos los atributos y propiedades vinculadas a cada una de las características de calidad identificadas).

10.1.1 Lineamientos Básicos

La Tabla 10.1 resume el conjunto de lineamientos establecidos en (Blas, Gonnet y Leone 2014) a fin de determinar los aspectos asociados a la calidad del software que son susceptibles de ser evaluados por medio de simulación.

Estos lineamientos no referencian de forma específica a modelos de simulación asociados al diseño de arquitecturas de software web, sino que refieren a un "modelo de simulación de software" genérico. De esta manera, los lineamientos definidos pueden ser aplicados en otro tipo de modelo de simulación de interés para el estudio de la calidad de productos de software.

Nº	ENUNCIADO
1	Sólo es posible simular aspectos de calidad que puedan observarse en tiempo de ejecución.
2	La evaluación del aspecto de calidad debe ser cuantitativa.
3	La información requerida para medir el atributo debe estar disponible.

Tabla 10.1. Lineamientos básicos para la simulación de atributos de calidad.

En relación al primer lineamiento, es importante destacar que las propiedades que pueden ser observadas en tiempo de ejecución permiten establecen una relación entre el estado interno del sistema y la respuesta resultante. En este sentido, tales propiedades son las que evidencian la calidad del sistema en respuesta a las solicitudes realizadas por los usuarios. Si una solicitud es resuelta de forma correcta pero no en el tiempo esperado, aunque se ha cumplido con una funcionalidad, no se han respetado los requerimientos no funcionales asociados. Luego, el objetivo detrás de este lineamiento es analizar cada una de las características de calidad en virtud de determinar si la respuesta del producto de software en tiempo de ejecución condiciona su evaluación. De ser así, la característica se considera *dinámica*.

El segundo lineamiento apunta a la naturaleza numérica de las técnicas de simulación. Si la medición de la característica bajo análisis es subjetiva a la apreciación de los grupos de interés, no es posible estimar su valor por medio de modelos de simulación. Aunque eventualmente podrían establecerse rangos aplicables a distintos niveles cualitativos, este no es el objetivo de la propuesta (ya que los mismos niveles podrían ser definidos en base a otro tipo de modelos que brinden un soporte propio -es decir, sin requerir un proceso de adecuación- para la resolución de estos casos).

Finalmente, el último lineamiento refiere a la disponibilidad de la información requerida para la efectiva medición de la característica bajo análisis. En este sentido, el lineamiento apunta a las herramientas de información que brindan el soporte requerido para la construcción de un modelo de simulación que posibilite la implementación del comportamiento asociado al producto de software a fin de garantizar la obtención del conjunto de datos necesarios para realizar la medición. Bajo esta perspectiva, al momento de diseñar el modelo de simulación, toda la información asociada a la formulación del comportamiento que será evaluado debe estar disponible. En este punto es importante destacar que no es necesario que toda la información se encuentre explícitamente declarada, sino que la misma puede obtenerse de resultados de simulaciones previas y/o de otro tipo de estimaciones. En caso de que la información requerida no se encuentre definida, no se recomienda aplicar un proceso de simulación (ya que los resultados que se obtengan no serán fiables, pues el modelo no ha representado de forma completa el comportamiento del producto de software asociado).

10.1.2 Taxonomía de Características de Calidad Simulables

Tomando como punto de partida el modelo SaaS (esquematizado en la *Figura 6.1*), se propone a continuación una calificación de sus características en función de los criterios detallados en la sección precedente. En este punto, es importante destacar que

el objetivo es utilizar modelos de simulación de arquitecturas de software, por lo que se restringe la aplicación de los criterios a este dominio de análisis.

Debido a que sólo es posible simular aspectos que puedan ser observados en tiempo de ejecución (lineamiento 1), la Tabla 10.2 presenta la clasificación del conjunto de características de calidad SaaS según su correspondencia a propiedades *estáticas* o *dinámicas*.

DINÁMICA	**ESTÁTICA**
Functional Suitability	*Maintainability*
Performance Efficiency	*Portability*
Compatibility	
Usability	
Reliability	
Security	

Tabla 10.2. Evolución temporal de las características del modelo de calidad SaaS.

Como puede observarse, las características “Mantenibilidad” (*maintainability*) y “Portabilidad” (*portability*) son las únicas cuya evolución temporal corresponde a un esquema estático. Esto se debe a que la medición de estas propiedades no guarda relación con estados internos que puedan alterar los resultados obtenidos. Por este motivo, no es necesario ejecutar el sistema de software para evidenciar su evolución. En contraposición, resulta evidente que la característica “Funcionalidad” (*functional suitability*) de un sistema debe ser evaluada en tiempo de ejecución. Lo mismo ocurre con las características “Confiabilidad” (*reliability*), “Compatibilidad” (*compatibility*), “Seguridad” (*security*), “Usabilidad” (*usability*) y “Rendimiento” (*performance efficiency*). En todos estos casos la evaluación de las propiedades requiere de una valoración del comportamiento del sistema ante estímulos específicos. Al encontrarse una dependencia entre la propiedad y la respuesta del sistema, las características se convierten en dinámicas.

Sobre los aspectos dinámicos identificados, se aplicaron los lineamientos restantes. En este caso, los criterios se centran en base al uso del diseño de la arquitectura de software como constructor base de los modelos de simulación. Esto implica que la información requerida para estimar la característica debe existir (o poder obtenerse) durante la etapa de diseño y, además, que su valoración debe ser cuantitativa.

En este sentido, las características “Funcionalidad”, “Compatibilidad”, “Usabilidad” y “Seguridad” no pertenecen al conjunto de características que pueden ser estimadas por medio de simulación arquitectónica. La información necesaria para evaluar la característica “Funcionalidad” (de acuerdo al conjunto de subcaracterísticas vinculadas, las cuales han sido expuestas en el apartado *"Funcionalidad (Functional Suitability)"*) no se encuentra completamente disponible como parte de la descripción arquitectónica (por lo que viola el tercer lineamiento), requiriendo en algunos casos de una evaluación cualitativa del usuario (por lo que no cumple el segundo criterio). Por su parte, la medición de la característica “Compatibilidad” requiere simular el comportamiento de los sistemas de software vinculados al producto bajo estudio. Esto constituye, en la mayoría de los casos, información desconocida a nivel de diseño (por lo que no cumple el tercer criterio). Aun cuando se tuviese la certeza de que el comportamiento de los otros sistemas se encuentra correctamente definido como parte de un modelo de simulación externo, la fiabilidad de los resultados obtenidos puede ser cuestionada. En el caso de la característica “Usabilidad”, su evaluación se asocia con comprobaciones subjetivas del comportamiento del software por parte del usuario final. En este sentido, suele ser difícil reducir su medición a valores numéricos (motivo por el cual no cumple el segundo lineamiento). Finalmente, la característica “Seguridad” requiere de una definición pormenorizada de los datos y funciones del producto de software que deben ser protegidos de accesos malintencionados. Dado que, usualmente, en la etapa de diseño arquitectónico este tipo de información aún no ha sido ni especificado ni

modelado formalmente, no es posible analizar esta característica por medio de simulación (violación del segundo criterio).

En este contexto, la Tabla 10.3 presenta la taxonomía final de características de calidad SaaS que pueden estimarse mediante la simulación del diseño arquitectónico (propiedades detalladas en la columna *simulable*). Tanto el "Rendimiento" como la "Confiabilidad" constituyen características simulables ya que la información necesaria para su derivación existe (o puede estimarse) en el momento en el cual se desarrolla el diseño de la arquitectura y sus mediciones corresponden a evaluaciones cuantitativas.

SIMULABLE	NO SIMULABLE
Performance Efficiency	*Functional Suitability*
Reliability	*Compatibility*
	Usability
	Security

Tabla 10.3. Características de calidad simulables durante la etapa de diseño.

10.2 Identificación de Objetivos de Simulación

Como resultado de la aplicación de los criterios de simulación previos, se restringe el conjunto de características generales a aquellos casos en los que es posible realizar una evaluación de la calidad por medio del uso de técnicas de simulación sobre el diseño de las arquitecturas de software web. Este subconjunto de características puede combinarse con el análisis de información requerida (propuesto en el apartado *"Análisis Información Requerida vs. Información Disponible"*) a fin de determinar el conjunto de datos a ser relevados como resultado del proceso de simulación. Es decir, en base a las métricas propuestas como parte del esquema de calidad SaaS (detallado en *"Definición de un Esquema de Calidad para Servicios Web"*), es posible determinar la información de salida básica asociada a los modelos de simulación a fin de especificar su posterior comportamiento en relación a estas directrices.

10.2.1 Análisis de la Información asociada al Esquema de Calidad SaaS

Tomando como base la instancia *QSO** diseñada como esquema de calidad para servicios web, se aplican las *reglas SWRL de la ontología QSO para derivar nuevo conocimiento* en virtud de obtener la información mínima requerida. Además, con el objetivo de enfocar su aplicación en los datos a ser relevados como parte del proceso de simulación de las arquitecturas de software web, se restringe el alcance de esta información a las características identificadas en la Tabla 10.3.

La Figura 10.1 presenta una captura de pantalla del resultado obtenido luego de la ejecución del *análisis de información requerida* en la herramienta Protégé (en la cual se ha generado la instancia QSO* de la ontología QSO sobre la cual se ha aplicado el proceso de razonamiento propuesto en base a las reglas SWRL). Los elementos asociados a la clase “RequiredData” quedan resaltados en la parte inferior-derecha de la pantalla. Como puede observarse, estos elementos han sido inferidos (etiqueta *inferred*) como parte del proceso de análisis. Es importante destacar que cada una de las abreviaturas utilizadas para identificar los elementos corresponde a términos utilizados en las métricas propuestas como parte del esquema de calidad. Tales términos han sido definidos en el apartado *"Definición de Métricas de Software"*. Por ejemplo, el elemento *ET* representa el término *execution time* de la métrica *comportamiento temporal percibido por el usuario*.

En este contexto, se ha identificado 14 elementos de información requeridos para analizar el 100% de las métricas asociadas a las características simulables del esquema de calidad SaaS. Estos elementos sientan las bases para el estudio de las propiedades “Rendimiento” y “Confiabilidad” desde el punto de vista del esquema de calidad propuesto en base a un modelo de simulación de la arquitectura web. Sin embargo, dado que la arquitectura a analizar *no comprende toda la estructura de la aplicación*, no se podrá obtener toda la información requerida del modelo de simulación.

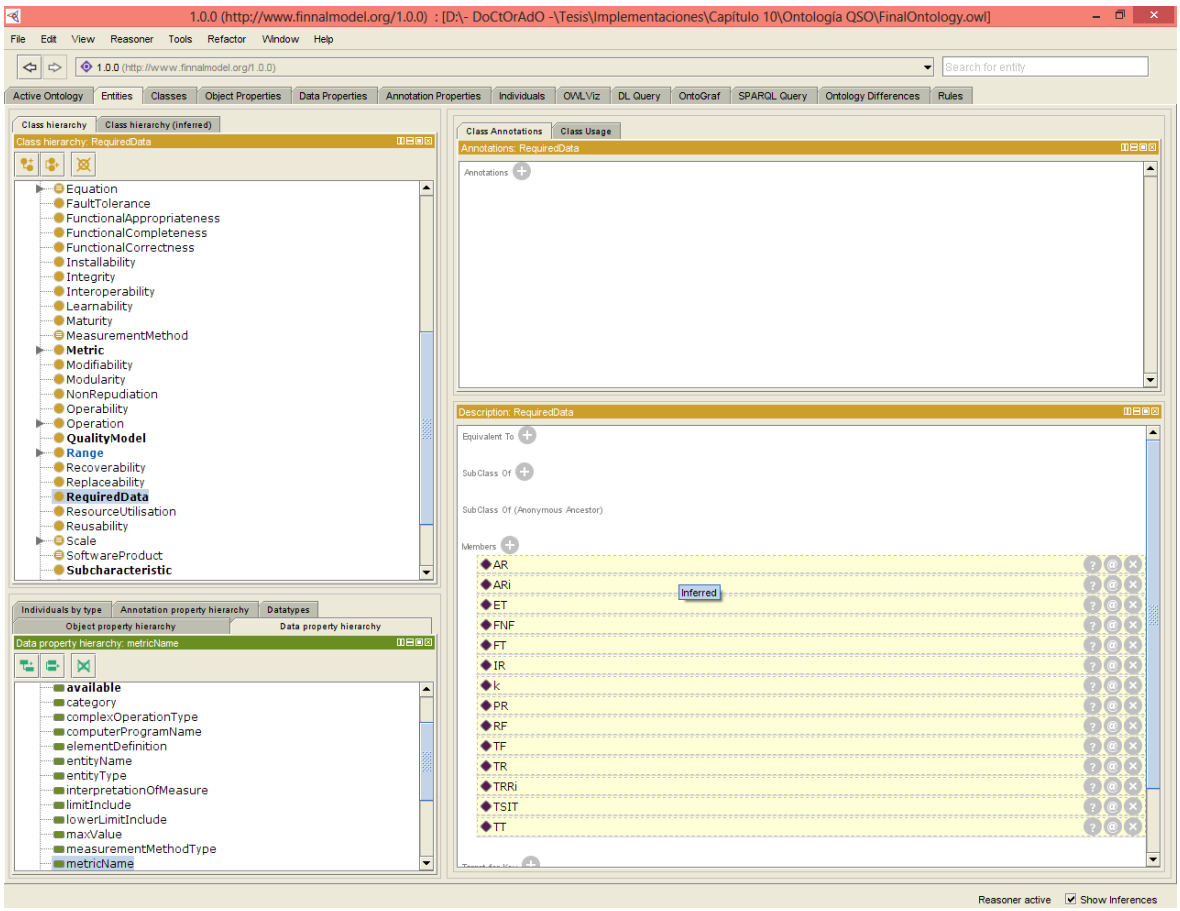


Figura 10.1. Información requerida según el esquema de calidad web propuesto.

Los datos asociados al relevamiento de recursos pertenecen a mediciones tomadas sobre elementos de la capa de infraestructura. Dado que esta capa no formará parte del modelo de simulación, esta información no podrá ser relevada. Luego, estos datos no se encuentran disponibles. Reflejando esta información como parte de la instancia QSO* (según el *marcado de información disponible*), solamente 9 de los 14 elementos requeridos pueden ser obtenidos del modelo de simulación bajo desarrollo[1].

De acuerdo a los lineamientos establecidos como parte del proceso de análisis de la información, una vez indicada la información disponible es necesario determinar el

[1] Excluidos: *AR* (*amount of allocated resources*), *PR* (*amount of pre-defined resources*), *k* (*number of resources requests*), AR_i (*amount of allocated resources of the* i^{th} *request*) y TRR_i (*total requested resources of* i^{th} *request*).

conjunto de métricas que podrá ser calculado haciendo uso de estos datos a fin de obtener las pautas requeridas para modelar los componentes de simulación. En este caso, la Figura 10.2 presenta una captura de pantalla en la cual se visualizan los individuos que (por medio de las inferencias diseñadas) han sido asignados a la clase "CalculableMetric". De los 17 elementos identificados, los individuos resaltados en azul corresponden a 5 de las *métricas de alto nivel* propuestas en el esquema de calidad (ver *Tabla 6.2*). Los elementos restantes se dividen en dos grupos, a saber:

- *Grupo MD (Métricas Directas)*: Incluye las métricas que pueden ser relevadas de forma directa a fin de habilitar el cálculo de las *métricas indirectas de alto nivel*. En este grupo se encuentran *ET*, *TSIT*, *TR*, *IR*, *FT*, *TT*, *FNF*, *TF* y *RF*.
- *Grupo MI (Métricas Indirectas)*: Incluye las métricas que deben ser relevadas por medio de la operación de una o más métricas incluidas en el *grupo MD* en una etapa previa a su utilización como parte del cálculo de las *métricas indirectas de alto nivel*. En este grupo se encuentran *AT*, *CR* y *Tf* (donde, por ejemplo, *AT* = *TT* – *FT*).

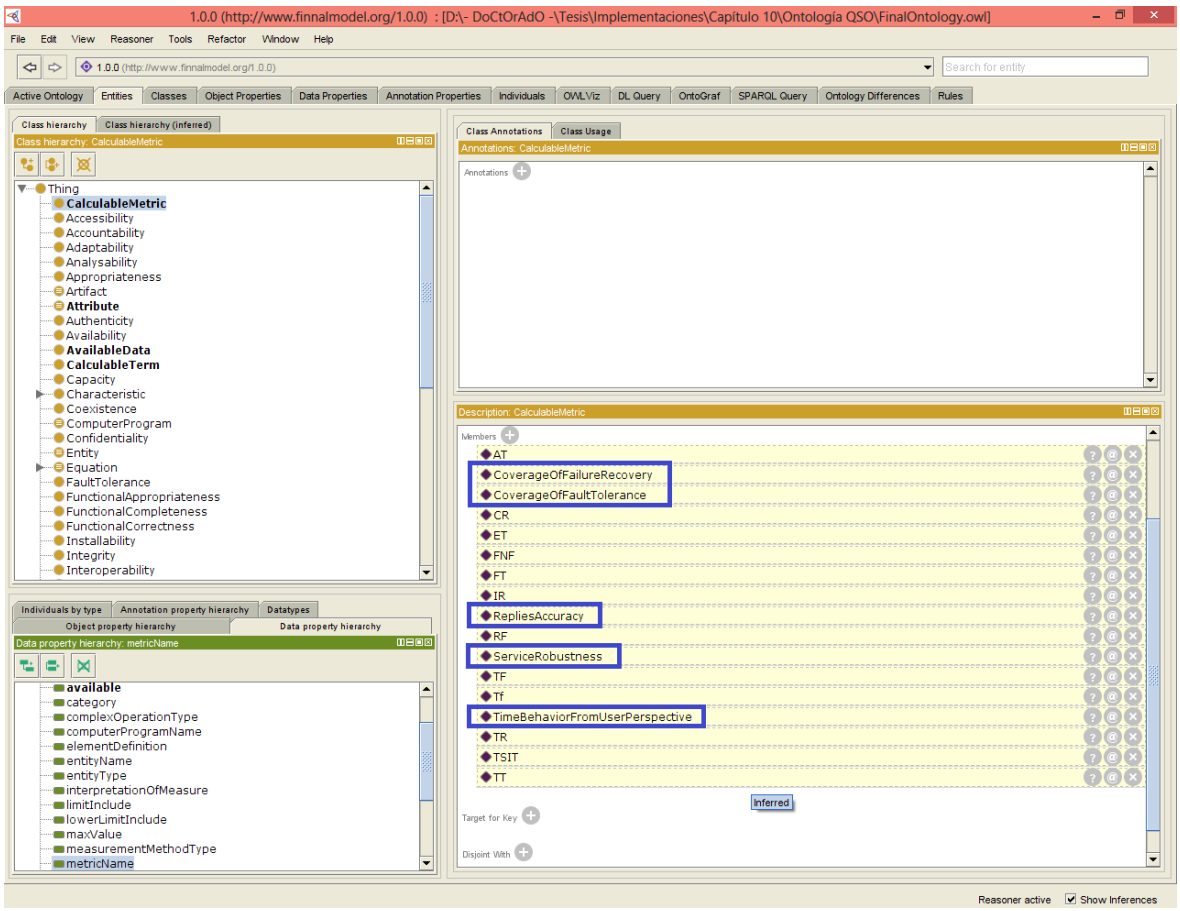


Figura 10.2. Métricas calculables a partir de la información disponible.

Luego, relevando las métricas del grupo *MD* es posible calcular, en una etapa posterior, las 8 métricas restantes (esto es, las 3 que pertenecen al grupo *MI* y, en base a sus resultados, las 5 métricas de alto nivel). La Figura 10.3 esquematiza las dependencias de cálculo entre los distintos grupos de métricas objetivo.

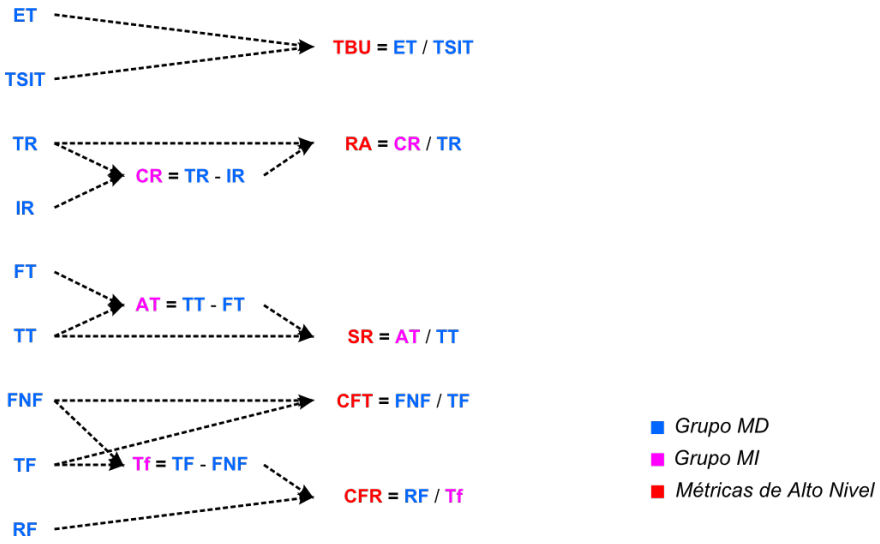


Figura 10.3. Dependencias entre métricas calculables.

10.2.2 Objetivos del Modelo de Simulación

De acuerdo a las relaciones establecidas con anterioridad, si se relevan las métricas incluidas en el grupo *MD* es posible calcular el resto de los valores.

En este contexto, a fin de diseñar un modelo flexible que garantice la posibilidad de extensión futura pero que además permita determinar los valores asociados a todas las métricas requeridas, el modelo de simulación final debe dividirse en dos componentes:

- *Modelo de Simulación de la Arquitectura de Software Web (MSASW)*: Representa el comportamiento del producto de software según su diseño arquitectónico.
- *Modelo de Relevamiento de Métricas (MRM)*: Toma las salidas del modelo *MSASW* y las utiliza para calcular las métricas de calidad requeridas[2].

[2] En esta tesis este modelo se ha incluido en el *marco experimental*. Sin embargo, una vez almacenadas las salidas de *MSASW* de forma coherente, es posible realizar estos cálculos utilizando otro tipo de estrategias.

Modelo de Simulación de la Arquitectura de Software Web (MSASW)

Uno de los principales objetivos de este modelo debe ser el diseño de los componentes arquitectónicos cuyo comportamiento posibilite la obtención de eventos de salida que permitan calcular el subconjunto de métricas mínimo. De esta manera, no se complejiza su definición ya que no se deben diseñar operaciones adicionales para el cálculo de otros indicadores.

En virtud de identificar tales eventos, la Tabla 10.4 presenta una breve descripción de cada una de las métricas directas a relevar.

Nº	VAR.	DESCRIPCIÓN	UNIDAD
1	ET	Tiempo de procesamiento de solicitud de usuario.	Tiempo
2	TSIT	Tiempo total desde que la solicitud ingresa hasta que se emite su respuesta.	Tiempo
3	TR	Cantidad total de solicitudes que han tenido respuesta sobre el servicio.	Solicitudes
4	IR	Cantidad de solicitudes que han sido respondidas incorrectamente[3].	Solicitudes
5	FT	Tiempo que el servicio estuvo inactivo por falla de sus componentes[4].	Tiempo
6	TT	Tiempo total que el servicio estuvo operativo.	Tiempo
7	FNF	Cantidad de defectos que no corresponden a fallas en el servicio[5].	Defectos
8	TF	Cantidad total de defectos detectados en el sistema.	Defectos
9	RF	Cantidad de fallas reparadas en el sistema[6].	Fallas

Tabla 10.4. Estudio de las métricas directas objetivo de la simulación.

[3] Una solicitud se responde de forma incorrecta si, en algún punto de su resolución, ha sido procesada por un componente con funcionamiento defectuoso.
[4] La falla de un componente de aplicación se presenta cuando todas sus réplicas han entrado en estado de falla y, por este motivo, ninguna puede responder las solicitudes de usuario. En el caso de las réplicas asociadas a un componente de aplicación no definido, su falla se produce cuando falla alguno de sus componentes funcionales.
[5] Los defectos se presentan a nivel de componente funcional y afectan únicamente la ejecución actual (es decir, no generan inconvenientes en el resto de ciclo de vida del componente de aplicación).
[6] Una vez que un componente de aplicación entra en falla, la única forma de recuperarlo es removiendo la instancia y generando una nueva.

Como puede observarse, existen dos tipos de resultados: *i)* resultados asociados a las solicitudes (métricas 1 a 4), y *ii)* resultados asociados al servicio o sistema web (métricas 5 a 9). En el primer caso, si los eventos de salida transportan solicitudes, es posible incluir dentro de las mismas la información asociada a su relevamiento. Por su parte, en el segundo caso, basta con que los eventos de salida correspondan al estado de los componentes de aplicación individuales (junto con su identificación).

De esta manera, el modelo de simulación *MSASW* debe plantearse como objetivo la identificación de dos salidas: *solicitudes* y *estado del componente*. Para este último, el conjunto inicial de posibles estados de los componentes de software a modelar en relación a las métricas propuestas puede estar definido como *{ active, active-fault, inactive-failure }*.

Modelo de Relevamiento de Métricas (MRM)

En base a los eventos de salida de *MSASW*, debe computar el cálculo de todas las métricas requeridas. Por este motivo, sus entradas quedan definidas de acuerdo a los eventos de *solicitudes* y *estado del componente*.

Luego, en términos generales, el modelo MRM debe:

- Por cada evento *solicitud* que reciba:

  i) Tomar los valores ET y TSIT para, luego, computar TBU.
  ii) Incrementar en una unidad el valor de TR.
  iii) Si su respuesta ha sido incorrecta, incrementar en una unidad el valor de IR.

- Por cada evento *estado* que reciba:

  i) Determinar si el sistema ha quedado inactivo. En caso afirmativo, comenzar el proceso de cálculo de FT.

ii) Si el estado reporta que un componente no se encuentra funcionando correctamente[7], incrementar en una unidad el valor de TF.

iii) Si el estado reporta que un componente se encuentra defectuoso, incrementar en una unidad el valor de FNF.

iv) Si el estado reporta que un componente ha sido removido a causa de una falla, incrementar en una unidad el valor de RF.

Además, a fin de calcular el valor de TT, el modelo MRM debe mantener un reloj interno asociado al tiempo de simulación transcurrido (ya que se asume que el sistema se encuentra operativo durante todo este tiempo[8]).

10.3 Suposiciones y Restricciones de Diseño

10.3.1 Capa de Infraestructura

Tal como se ha establecido con anterioridad, el modelo de simulación a diseñar en función de la arquitectura de software web no contempla la capa de infraestructura. En este contexto, se asume que los componentes de aplicación que deben ser desplegados sobre los recursos de procesamiento, asociados a las ofertas de almacenamiento y/o vinculados por medio de ofertas de comunicación, pueden llevar a cabo la totalidad de sus funciones sin requerir de la definición explícita de esta interacción.

En este caso, las condiciones de ejecución específicas asociadas a la capa de infraestructura deben indicarse como parte de la información propia de cada uno de los modelos. Se asume durante la simulación que los componentes arquitectónicos ya se encuentran alojados en recursos de tecnología de la información. Además, al momento de determinar el fin de simulación, los componentes alcanzan un estado de

[7] Un componente puede no funcionar correctamente a causa de un defecto o de una falla.
[8] El sistema nunca puede dejar de estar operativo ya que no se modela la infraestructura subyacente.

inactividad asociado a su desalojo de los recursos a los que se encontraban vinculados. Esto es, se asume que el servicio queda no operativo.

Un corolario de la falta de modelado a nivel de infraestructura se vincula con la gestión de los estados de falla. Como se ha establecido en los apartados previos, una vez que un componente de aplicación entra en falla, la única forma de recuperarlo es removiendo la instancia y generando una nueva. Dado que esta acción solo puede ser llevada a cabo por un componente de administración específico que se vincule con los servicios de infraestructura subyacente para dar lugar al desalojo de la instancia actual y la asignación de recursos para una nueva instancia, una vez que los modelos propuestos entren en un estado de falla no podrán recuperarse. Luego, el valor de la *métrica RF* siempre será 0.

10.3.2 Componentes Arquitectónicos

Del conjunto de *componentes arquitectónicos* definidos para el diseño de las arquitecturas web, sólo se modelan los *componentes de aplicación definidos*, los *componentes de aplicación no definidos* y los *componentes funcionales*.

Debido a que los *componentes de administración* gestionan las funcionalidades de la aplicación en relación a los recursos subyacentes (los cuales pertenecen a la capa de infraestructura), no es posible incluir una descripción de estos elementos sin haber definido previamente el comportamiento de dichos recursos. Es decir, los componentes arquitectónicos incluidos en esta categoría refieren a la administración de componentes (no de procesos) por medio de vínculos con los recursos de tecnología de la información. Dado que estos recursos no han sido modelados, no es posible definir un modelo de simulación que refleje el comportamiento de este tipo de componente.

Además, teniendo en cuenta que el modelo bajo diseño corresponde a una primera aproximación de los elementos arquitectónicos requeridos, en los componentes de aplicación no se considera el manejo del estado ni la consistencia de la

información. Esto se debe a que para contemplar estas propiedades es necesario contar con un mecanismo de soporte a nivel de ofertas de almacenamiento (infraestructura). En este sentido, los modelos asociados a cada uno de los componentes arquitectónicos descriptos como parte de la formulación de la arquitectura de software web han sido manejados de forma genérica según su tipo.

10.3.3 Fallas del Sistema Web

Teniendo en cuenta que la falla del sistema web se genera a causa de fallas en sus componentes, y considerando que una vez que los componentes arquitectónicos entren en estado de falla no podrán recuperarse, una vez que el sistema falle no podrá recuperarse. Sin embargo, para que el sistema alcance un estado de falla que le impida seguir funcionando con normalidad deben quedan inactivas todos las instancias de un mismo componente de aplicación. Esto tiene dos implicancias, a saber:

- *Una instancia de un componente puede fallar mientras otra sigue activa (funcionando correctamente)*: En este contexto, todas las solicitudes que se asignen a la instancia en falla no serán respondidas, pero las que se asignen a las réplicas con capacidad de procesamiento (ya sea normal o defectuosa), serán respondidas con normalidad. Luego, el sistema seguirá prestando servicio.
- *La falla de todas las instancias de un componente específico no conllevan a la falla o inactividad de los componentes restantes*: Las instancias no tienen la capacidad de informar a un componente externo que han entrado en falla (por este motivo es que existen mecanismos específicos para su monitoreo como ser, por ejemplo, el esquema de *watchdog*). Luego, cuando se produce una falla en una o más instancias, las réplicas de los componentes activos seguirán funcionando (pues no saben que existe una parte inactiva del servicio).

En este sentido, el modelo de simulación diseñado asume (dada la inexistencia de la capa de infraestructura) que el sistema se encuentra activo en todo momento hasta que se alcance el fin de simulación (momento en el cual se informa a los componentes que han sido desalojados de los recursos -ficticios- de infraestructura). En este contexto, el relevamiento de la *métrica TT* coincide con el tiempo de simulación. Por el contrario, el relevamiento de la *métrica FT* contempla la estructura general del modelo en términos de la cantidad de instancias creadas para cada componente de aplicación. Teniendo en cuenta que los componentes de aplicación informan su estado (de acuerdo a las *salidas definidas como consecuencia de las métricas a relevar*), se mantiene una estructura interna como parte del modelo *MRM* con el objetivo de recibir esta información y determinar (en función de la cantidad de instancias en falla para los distintos componentes) el momento en el cual el sistema ha entrado en falla. Haciendo uso de esta información, el modelo *MRM* también realiza el relevamiento de la *métrica FNF* (contabilizando los estados que informan la ocurrencia de defectos -no fallas-) y de la *métrica TF* (procesando los estados que informan defectos y fallas a fin de determinar la cantidad inconvenientes que han tenido lugar durante la simulación).

10.4 Especificación del Modelo de Simulación MSASW

10.4.1 Especificación de Componente de Aplicación No Definido

Descripción del Modelo de Simulación

En relación a los *componentes no definidos* se aplica un conjunto de lineamientos básicos a fin de especificar su comportamiento. Estos lineamientos sientan las bases requeridas para la formulación de los modelos de simulación asociados.

De acuerdo a lo descripto en el apartado *"Componentes de Aplicación (No Definidos)"*, la especificación de este tipo de elementos se define en base a un conjunto de *componentes funcionales*. Estos últimos definen un conjunto de funciones básicas de

utilidad para la especificación de procesos más complejos en relación a cualquier tipo de aplicación de software web. Tales procesos corresponden a responsabilidades funcionales que han sido asociadas a los *componentes de aplicación no definidos*.

Luego, desde un punto de vista funcional, un *componente de aplicación no definido* puede ser visto como una secuencia explícita de funciones básicas cuyo objetivo consiste en la ejecución de una responsabilidad funcional (cuyo nivel de complejidad es mayor que el de las funciones simples) a fin de dar respuesta al procesamiento de las solicitudes de los usuarios. Bajo esta perspectiva, los componentes funcionales no poseen entidad propia sino que describen una *función de trabajo elemental* a ser utilizada como herramienta base en la definición de funciones más complejas. La definición de esta función depende del tipo de componente funcional bajo estudio.

Sin embargo, en su forma más simple, la *función de trabajo elemental* puede reducirse a una *función de procesamiento básica* con las siguientes características:

- Una vez invocada, se activa por un período de tiempo dado (*processingTime*) y luego retorna el control (y posiblemente un resultado) a quien la invocó.
- El algoritmo que describe el procesamiento a realizar no siempre se comporta de forma correcta, pudiendo generar: *i)* un procesamiento defectuoso de algunas solicitudes, o *ii)* fallas en su invocación.

Luego, si un componente funcional presenta un defecto, el componente de aplicación podrá seguir funcionando. Por el contrario, si una función falla (es decir, un componente funcional falla), el componente de aplicación no podrá cumplir con su responsabilidad. Luego, el componente de alto nivel falla como consecuencia de la falla de alguno de sus componentes internos.

En ese contexto, el modelo de simulación de *componente de aplicación no definido* propone que cada *componente funcional* quede especificado como una etapa de procesamiento (*processing*) configurada de acuerdo a su *tiempo de procesamiento*

(*processingTime*), *probabilidad de defecto* (*faultProbability*) y *probabilidad de falla dado un defecto* (*failureProbability*)[9]. La determinación del estado en el cual se encuentra una función específica debe realizarse en una etapa previa a su invocación (a fin de determinar si se podrá o no cumplir con el procesamiento requerido).

En este sentido, los posibles estados de un *componente de aplicación no definido* se corresponden con la combinación de perspectivas individuales, según:

- Su estado en relación a la infraestructura: asignado/no asignado a recursos de tecnología de la información (*running* / *expired*).
- Su estado en relación a su actividad: activo (*active*) y falla (*failure*).
- Su estado en relación al procesamiento: correcto (*ok*), defectuoso (*fault*).

Sin embargo, estos estados no son independientes (Tabla 10.5). Un componente arquitectónico que se encuentra en estado *expired* no puede tener asociado ningún estado de procesamiento ni de actividad (dado que no se encuentra en ejecución, no puede encontrarse en estado de falla ni procesando solicitudes de forma correcta o defectuosa). Para que un componente se encuentre en actividad (ya sea *active* o *failure*), el componente debe estar en estado *running*. Así mismo, teniendo en cuenta que por su definición un componente que entra en falla (*failure*) no se encuentra brindando servicio, un estado de procesamiento únicamente es aplicable si el estado de actividad es *active*.

Tabla 10.5. Estados combinados válidos en componentes de aplicación no definidos.

INFRAESTRUCTURA	ACTIVIDAD	PROCESAMIENTO
expired	-	-
running	*Failure*	-

[9] Teniendo en cuenta que las fallas también corresponden a un defecto, una vez que se ha determinado que el procesamiento se encuentra en un estado defectuoso, se debe evaluar si dicho estado corresponde a una falla que impide la invocación de la función.

running	*Active*	*ok*
running	*Active*	*fault*

De acuerdo con los estados precedentes, un componente que se encuentra en estado (*running*, *active*, *ok*) debe estar en alguna de las siguientes etapas: esperando solicitudes (*waiting*) o procesando solicitudes. Así mismo, cuando el componente se encuentra en estado (*running*, *failure*, -) no puede estar realizando ninguna actividad, por lo que se considera que se encuentra en una etapa de falla (*failure*). Por otra parte, si el componente se encuentra en estado (*expired*, -, -), no se encuentra operativo por lo que se encuentra en una etapa de inactividad (*inactive*).

En el caso de que el componente se encuentre procesando solicitudes, su comportamiento puede estar asociado a un procesamiento correcto (*processing*) o un procesamiento incorrecto (*processing with faults*). Como se ha mencionado con anterioridad, los componentes de aplicación se encuentran compuesto por múltiples funciones (*componentes funcionales*) que deben ser ejecutadas en un orden específico (vínculos entre componentes funcionales) con el objetivo de procesar las solicitudes de usuario. Dado que cada componente funcional tiene asociada una *probabilidad de defecto* y una *probabilidad de falla dado un defecto*, cada una de las funciones individuales puede encontrarse actuando correctamente, de forma defectuosa o, directamente, en falla. Si una función falla, todo el componente falla. Sin embargo, si una función se encuentra defectuosa, el componente de alto nivel puede seguir funcionando. Luego, si un componente de aplicación no definido se encuentra compuesto de N componentes funcionales, el componente de alto nivel tiene asociadas N etapas *processing* y *processing with faults* (una para cada función a ejecutar) pero una única etapa de falla (*failure*). Bajo esta perspectiva, las etapas $processing_i$ y $processing\ with\ faults_i$ (con $1 \leq i \leq N$) corresponden a los estados (*running*, *active*, *ok*) y (*running*, *active*, *fault*) respectivamente.

En relación a las etapas detalladas, los componentes de simulación inician su ejecución en etapa *waiting*. Esto implica que su estado es (*running*, *active*, *ok*), lo que indica que el componente de aplicación se encuentra alojado en un recurso de infraestructura estando activo y con un comportamiento normal. Ante esta situación, dado que el componente no se encuentra realizando ningún tipo de procesamiento, no puede evolucionar de forma autónoma hacia otra etapa. Sin embargo, como consecuencia del arribo de estímulos externos, el componente puede evolucionar hacia nuevas etapas de trabajo. En este contexto puede:

i) Evolucionar hacia *inactive* como consecuencia de la recepción de un estímulo externo que indica que ha sido desalojado (*out-of-use*) del recurso de tecnología de la información que tenía asociado.

ii) Evolucionar hacia *processing*$_1$, *processing with faults*$_1$ o *failure* como consecuencia de la recepción de una solicitud de usuario (*request*) que debe ser atendida.

En el caso ii), la evolución depende del comportamiento asociado a la primera función elemental que debe ser ejecutada para resolver el procesamiento objetivo (esto es, el *componente funcional* que no posee vínculos de entrada), a saber:

- Si esta función se encuentra procesando correctamente, podrá atender la solicitud. Luego, la solicitud debe ser almacenada y el componente debe evolucionar a la etapa *processing*$_1$.
- Si esta función se encuentra procesando de manera defectuosa, atenderá la solicitud pero con una operatoria incierta. Luego, debe almacenarse la solicitud indicando que será procesada bajo un comportamiento incierto (de forma incorrecta) y el componente debe evolucionar a la etapa *processing with faults*$_1$.
- Si esta función se encuentra en falla, no podrá atender la solicitud. Luego, la solicitud debe descartarse y el componente debe evolucionar a la etapa *failure*.

Una vez que el componente ha comenzado a procesar una solicitud (ya sea de forma correcta o incorrecta), debe continuar con la secuencia de funciones (definida en su descripción) salvo que arribe un estímulo que indique que ha sido desalojado (*out-of-use*) del recurso de infraestructura subyacente. De no producirse un desalojo, las etapas hacia las cuales puede evolucionar son independientes del tipo de procesamiento realizado con anterioridad. Luego, al tener que ejecutar la función *i+1* desde la etapa $processing_i$ o $processing\ with\ faults_i$ (con $1 \leq i \leq N-1$), el componente puede:

- Evolucionar a la etapa $processing_{i+1}$ (si la función *i+1* se encuentra procesando correctamente), actualizando la información de la solicitud almacenada.
- Evolucionar a la etapa $processing\ with\ faults_{i+1}$ (si la función *i+1* se encuentra procesando de manera defectuosa), indicando que la solicitud almacenada será procesada bajo un comportamiento incierto (de forma incorrecta).
- Evolucionar a la etapa *failure* (si la función *i+1* se encuentra en falla), descartando la solicitud.

Si la función actual corresponde $i = N$, el componente de alto nivel ha finalizado el procesamiento de la solicitud. Luego, debe evolucionar hacia la etapa *waiting* enviando la solicitud con su respuesta hacia el siguiente componente de aplicación diseñado en la arquitectura (indicando que no existe solicitud asociada al componente de alto nivel luego del cambio de etapa).

En todos los casos, si se produce un desalojo mientras el componente se encuentra en cualquier etapa de procesamiento (ya sea correcto o con errores) de la secuencia definida, se debe evolucionar hacia la etapa *inactive* (descartando la solicitud actual).

Una vez que el componente ingresa a la etapa *failure*, no puede evolucionar hacia una nueva etapa hasta que arribe un estímulo externo de desalojo (*out-of-use*). Dicho estímulo genera una evolución hacia la etapa *inactive*. Desde la etapa *inactive*, el

componente no puede evolucionar en ningún caso hacia una nueva etapa. Esto se debe a que se considera que ha finalizado su ejecución[10].

Todas las solicitudes de usuario (*request*) que arriben al modelo cuando el componente se encuentra en una etapa distinta de *waiting*, deben ser descartadas (ya que el componente estará atendiendo una solicitud, en estado de falla o inactivo -por lo que no podrá dar curso a nuevas peticiones-).

La Tabla 10.6 resume los estados, las etapas y el conjunto de posibles evoluciones entre los mismos. De esta secuencia, se observa que existen dos posibles estímulos externos (que se traducen a eventos de entrada del modelo de simulación) a considerar: *out-of-use* y *request*. En el caso del evento *out-of-use* no existen propiedades adicionales a considerar, ya que el evento en sí mismo representa toda la información necesaria para que tenga lugar la evolución de las etapas asociadas. Sin embargo, para el evento *request* la evolución entre etapas requiere de la actualización de información asociada al estímulo específico. La Tabla 10.7 identifica un subconjunto de las propiedades requeridas sobre este tipo de estímulo para la manipulación de este evento. Como puede observarse, estas propiedades buscan representar los resultados asociados a las *métricas propuestas para el relevamiento de la calidad de la aplicación en relación a las solicitudes*.

Tabla 10.6. Evolución de estados en componentes de aplicación no definidos.

ESTADO	ETAPA	EVOLUCIÓN DE ETAPA	EVOLUCIÓN DEL ESTADO
(running, active, ok)	*waiting*	*waiting → inactive*$^{(*)}$	*(expired, -, -)*
		waiting → processing$_1$$^{(**)}$	*(running, active, ok)*
		waiting → processing with faults$_1$$^{(**)}$	*(running, active, fault)*

[10] Cuando se requiera una nueva instancia de este componente, se creará una nueva (no se reactivará una previa).

(running, active, fault)	*processing*$_i$ [(***)]	*waiting → failure*[(**)]	*(running, failure, -)*
		processing$_i$ *→ processing*$_{i+1}$	*(running, active, ok)*
		processing$_i$ *→ processing with faults*$_{i+1}$	*(running, active, fault)*
		processing$_i$ *→ failure*	*(running, failure, -)*
		processing$_i$ *→ inactive*[(*)]	*(expired, -, -)*
	processing$_N$	*processing*$_N$ *→ waiting*	*(running, active, ok)*
		processing$_N$ *→ inactive*[(*)]	*(expired, -, -)*
	processing with faults$_i$ [(***)]	*processing with faults*$_i$ *→ processing*$_{i+1}$	*(running, active, ok)*
		processing with faults$_i$ *→ processing with faults*$_{i+1}$	*(running, active, fault)*
		processing with faults$_i$ *→ failure*	*(running, failure, -)*
		processing with faults$_i$ *→ inactive*[(*)]	*(expired, -, -)*
	processing with faults$_N$	*processing with faults*$_N$ *→ waiting*	*(running, active, ok)*
		processing with faults$_N$ *→ inactive*[(*)]	*(expired, -, -)*
(running, failure, -)	*failure*	*failure → inactive*[(*)]	*(expired, -, -)*
(expired, -, -)	*inactive*	-	-

(*) Por arribo de un desalojo (*out-of-use*).
(**) Por arribo de una solicitud (*request*).
(***) *$1 \leq i \leq N-1$ con $N \neq 1$.*

Tabla 10.7. Propiedades de métricas asociadas al evento "request".

DENOMINACIÓN	DESCRIPCIÓN	ACTUALIZACIÓN	UNIDAD
executionTime (*)	Tiempo que ha estado en etapas del tipo *processing* o *processing with faults*.	Cuando el evento está por abandonar una etapa del tipo *processing* o *processing with faults* se acumula en *processingTime* el tiempo transcurrido en dicha etapa.	Tiempo
totalTime (**)	Tiempo que la solicitud ha estado dentro del servicio.	Cuando el evento está por abandonar una etapa cualquiera se acumula en *totalTime* el tiempo transcurrido en dicha etapa.	Tiempo
incorrect (***)	Bandera que indica si la solicitud ha pasado por alguna etapa del tipo *processing with faults*.	Se coloca en *true* cuando el evento está por ser transferido a una etapa del tipo *processing with faults*.	-

(*) Esto corresponde al relevamiento de la *métrica ET* de la solicitud actual.
(**) Esto corresponde al relevamiento de la *métrica TSIT* de la solicitud actual.
(***) Esto corresponde a parte del relevamiento de la *métrica IR*.

Mapeo de la Descripción Arquitectónica al Modelo de Simulación

La Figura 10.4 presenta un *componente de aplicación no definido* denominado *Example* compuesto de cuatro componentes funcionales: *user interface component*, *processing component (1)*, *data access component* y *processing component (2)*. Esta representación ha sido generada haciendo uso de la *herramienta de modelado* implementada en base al metamodelo propuesto en el *Capítulo 8*. En relación a cada uno de los componentes definidos, la Tabla 10.8 indica el conjunto de parámetros a fijar con el objetivo de mapear esta descripción arquitectónica a un modelo de simulación específico diseñado como modelo esencial RDEVS (nivel de componente ya que, luego,

se instanciaran múltiples réplicas de *Example* sobre las cuales se lleva a cabo el proceso de ruteo de solicitudes).

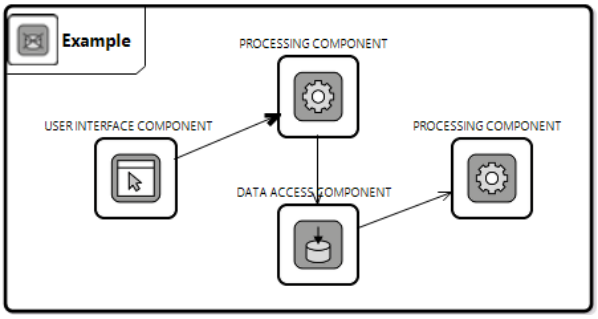


Figura 10.4. Diseño arquitectónico de un componente de aplicación no definido.

Tabla 10.8. Atributos (parámetros) a fijar en los componentes arquitectónicos.

COMPONENTE	PARÁMETRO	
Aplicación no definido	*name*	Nombre del componente.
User Interface Component *Processing Component (1)* *Data Access Component* *Processing Component (2)*	*processingTimeDistribution*	Distribución de probabilidad a ser utilizada para la obtención del *processingTime*.
	faultProbability	Probabilidad de que se presente un defecto.
	failureProbability	Probabilidad de que dado un defecto, el mismo corresponda a una falla.
	faultDistribution	Distribución de probabilidad que representa la forma en la cual se generan los defectos cuyo objetivo es obtener un valor comparable con *faultProbability*.
	failureDistribution	Distribución de probabilidad que representa la forma en la cual se generan los errores cuyo objetivo es obtener un valor comparable con

failureProbability.

De acuerdo a lo descripto en los apartados previos, el modelo de simulación de un *componente de aplicación no definido* debe:

- Manipular eventos *request* definidos en función del tipo de solicitud realizada y las propiedades de la Tabla 10.7.
- Recibir *eventos de entrada request* y *out-of-use*.
- Enviar *eventos de salida request* y *component-state* (que informen el nombre del componente arquitectónico de alto nivel y el *estado* actual en el cual se encuentra como parte de su funcionamiento).
- Quedar definido en base a una etapa *waiting*, una etapa *failure*, una etapa *inactive*, cuatro etapas *processing* (una para cada componente funcional) y cuatro etapas *processing with faults* (una para cada componente funcional).

Teniendo en cuenta que el modelo de simulación debe manipular distintos tipos de solicitudes, es necesario incluir como parte de la especificación del estímulo *request* los elementos propios de su definición. No todas las solicitudes de usuario son iguales. Cada solicitud tiene asociada un pedido específico cuya resolución involucra su procesamiento por un subconjunto de componentes arquitectónicos. Dicho subconjunto queda definido como parte del tipo de pedido asociado a la solicitud.

Luego, un elemento *request* queda definido como

$$request = (metricsInfo , typeInfo) \quad (10.1)$$

donde

metricsInfo = (executionTime, totalTime, incorrect) siendo cada elemento una propiedad definida en la Tabla 10.7.,

typeInfo = (id, type, componentsList) con

id como identificador único de solicitud,

type como tipo de solicitud,

componentsList como listado ordenado de los nombres de los componentes arquitectónicos por los cuales debe ser procesado el elemento *request*[11].

Esta definición implica que el *request* conoce los destinos por los cuales debe ser procesado y el orden en el cual debe irlos recorriendo. Si se elimina el primer elemento de *componentsList* a medida que el *request* es procesado por un componente, el primer elemento de la lista siempre indicará el siguiente destino. Cuando la lista quede vacía, se habrá finalizado el procesamiento de la solicitud (es decir, se habrá resuelto el pedido).

Luego, la Figura 10.5 presenta una representación gráfica de la definición del modelo de simulación asociado al componente *Example*. En esta figura se esquematizan las posibles transiciones entre las distintas etapas (las cuales quedan definidas como fases -*phase*- del modelo)[12]. En este contexto, como una primera aproximación, el conjunto de posibles estados internos del modelo esencial RDEVS queda definido de acuerdo a una fase y al tiempo que el modelo debe permanecer en dicho estado: *(phase, σ*[13] *)*. En cada fase del tipo *processing* y *processing with faults*, el valor de *σ* debe asociarse al *processingTime* definido para el componente funcional representado en dicha etapa. Dado que el tiempo de procesamiento de una función depende de las

[11] Este listado no puede finalizar en un *componente de aplicación definido*, ya que este tipo de elementos se encarga de gestionar funciones (esto es, no procesan las solicitudes sino que gestionan su distribución). Luego, el *componentList* debe finalizar en un *componente de aplicación no definido*.
[12] No se detallan las transiciones asociadas al arribo de un evento *request* en fases distintas a *waiting*.
[13] Que corresponde al "tiempo que debe transcurrir hasta la siguiente transición interna del modelo".

condiciones de ejecución específicas del entorno sobre el cual está siendo ejecutada, el *processingTime* no se corresponde con un valor fijo. Luego, dicho valor se obtiene de la *processingTimeDistribution* asociada al componente funcional (la cual debe ser configurada según los intereses de quien modela el sistema conforme la información disponible del componente funcional). Así mismo, considerando que las fases *waiting* (fase asociada al estado inicial indicada con líneas de punto), *failure* e *inactive* se corresponden a etapas de las cuales el modelo no puede evolucionar si no se produce un estímulo externo, dichas fases quedan asociadas a estados pasivos ($\sigma = \infty$).

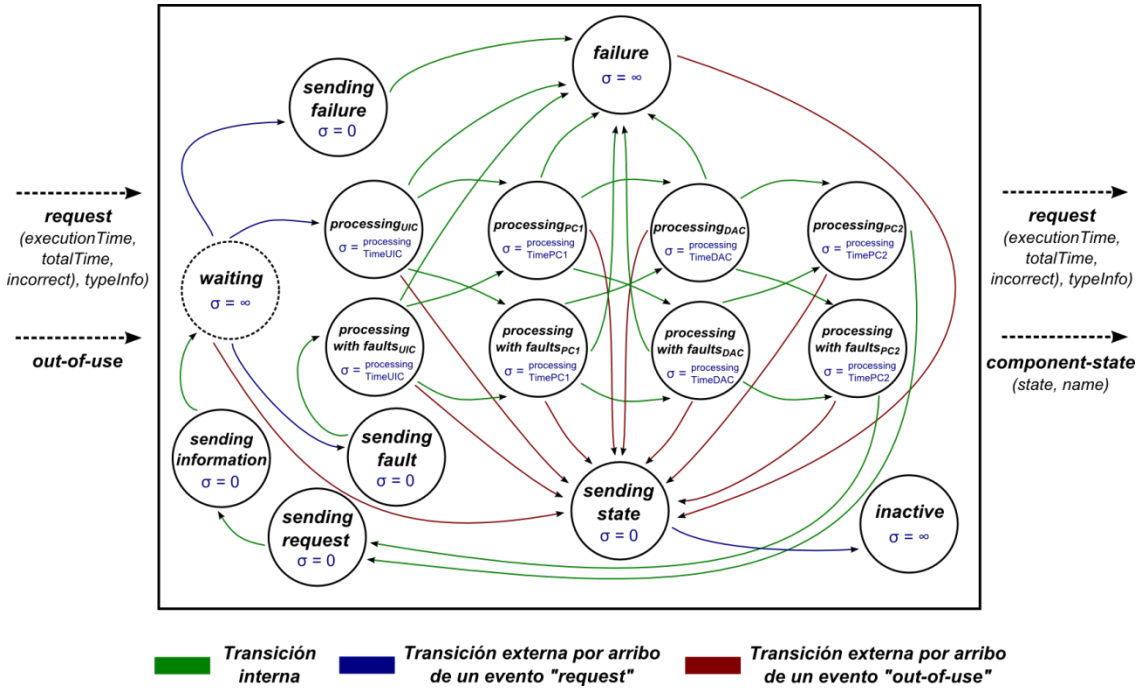


Figura 10.5. Esquema de transiciones, entradas y salidas del modelo "Example".

Considerando que al ejecutar una transición interna el modelo debe emitir una salida para posteriormente cambiar de estado, como consecuencia de todas las transiciones internas definidas en la Tabla 10.6 se envía un evento del tipo *component-state* que informa el estado hacia el cual el componente está por evolucionar. Dado que la evolución hacia el estado *inactive* se produce como consecuencia del arribo de un

evento *out-of-use*, se incorpora un estado transitorio ($\sigma = 0$) denominado *sending state* que (efectivamente) realiza el envío del informe del estado resultante en una instancia previa a quedar en el estado pasivo. La misma estrategia es utilizada para informar los estados defectuosos y las fallas que se producen como consecuencia del intento de procesamiento en el componente UIC cuando arriba un *request* estando en fase *waiting*. En estos casos, se utilizan dos estados transitorios ($\sigma = 0$) definidos por las fases *sending fault* y *sending failure*.

Cuando el componente finaliza el procesamiento de una solicitud (evolución $processing_{CP2} \rightarrow waiting$ o $processing\ with\ faults_{CP2} \rightarrow waiting$), debe enviar un evento *request* con el resultado obtenido en la solicitud que acaba de procesar. Luego, a fin de que el modelo conozca (en cada posible estado) la solicitud bajo procesamiento, se incorpora el elemento *actualRequest* como parte de la definición del estado. De esta forma, el estado queda definido (hasta el momento) como *(phase, σ, actualRequest)*. Para el envío de dicha solicitud, se define un nuevo estado transitorio ($\sigma = 0$) asociado a la fase *sending request*. Una vez que el modelo entra en este estado, realiza el envío de la solicitud hacia sus destinatarios y, luego, continúa hacia un nuevo estado transitorio con fase *sending information*. Desde este último, envía información de estado hacia el *componente de aplicación definido* que lo gestiona con el objetivo de informarle que ha finalizado su tarea de procesamiento (y, por lo tanto, se encuentra liberado para recibir nuevas solicitudes). Luego, regresa a un estado pasivo con fase *waiting*.

En este contexto, para definir la forma en la cual se producen las transiciones entre estados de procesamiento (y las salidas asociadas) es necesario conocer el estado en el cual se encuentra la función que será invocada en el siguiente paso (donde cada función corresponde a un componente funcional). Con este objetivo, se ha definido un algoritmo de determinación de estado básico (Figura 10.6) que, haciendo uso de las probabilidades y distribuciones de defectos y fallas indicadas como parte de la especificación de los componentes funcionales, determina el estado de la función asociada.

La evaluación de este algoritmo debe producirse en un estado previo a la transición requerida. Esto se debe a que el modelo esencial define su función de salida y su función de transición interna en base al estado actual. Dado que la determinación de ambos resultados depende del estado de la función a ser invocada como consecuencia del fin de procesamiento actual, y considerando que una vez que el modelo ingresa a un estado su definición no puede alterarse, esta información debe conocerse con anterioridad a fin de generar los eventos de salida y el cambio de estado en función del estado actual.

```
getNextFunctionState(float faultProbability, float failureProbability) {
     faultRandomNumber = getRandomNumber(faultDistribution);
     if(faultRandomNumber < faultProbability) {
          failureRandomNumber = getRandomNumber(failureDistribution);
          if(failureRandomNumber < failureProbability) return failure;
          else return fault;
     else return ok;
}
```

Figura 10.6. Determinación del estado de una función (componente funcional).

Luego, el estado del modelo queda definido como *(phase, σ, actualRequest, nextFunctionState)* donde *nextFunctionState* es calculado haciendo uso del algoritmo *getNextFunctionState()*.

Modelo Esencial "Example"

El modelo esencial RDEVS del componente de aplicación no definido *Example* queda definido por la estructura

$$Example = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau > \quad (10.2)$$

donde:

$X \equiv$ { *request* , *out-of-use* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

$Y \equiv$ { *request* , *component-state* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

component-state = (state, "Example") donde *state ∈ { (running, active, ok), (running, active, fault), (running, failure), expired }*

$S \equiv$ *phase x σ x actualRequest x nextFunctionState* con

phase ∈ { waiting, failure, inactive, sending fault, sending failure, sending request, sending state, sending information, processing UIC, processing with faults UIC, processing PC1, processing with faults PC1, processing DAC, processing with faults DAC, processing PC2, processing with faults PC2 }

$\sigma \in \mathbb{R}_0^+$

actualRequest ∈ { (metricsInfo , typeInfo) } ∪ { ø }

nextFunctionState ∈ { ok, failure, fault } ∪ { - }

$$\delta_{int}(s) = \begin{cases} \delta_{int}(\text{sending request},\sigma,\text{actualRequest},\text{nextFunctionState}) = \\ (\text{sending information}\ ,\ 0,\ \text{actualRequest},\ -) \\ \\ \delta_{int}(\text{sending information},\sigma,\text{actualRequest},\text{nextFunctionState}) = \\ (\text{waiting},\ \infty,\ \emptyset, \\ \text{getNextFunctionState}(\text{faultProbabilityUIC},\text{failureProbabilityUIC})) \\ \\ \delta_{int}(\text{sending state},\sigma,\text{actualRequest},\text{nextFunctionState}) = \\ (\text{inactive},\ \infty,\ \emptyset,\ -) \\ \\ \delta_{int}(\text{sending fault},\sigma,(\text{executionTime},\ \text{totalTime},\ \text{incorrect}),\text{typeInfo}), \\ \text{nextFunctionState}) = \\ (\text{processing with faults UIC},\text{processingTimeUIC}, \\ ((\text{executionTime} + \text{processingTimeUIC},\ \text{totalTime} + \text{processingTimeUIC}, \\ \text{true}),\text{typeInfo}), \\ \text{getNextFunctionState}(\text{faultProbabilityPC1},\text{failureProbabilityPC1}) \\ \\ \delta_{int}(\text{sending failure},\sigma,\text{actualRequest},\text{nextFunctionState}) = \\ (\text{failure},\ \infty,\ \emptyset,\ -) \\ \\ \delta_{int}(\text{phase},\sigma,\text{actualRequest},\text{failure}) = \\ (\text{failure},\ \infty,\ \emptyset,\ -) \\ \\ \delta_{int}(\text{processing PC2},\sigma,\text{actualRequest},-) = \\ \delta_{int}(\text{processing with faults PC2},\sigma,\ \text{actualRequest},\ -) = \\ (\text{sending request},\ 0,\ \text{actualRequest},-) \end{cases}$$

$$\delta_{int}(s) = \begin{cases} \delta_{int}(\text{processing UIC},\sigma,((\text{executionTime},\text{totalTime},\text{incorrect}),\text{typeInfo}),\text{ok}) = \\ \delta_{int}(\text{processing with faults UIC},\sigma,((\text{executionTime},\text{totalTime},\text{incorrect}),\\ \text{typeInfo}),\text{ok}) = \\ (\text{processing PC1}, \text{processingTimePC1},\\ ((\text{executionTime} + \text{processingTimePC1}, \text{totalTime} + \text{processingTimePC1},\\ \text{incorrect}), \text{typeInfo}),\\ \text{getNextFunctionState}(\text{faultProbabilityDAC}, \text{failureProabadilityDAC})) \\ \\ \delta_{int}(\text{processing PC1},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}), \text{typeInfo}),\text{ok}) = \\ \delta_{int}(\text{processing with faults PC1},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\\ \text{typeInfo}),\text{ok}) = \\ (\text{processing DAC}, \text{processingTimeDAC},((\text{executionTime}+\text{processingTimeDAC},\\ \text{totalTime} + \text{processingTimeDAC}, \text{incorrect}), \text{typeInfo}),\\ \text{getNextFunctionState}(\text{faultProbabilityPC2},\text{failureProbabilityPC2})) \\ \\ \delta_{int}(\text{processing DAC},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}), \text{typeInfo}),\text{ok}) = \\ \delta_{int}(\text{processing with faults DAC},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\\ \text{typeInfo}),\text{ok}) = \\ (\text{processing PC2}, \text{processingTimePC2}, ((\text{executionTime} + \text{processingTimePC2},\\ \text{totalTime} + \text{processingTimePC2}, \text{incorrect}) ,\text{typeInfo}), \text{-}) \\ \\ \delta_{int}(\text{processing UIC},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}), \text{typeInfo}),\text{fault}) = \\ \delta_{int}(\text{processing with faults UIC},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\\ \text{typeInfo}), \text{fault}) = \\ (\text{processing with faults PC1}, \text{processingTimePC1},\\ ((\text{executionTime} + \text{processingTimePC1}, \text{totalTime} + \text{processingTimePC1},\\ \text{true}), \text{typeInfo}),\\ \text{getNextFunctionState}(\text{faultProbabilityDAC},\text{failureProbabilityDAC})) \\ \\ \delta_{int}(\text{processing PC1},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\text{typeInfo}),\text{fault}) = \\ \delta_{int}(\text{processing with faults PC1},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\\ \text{typeInfo}), \text{fault}) = \\ (\text{processing with faults DAC}, \text{processingTimeDAC},\\ ((\text{executionTime} + \text{processingTimeDAC}, \text{totalTime} + \text{processingTimeDAC},\\ \text{true}), \text{typeInfo}),\\ \text{getNextFunctionState}(\text{faultProbabilityPC2},\text{failureProbabilityPC2})) \\ \\ \delta_{int}(\text{processing DAC},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\text{typeInfo}),\text{fault}) = \\ \delta_{int}(\text{processing with faults DAC},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}),\\ \text{typeInfo}), \text{fault}) = \\ (\text{processing with faults PC2}, \text{processingTimePC2},\\ ((\text{executionTime} + \text{processingTimePC2}, \text{totalTime} + \text{processingTimePC2},\\ \text{true}), \text{typeInfo}), \text{-}) \end{cases}$$

$$\delta_{ext}(s,e,x) = \begin{cases} \delta_{ext}(s,e,\text{out-of-use}) = (\text{sending state}, 0, \emptyset, -) \\ \\ \delta_{ext}((\text{waiting},\sigma,\text{actualRequest},\text{ok}),e,((\text{executionTime}, \text{totalTime}, \text{incorrect}), (\text{id},\text{type},\text{componentsList}))) = \\ (\text{processing UIC}, \text{processingTimeUIC}, \\ ((\text{executionTime} + \text{processingTimeUIC}, \text{totalTime} + \text{processingTimeUIC}, \\ \text{incorrect}), (\text{id},\text{type},\text{componentsList.withoutFirst}())), \\ \text{getNextFunctionState}(\text{faultProbabilityPC1},\text{failureProbabilityPC1}) \\ \\ \delta_{ext}((\text{waiting},\sigma,\text{actualRequest},\text{fault}),e,(\text{metricsInfo},(\text{id},\text{type},\text{componentsList})\}) = \\ (\text{sending fault},0, \{\text{metricsInfo},(\text{id},\text{type},\text{componentsList.withoutFirst}())),-) \\ \\ \delta_{ext}((\text{waiting},\sigma,\text{actualRequest},\text{failure}),e,\text{request}) = \\ (\text{sending failure}, 0, \emptyset, -) \\ \\ \delta_{ext}((\text{phase},\sigma,((\text{executionTime}, \text{totalTime}, \text{incorrect}), (\text{typeInfo})), \\ \text{nextFunctionState}), e, \text{request}) = (\text{phase},\sigma\text{-}e, ((\text{executionTime} + e, \\ \text{totalTime} + e, \text{incorrect}), (\text{typeInfo})) , \text{nextFunctionState}) \end{cases}$$

$$\lambda(s) = \begin{cases} \lambda(\text{sending state}, \sigma, \text{actualRequest}, \text{nextFunctionState}) = (\text{expired}, \text{"Example"}) \\ \\ \lambda(\text{sending request}, \sigma, \text{actualRequest}, \text{nextFunctionState}) = \\ \lambda(\text{sending information}, \sigma, \text{actualRequest}, \text{nextFunctionState}) = \\ \text{actualRequest} \\ \\ \lambda(\text{processing PC2}, \sigma, \text{actualRequest}, \text{nextFunctionState}) = \\ \lambda(\text{processing with faults PC2}, \sigma, \text{actualRequest}, \text{nextFunctionState }) = \\ \lambda(\text{phase}, \sigma, \text{actualRequest}, \text{ok}) = \\ ((\text{running},\text{active},\text{ok }),\text{"Example"}) \\ \\ \lambda(\text{sending failure}, \sigma, \text{actualRequest}, \text{nextFunctionState}) = \\ \lambda(\text{phase}, \sigma, \text{actualRequest}, \text{failure}) = \\ ((\text{running}, \text{failure}), \text{"Example"}) \\ \\ \lambda(\text{sending fault}, \sigma, \text{actualRequest}, \text{nextFunctionState}) = \\ \lambda(\text{phase}, \sigma, \text{actualRequest}, \text{fault}) = \\ ((\text{running}, \text{active}, \text{fault}), \text{"Example"}) \end{cases}$$

$$\tau(s) = \tau(\text{ phase}, \sigma, \text{actualRequest}, \text{nextFunctionState }) = \sigma$$

10.4.2 Especificación de Componente de Aplicación Definido

Como se ha establecido en el apartado *"Componentes de Aplicación (Definidos)"*, estos elementos arquitectónicos refieren a componentes de software que son comúnmente utilizados como parte del diseño arquitectónico de las aplicaciones de software web. Tales componentes se vinculan a *componentes de administración* específicos (*Balanceador de Carga Elástico* para el caso del *Balanceador de Carga* y *Cola de Mensajes Elástica* para la *Cola de Mensajes*) a fin de intercambiar información relacionada con la cantidad de réplicas requeridas/activas del *componente de aplicación no definido* que se encuentran gestionando en un momento.

El comportamiento asociado a cada *componente de aplicación definido* queda determinado según sus responsabilidades funcionales asociadas. En este sentido, para el diseño de los modelos de simulación vinculados al *Balanceador de Carga* y la *Cola de Mensajes* se establecieron un conjunto de lineamientos básicos que definen su comportamiento común como *componente de aplicación definido*. Luego, en casa caso, se incorporaron los lineamientos propios del comportamiento de cada componente.

En este contexto, se formulan los modelos de simulación de los *componentes de aplicación definidos* teniendo en cuenta que:

- *Conocen el componente de aplicación no definido que se encuentran gestionando*: Sin esta información, es imposible que un componente de aplicación definido pueda gestionar de forma eficiente un conjunto de réplicas de *componentes de aplicación no definidos*.
- *Mantienen una estructura de comunicación bidireccional con las réplicas que gestionan*: Esta propiedad se observa en la composición del diseño arquitectónico por medio de la definición de una conexión bidireccional específica que establece la existencia de un intercambio de información entre ambos elementos. Dicho intercambio refiere a: *i)* las solicitudes de usuario que son liberadas desde el *componente de aplicación definido* hacia una réplica

específica del *componente de aplicación no definido*, y *ii)* la información de trabajo que indica a un *componente de aplicación definido* que una réplica de un *componente de aplicación no definido* ha finalizado su trabajo y se encuentra a la espera de nuevas solicitudes de procesamiento.

- *La probabilidad de que se presenten defectos y/o fallas en este tipo de componentes es despreciable (aproximadamente nula)*: Esto se debe a que este tipo de componentes arquitectónicos corresponden a elementos que, por ser altamente difundidos y utilizados, han mejorado su implementación conforme la cantidad de aplicaciones en los que se los ha utilizado. Luego, no se modelan estados de defecto ni estados de fallas.
- *Pueden ser desalojados de los recursos de tecnología de la información si se da de baja el servicio y/o aplicación web vinculado*: Por lo tanto, se modela la recepción de eventos *out-of-use* junto con la evolución del modelo al estado de *inactive* como consecuencia del desalojo producido.

Dado que los modelos de simulación de los *componentes de aplicación definidos* deben gestionar los modelos de simulación de los *componentes de aplicación no definidos*, los eventos a intercambiar entre ambos componentes deben ser del mismo tipo. Luego, en líneas generales, el diseño de los modelos de simulación de los *componentes de aplicación definidos* debe:

- Recibir eventos de entrada *request* y *out-of-use*.
- Enviar eventos de salida *request* y *component-state*.

Considerando que el objetivo del modelo de simulación final es el relevamiento de las métricas indicadas en la Tabla 10.4 y que no se mantienen fases *fault* / *failure* en este tipo de componentes arquitectónicos, se establece que los *componentes de aplicación definidos* empiezan en estado *(running, active, ok)* y sólo pueden evolucionar hacia el estado *expired*. Luego, el evento de salida *component-state* se producirá únicamente cuando el componente evolucione desde la etapa *waiting* hacia *inactive*.

Descripción del Modelo de Simulación "MessageQueue"

El componente *Cola de Mensajes* (*Message Queue*) gestiona la distribución de mensajes entre múltiples réplicas de un mismo *componente de aplicación no definido*. Para esto, el modelo de simulación diseñado utiliza dos parámetros: *i) applicationComponent* que representa el tipo de *componente de aplicación no definido* que debe gestionar, y *ii) replicas* que indica el conjunto de instancias del *applicationComponent* que se encuentra vinculadas a la cola en un instante de tiempo dado.

Al iniciar su ejecución, el componente queda a la espera del arribo de solicitudes de usuario que deben ser procesadas por las *replicas* del *applicationComponent* (etapa *waiting*). Cuando ingresa un evento externo, pueden darse cuatro casos, a saber:

- El evento que ingresa corresponde a un evento *out-of-use*, lo que indica que el componente debe evolucionar hacia la etapa *inactive* como consecuencia de que la aplicación web ha quedado en desuso.
- El evento que ingresa corresponde a un evento *request* y existe una réplica del *applicationComponent* que puede ejecutar su procesamiento inmediato, por lo que el componente debe reenviar la solicitud de usuario a dicha réplica y registrar que la misma se encuentra en estado de procesamiento.
- El evento que ingresa corresponde a un evento *request* pero no existen réplicas del *applicationComponent* que se encuentren libres para la ejecución de su procesamiento inmediato, por lo que el componente debe colocar la solicitud de usuario en una cola de mensajes pendientes por despachar y regresar a *waiting*.
- El evento que ingresa indica que existe alguna réplica que ha finalizado su procesamiento y se encuentra disponible para que se le asignen nuevos trabajos (solicitudes de usuario). Dado que la última acción de los *componentes de aplicación no definidos* es el envío del *request* que acaban de procesar, el

evento que arriba al *MessageQueue* para indicar que existe una réplica que se encuentra en fase *waiting* corresponde a un evento del tipo *request*. En este contexto, la llegada de este evento depende del estado de la cola: *i)* si existen elementos pendientes de procesamiento, se debe proceder a enviar la primera solicitud de usuario que se encuentre en la cola a la réplica del procesador que ha informado que se encuentra libre de trabajo (y marcar que dicha réplica ya no se encuentra disponible); en cambio *ii)* si no existen elementos pendientes de procesamiento, se debe registrar la liberación de la réplica y el modelo debe regresar a la etapa *waiting*.

La Figura 10.7 presenta una representación gráfica de la definición del modelo de simulación asociado al componente arquitectónico definido *Message Queue*. Al igual que en la Figura 10.5, se esquematizan las posibles transiciones con un alto nivel de abstracción entre las distintas etapas de trabajo a fin de indicar la dinámica modelada. Tales etapas quedan definidas como fases -*phase*- del modelo de simulación asociado.

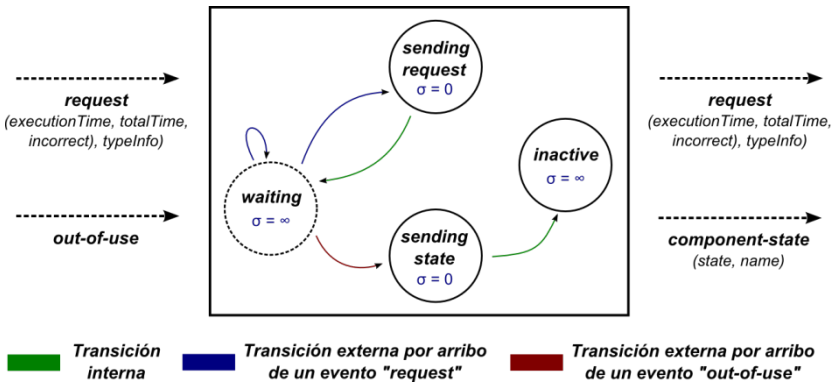


Figura 10.7. Esquema de transiciones, entradas y salidas de "MessageQueue".

Como una primera aproximación, el conjunto de posibles estados internos del modelo esencial RDEVS queda definido de acuerdo a una fase, el tiempo que el modelo debe permanecer en dicho estado y la cola de solicitudes pendientes de

procesamiento: *(phase, σ, queue)*. Dado que los valores indicados para *σ* corresponden a estados pasivos y transitorios, a fin de que el modelo determine la cantidad de instantes de tiempo que una solicitud ha sido retenida en este componente (con el objetivo de actualizar la propiedad *totalTime* asociada al *request* que debe despachar para procesamiento) se incorpora el atributo de tiempo de simulación a la definición del estado. Luego, el estado parcial queda definido como *(phase, σ, queue, simulationTime)*.

Para determinar que un *request* debe ser despachado para su procesamiento el componente debe conocer, en un momento dado, el conjunto de réplicas que se encuentran libres. Al mismo tiempo, al recibir un *request* debe determinar si corresponde a: *i)* un nuevo pedido de procesamiento, o *ii)* una confirmación de que alguna réplica ha finalizado su trabajo. Teniendo en cuenta que los eventos *request* no llevan (ni deben llevar) información asociada a los componentes específicos por los cuales ha sido procesado (es decir, no indica la réplica sino el tipo de componente), esta información debe mantenerla internamente quien despacha las solicitudes. De acuerdo con estos requerimientos, el estado del modelo queda definido como *(phase, σ, queue, simulationTime, freeReplicas, requestsInProcess)*.

Como ya se ha enunciado con anterioridad, el modelo inicia en fase *waiting* con los conjuntos *queue* y *requestsInProcess* vacíos y todos los elementos de *replicas* en *freeReplicas*. Si arriba un evento *out-of-use*, debe generar una transición hacia la fase *sending state* desechando toda la información asociada a su operatoria. Desde este estado transitorio, el modelo enviará un evento *component-state* informando que ha entrado en estado *expired* y, luego, ejecutará una transición hacia el estado pasivo asociado a la fase *inactive*.

Sin embargo, si estando en fase *waiting* arriba un evento *request*, el modelo debe examinar el resto de los componentes del estado para determinar su evolución. En primera instancia, debe determinar si el *id* del *request* que acaba de arribar se encuentra incluido en *requestsInProcess*. De ser así, la solicitud recibida ya había sido

enviada a una réplica y ha finalizado su procesamiento. En este caso, debe retirarse este elemento de la lista *requestsInProcess* e incorporar la réplica que tenía asociada a la lista de *freeReplicas*. Además, debe chequearse el estado de *queue* de la siguiente manera: *i)* si existen elementos pendientes de procesamiento se debe evolucionar hacia el estado transitorio asociado a la fase *sending request*; *ii)* si no existen elementos pendientes de procesamiento, se debe regresar al estado pasivo asociado a la fase *waiting*.

En caso de que el *id* de *request* no se encuentre incluido en *requestsInProcess*, debe chequearse que la solicitud debe ser enviada a un *applicationComponent*[14] y, en caso afirmativo[15], debe anexarse como nuevo elemento de *queue* (que actúa de acuerdo a un comportamiento First-In-First-Out). Luego, debe evaluarse *freeReplicas*. Si este elemento corresponde al conjunto vacío, quiere decir que no existen componentes disponibles para atender esta solicitud. Luego, se regresa al estado pasivo asociado a la fase *waiting*. En cambio, si *freeReplicas* ≠ ø se debe evolucionar hacia la fase *sending request*.

Como consecuencia de que el modelo alcance un estado con fase *sending request*, se despacha una nueva solicitud para procesamiento (la primera de *queue*) removiendo la réplica asignada del conjunto *freeReplicas* e incorporando un nuevo elemento en la lista *requestsInProcess*. En este contexto, luego del envío de la solicitud, el estado del modelo evoluciona a un estado pasivo con fase *waiting*.

Modelo Esencial "MessageQueue"

El modelo esencial RDEVS del componente de aplicación definido *Message Queue* queda definido por la estructura

[14] Esto se debe a que, por la forma en la que son estructuradas las arquitecturas de software, existe la posibilidad de que un *Message Queue* reciba eventos desde distintos *componentes de aplicación no definidos*, por lo que debe chequear que el próximo componente a procesar sea del tipo que se encuentra gestionando.

[15] En otro caso, se descarta el *request* y se regresa el modelo al estado en el que se encontraba previo al arribo del evento.

$$MessageQueue = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau > \quad (10.3)$$

donde:

$X \equiv$ { *request* , *out-of-use* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

$Y \equiv$ { *request* , *component-state* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

component-state = (state, "MessageQueue") donde *state* $\in$ *{ expired }*

$S \equiv$ *phase x σ x queue x simulationTime x freeReplicas x requestsInProcess* con

phase $\in$ *{ waiting, inactive, sending request, sending state }*

{ σ, simulationTime } $\in \mathbb{R}_0^+$

queue es una cola de elementos definidos como *(startTime, request)*

requestsInProcess es una cola de elementos del tipo *request*

freeReplicas es una cola de elementos definidos como *(id, replica)*

$$\delta_{int}(s) = \begin{cases} \delta_{int}(\text{sending state }, \sigma, \text{ queue, simulationTime, freeReplicas, requestsInProcess}) = \\ \quad (\text{inactive }, \infty, \text{ queue, simulationTime, freeReplicas, requestsInProcess}) \\ \\ \delta_{int}(\text{sending request }, \sigma, (\text{startTime, (metricsInfo, (id, type, componentsList)))} \\ \quad \cup \text{ queue, simulationTime, (replica1, replica2, ..., replicaN),} \\ \quad \text{requestsInProcess) =} \\ \quad (\text{waiting }, \infty, \text{ queue, simulationTime, (replica2, ..., replicaN),} \\ \quad \text{requestsInProcess} \cup (\text{id, replica1})) \end{cases}$$

$$\delta_{ext}(s,e,x) = \begin{cases} \delta_{ext}(s,e,\text{out-of-use}) = (\text{sending state}, 0, \emptyset, 0, \emptyset, \emptyset) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \emptyset, \text{simulationTime}, \text{freeReplicas}, \\ (\ldots(i1,r1), (\text{id},\text{replica}), (i3,r3),\ldots)), e, \\ (\text{metricsInfo},(\text{id},\text{type},\text{componentsList}))) = \\ (\text{waiting}, \sigma, \emptyset, \text{simulationTime} + e, \text{freeReplicas} \cup \{\text{replica}\}, \\ (\ldots(i1,r1),(i3,r3),\ldots)) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{queue}, \text{simulationTime}, \text{freeReplicas}, \\ (\ldots(i1,r1), (\text{id},\text{replica}), (i3,r3),\ldots)), e, \\ (\text{metricsInfo},(\text{id},\text{type},\text{componentsList}))) = \\ (\text{sending request}, 0, \text{queue}, \text{simulationTime} + e, \text{freeReplicas} \cup \{\text{replica}\}, \\ (\ldots(i1,r1),(i3,r3),\ldots)) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{queue}, \text{simulationTime}, \emptyset, \text{requestsInProcess}), e, \\ (\text{metricsInfo},(\text{id},\text{type},(\text{component}, \text{applicationComponent}, \text{componentsList}))) = \\ (\text{waiting}, \infty, \text{queue} \cup (\text{simulationTime} + e, \\ (\text{metricsInfo},(\text{id},\text{type},(\text{applicationComponent}, \text{componentsList})))), \\ \text{simulationTime} + e, \emptyset, \text{requestsInProcess}) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{queue}, \text{simulationTime}, \text{freeReplicas}, \\ \text{requestsInProcess}), e, (\text{metricsInfo}, \\ (\text{id},\text{type},(\text{component}, \text{applicationComponent}, \text{componentsList})))) = \\ (\text{sending request}, 0, \text{queue} \cup (\text{simulationTime} + e, \\ (\text{metricsInfo},(\text{id},\text{type},(\text{applicationComponent}, \text{componentsList})))), \\ \text{simulationTime} + e, \text{freeReplicas}, \text{requestsInProcess}) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{queue}, \text{simulationTime}, \text{freeReplicas}, \text{requestsInProcess}), \\ e, (\text{metricsInfo},(\text{id},\text{type},\text{componentsList}))) = \\ (\text{waiting}, \sigma - e, \text{queue}, \text{simulationTime} + e, \text{freeReplicas}, \text{requestsInProcess}) \\ \\ \delta_{ext}(\text{phase},\sigma,\text{queue},\text{simulationTime},\text{freeReplicas},\text{requestsInProcess}),e,\text{request}) = \\ (\text{phase},\sigma - e, \text{queue}, \text{simulationTime}+e, \text{freeReplicas}, \text{requestsInProcess}) \end{cases}$$

$$\lambda(s) = \begin{cases} \lambda(\text{sending state}, \sigma, \text{queue}, \text{simulationTime}, \text{freeReplicas}, \text{requestsInProcess}) = \\ (\text{expired}, \text{"MessageQueue"}) \\ \\ \lambda(\text{sending request}, \sigma, (\text{startTime}, \{(\text{executionTime}, \text{totalTime}, \text{incorrect}), \\ (\text{id},\text{type},\text{componentsList})\}) \cup \text{queue}, \text{simulationTime}, \\ (\text{replica1}, \text{replica2}, \ldots, \text{replicaN}), \text{requestsInProcess}) = \\ ((\text{executionTime}, \text{totalTime} + \text{simulationTime}, \text{incorrect}), \\ (\text{id},\text{type},\text{componentsList})) \end{cases}$$

$\tau(s) = \tau(\ phase, \sigma, queue, simulationTime, freeReplicas, requestsInProcess) = \sigma$

Descripción del Modelo de Simulación "LoadBalancer"

El componente *Balanceador de Carga* (*Load Balancer*) se antepone como primer elemento de una arquitectura de software web con el objetivo de distribuir las solicitudes de usuario entre múltiples réplicas de un *componente de aplicación no definido* (a fin de garantizar su procesamiento inmediato inicial). Existen múltiples algoritmos disponibles para realizar esta distribución. Si en un momento dado no existen réplicas libres para el procesamiento de las solicitudes, este *componente de aplicación definido* interactúa con los *componentes de administración* a fin de indicar que se requiere la creación de nuevas réplicas. Tales componentes, a su vez, interactúan con las ofertas de procesamiento ubicadas en la capa de infraestructura para determinar si aceptan o deniegan este pedido. Luego, en la mayoría de los casos, el balanceador de carga no almacena solicitudes a la espera de que se liberen réplicas para su procesamiento.

Siguiendo este comportamiento junto con los lineamientos básicos establecidos para los *componentes de aplicación definidos* y, considerando que la capa de infraestructura no es incorporada como parte del modelo de simulación, el modelo *LoadBalancer* ha sido definido de la siguiente manera:

- Utiliza como parámetros la información *applicationComponent* y *replicas* (la cual queda definida de la misma forma que los parámetros del modelo de simulación *MessageQueue*).
- Implementa un algoritmo de distribución *round robin* basado en la disponibilidad de procesamiento de las réplicas vinculadas al balanceador.
- No debe manejar el tiempo de simulación ya que la asignación a las réplicas se produce de forma instantánea.
- En el caso de que una *request* no pueda ser entregada para procesamiento inmediato, la descarta.

Luego, al iniciar su ejecución el componente queda a la espera del arribo de solicitudes de usuario que deben ser procesadas por las *replicas* del *applicationComponent* (etapa *waiting*). Ante el arribo de un evento externo, pueden darse cuatro situaciones, a saber:

- Si el evento que ingresa es un evento *out-of-use*, se lo trata de la misma forma que en el modelo *MessageQueue*.
- Si el evento que ingresa es un evento *request* que informa que una réplica ha quedado libre de trabajo, se actualiza la información de distribución y se regresa a la etapa *waiting*.
- Si el evento que ingresa es un evento *request* y existe una réplica del *applicationComponent* que puede ejecutar su procesamiento inmediato, se ingresa a la etapa *sending request*.
- Si el evento que ingresa es un evento *request* y no existe una réplica del *applicationComponent* que puede ejecutar su procesamiento inmediato, se descarta el pedido y se regresa a la etapa *waiting*.

La representación gráfica de la definición del modelo *LoadBalancer* es similar a la propuesta para *MessageQueue* (Figura 10.7). Sin embargo, ambos modelos se diferencian en la forma en la cual definen su estado y las transiciones entre los mismos.

El estado del modelo *LoadBalancer* queda definido como *(phase, σ, lastReplicaUsed, replicasState, actualRequest)*. Su estado inicial corresponde a un estado en fase *waiting*. Si arriba un evento *out-of-use*, el modelo evoluciona a un estado transitorio asociado a la fase *sending state* desde el cual genera un evento *component-state* que informa que su nuevo estado es *expired* (previo a ejecutar una transición hacia el estado pasivo asociado a la fase *inactive*). Por el contrario, si arriba un evento *request* debe determinar si corresponde a una nueva solicitud de usuario o al informe de una réplica que ha quedado disponible para procesamiento. Para esto, el modelo examina la lista *replicasState* (en la cual almacena los identificadores de la última solicitud que ha

sido asignada a cada uno de los elementos definidos en *replicas*). Si se encuentra que el identificador del *request* pertenece a *replicasState*, el modelo actualiza la información de las réplicas disponibles y regresa a un estado pasivo con fase *waiting*. En caso contrario, el modelo examina si existen réplicas disponibles por medio de la inspección de *replicasState*. Si todas las réplicas se encuentran ocupadas, se descarta el *request* y se regresa a la etapa *waiting*. En otro caso, el modelo determina la réplica a utilizar para procesar el *request* utilizando *lastReplicaUsed* como valor inicial de la iteración que recorre *replicasState*. Cuando encuentra un elemento de *replicasState* que no posee un identificador de solicitud asociado, le asigna el identificador de *actualRequest* y actualiza *lastReplicaUsed* con su información. Posteriormente, evoluciona a un estado transitorio con *actualRequest* = *request* y fase *sending request*.

Desde dicho estado, el modelo envía el *actualRequest* a la réplica indicada en *lastReplicaUsed* y regresa al estado pasivo con fase *waiting* y *actualRequest* = *ø.*

Modelo Esencial "Load Balancer"

El modelo esencial RDEVS del componente de aplicación definido *Load Balancer* queda definido por la estructura

$$LoadBalancer = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau > \qquad (10.4)$$

donde:

$X \equiv$ { *request* , *out-of-use* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

$Y \equiv$ { *request* , *component-state* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

component-state = (state, "LoadBalancer") donde *state ∈ { expired }*

S ≡ phase x σ x lastReplicaUsed x replicasState x actualRequest con

phase ∈ { waiting, inactive, sending request, sending state }

$\sigma \in \mathbb{R}_0^+$

lastReplicaUsed es un valor entero que indica la última posición de *replicasState* que ha sido utilizada para procesar una solicitud de usuario

replicasState es un arreglo de enteros que almacena en la posición de cada réplica el identificador de la solicitud que se encuentra siendo procesada en ese instante de tiempo o el valor *free*

actualRequest es un elemento del tipo *request*

$$\delta_{int}(s) = \begin{cases} \delta_{int}(\text{sending state}, \sigma, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) = \\ \quad (\text{inactive}, \infty, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) \\ \delta_{int}(\text{sending request}, \sigma, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) = \\ \quad (\text{waiting}, \infty, \text{lastReplicaUsed}, \text{replicasState}, \emptyset) \end{cases}$$

$$\delta_{ext}(s,e,x) = \begin{cases} \delta_{ext}(s,e,\text{out-of-use}) = (\text{sending state}, 0, -, \emptyset, \emptyset) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{lastReplicaUsed}, (\ldots,id1, id, id3,\ldots), \text{actualRequest}),e, \\ (\text{metricsInfo},(id,\text{type},\text{componentsList}))) = \\ (\text{waiting}, \infty, \text{lastReplicaUsed}, (\ldots,id1, \text{free}, id3,\ldots), \text{actualRequest}) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{lastReplicaUsed}, (id1,id2,\ldots, idN), \\ \text{actualRequest}),e,\text{request}) = \\ (\text{waiting}, \infty, \text{lastReplicaUsed}, (id1,id2,\ldots, idN), \text{actualRequest}) \\ \\ \delta_{ext}((\text{waiting}, \sigma, \text{lastReplicaUsed}, (id1\ idlastReplicaUsed,\ldots, \\ \text{freereplica},\ldots,idN,\ldots), \text{actualRequest}), \\ e,(\text{metricsInfo},(id,\text{type},\text{componentsList})))) = \\ (\text{sending request}, 0, \text{replica}, \\ (id1, idlastReplicaUsed,\ldots, id,\ldots, idN), \text{request}) \\ \\ \delta_{ext}((\text{phase} ,\sigma, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}),e,\text{request}) = \\ (\text{phase},\sigma\text{-}e, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) \end{cases}$$

$$\lambda(s) = \begin{cases} \lambda(\text{sending state}, \sigma, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) = \\ (\text{expired}, \text{"LoadBalancer"}) \\ \\ \lambda(\text{sending request} ,\sigma, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) = \\ \text{actualRequest} \end{cases}$$

$$\tau(s) = \tau(\text{phase}, \sigma, \text{lastReplicaUsed}, \text{replicasState}, \text{actualRequest}) = \sigma$$

10.4.3 Especificación de una Arquitectura Web

Mapeo de la Descripción Arquitectónica al Modelo de Simulación

La Figura 10.8 presenta una arquitectura de software web denominada *Simple* cuya definición involucra dos *componentes de aplicación no definidos* (*Example* y *Data*) y dos componentes de aplicación definidos (uno del tipo *Load Balancer* y otro del tipo *Message Queue*). Al igual que en el caso de la Figura 10.4, la representación ha sido generada utilizando la *herramienta de modelado* implementada.

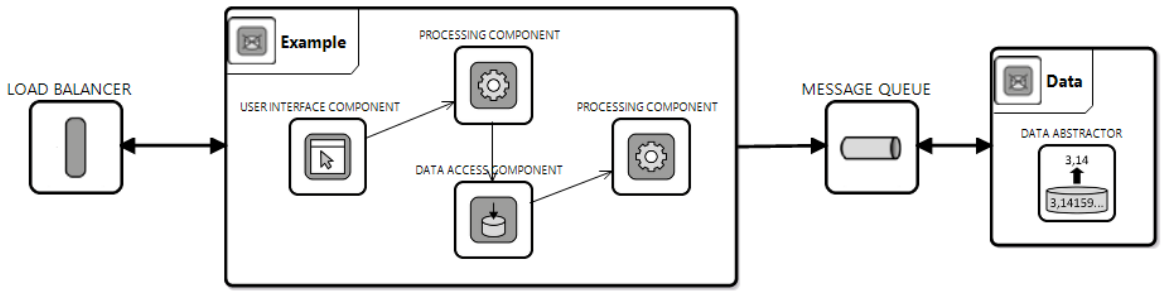


Figura 10.8. Diseño de una arquitectura de software web.

Dada la inexistencia de la capa de infraestructura (la cual gestiona la creación de réplicas de componentes), la especificación de un modelo de simulación asociado a una arquitectura dada requiere de la definición de una estructura de ejecución explícita que contemple la existencia de múltiples réplicas para cada *componente de aplicación no definido* (no permitiéndose la creación de réplicas de los *componentes de aplicación definidos*[16]). En este sentido, la Figura 10.9 esquematiza una estructura de ejecución para la arquitectura *Simple*, la cual queda definida en base a tres réplicas del componente *Example* y dos réplicas del componente *Data*. Esta estructura puede ser vista como un modelo de red RDEVS sobre el cual se determina el camino de procesamiento que deben seguir las distintas solicitudes de usuario a fin de relevar un conjunto de métricas definidas.

[16] Los *componentes de aplicación definidos* tienen por objetivo gestionar réplicas de *componentes de aplicación no definidos*. Luego, un conjunto de réplicas no puede ser gestionado por más de un elemento.

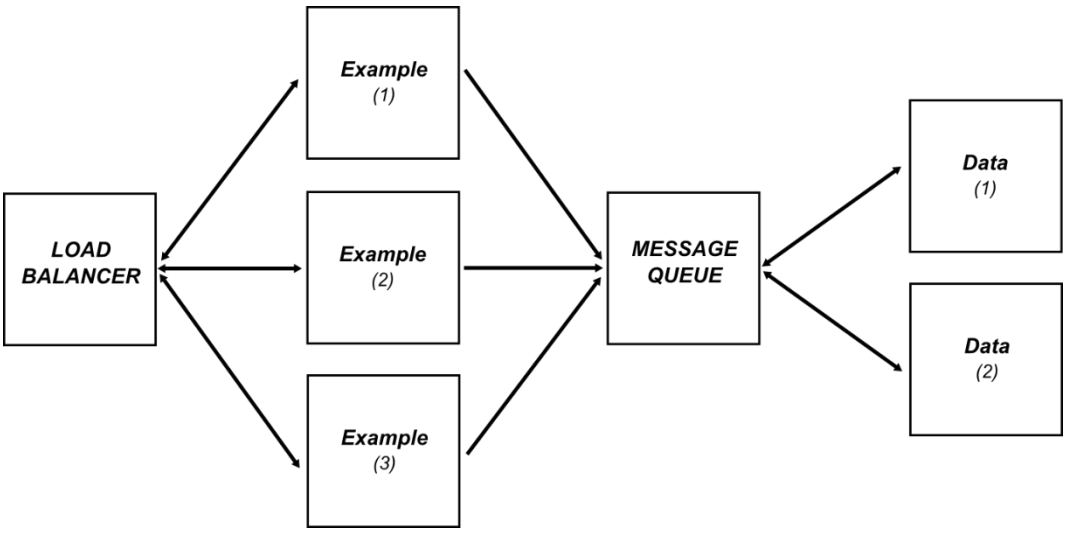


Figura 10.9. Posible estructura de ejecución de la arquitectura "Simple".

En este punto de diseño, los *componentes de aplicación no definidos* y los *componentes de aplicación definidos* han sido especificados como modelos esenciales RDEVS (es decir, componentes a ser utilizados en múltiples entidades de ruteo como parte de un proceso de mayor nivel). Si el proceso de ruteo sobre el cual se deben gestionar los eventos (es decir, el modelo de red) queda definido en base a la estructura de ejecución de la arquitectura de software web, se tiene que:

- Las réplicas de los *componentes de aplicación no definidos* deben ser diseñadas como modelos de ruteo RDEVS.
- Las diferentes instancias de los *componentes de aplicación definidos* deben ser diseñadas como modelos de ruteo RDEVS.

Como ya se ha establecido, la definición de una entidad de ruteo requiere la especificación de la información a utilizar para: *i)* aceptar los eventos que ingresan al modelo, y *ii)* direccionar los eventos que salen del modelo. Una vez que los componentes ya han sido definidos, la formulación de entidades implica únicamente la especificación de dicha información para cada caso. Esto se debe a que tanto el

comportamiento propio del componente como la gestión de los eventos con identificación, se encuentran definidos como parte de la formalización del modelo de ruteo.

Si se considera que existen distintos tipos de componentes arquitectónicos (definidos y no definidos), se debe diseñar una entidad de ruteo para cada réplica o instancia a ser incorporada en el modelo de red. Cada entidad aplicará una estrategia diferente para la definición de los destinatarios (es decir, tendrá una función de ruteo diferente). Esto se debe a que la decisión del conjunto de posibles destinos para un evento dado, se toma en función del estado actual definido en el modelo. Dado que el modelo de *componente de aplicación no definido* difiere de los modelos de *componentes de aplicación definidos*, los modelos de ruteo propuestos en cada caso utilizarán función de ruteo específicas.

Luego, el modelo de red *Simple* queda formado por siete modelos de ruteo. Cada modelo incluye, de forma individual, un identificador único como parte de la estructura objetivo del proceso de ruteo (Tabla 10.9).

Tabla 10.9. Modelos de ruteo que componen el modelo de red "Simple".

ENTIDAD	NOMBRE DEL MODELO	IDENTIFICADOR
Load Balancer	*LoadBalancerInstance1*	1
Message Queue	*MessageQueueInstance1*	2
Example (1)	*ExampleInstance1*	30
Example (2)	*ExampleInstance2*	31
Example (3)	*ExampleInstance3*	32
Data (1)	*DataInstance1*	40
Data (2)	*DataInstance2*	41

Manejo de Réplicas de un Componente de Aplicación No Definido (Modelo de Ruteo RDEVS)

Tal como se ha establecido con anterioridad, las réplicas de los componentes arquitectónicos del tipo *componente de aplicación no definido* quedan especificadas como modelos de ruteo RDEVS. Luego, teniendo en cuenta que el estado del modelo réplica quedará definido a semejanza del estado del modelo esencial diseñado para representar el componente arquitectónico original, la réplica (en todo momento) conoce la solicitud de usuario que se encuentra bajo procesamiento (esto es, el campo *actualRequest* que lleva en su interior la información vinculada al camino que debe seguir la solicitud dentro de los componentes arquitectónicos que componen el proceso de ruteo).

En este contexto, tomando como ejemplo el componente *Example* (definido en la Ecuación 10.2), la formalización de los modelos de ruteo denominados *ExampleInstance1*, *ExampleInstance2* y *ExampleInstance3* se especifica de acuerdo a las siguientes características:

- El modelo *Example* es utilizado como componente base de la definición de las diferentes instancias.
- El conjunto de entidades remitentes habilitadas incluye únicamente el modelo de ruteo asociado al componente *Load Balancer* (ya que es el único elemento de la estructura de ejecución que tiene conexiones explícitas con el componente arquitectónico *Example*).
- La función de ruteo establece que:

  i) Los eventos de salida del tipo *component-state* deben ser enviados hacia el exterior de la red (dado que son generados con el objetivo de relevar métricas de estado del sistema).

  ii) Los eventos *request* que poseen una *componentsList* vacía deben ser enviados hacia el exterior de la red (ya que corresponden a solicitudes

de usuario que han sido procesadas por todos los componentes arquitectónicos indicados como parte de su camino).

iii) Los eventos *request* que poseen una *componentsList* no vacía deben ser enviados hacia el próximo destino de procesamiento (ya que aún deben ser procesados por otros componentes arquitectónicos).

iv) Los destinos asociados a los eventos *request* que poseen una *componentsList* no vacía corresponden a los modelos de ruteo asociados al componente *Message Queue* y *Load Balancer* (ya que son los únicos elementos de la estructura de ejecución que tienen conexiones explicitas con el componente arquitectónico *Example*).

Luego, la formalización de los modelos de ruteo asociados a las distintas réplicas de *Example* varía únicamente el valor del identificador *u* asociado a cada entidad. A modo de ejemplo de dicha formalización, el modelo de ruteo RDEVS de la réplica *ExampleInstance1* queda definido por la estructura

$$ExampleInstance1 = < (u, W, \delta_r) , Example, Instance > \quad (10.5)$$

donde:

$u = 30,$

$W = \{1\},$

$$\delta_r(s) = \begin{cases} \delta_r(\text{sending request, } \sigma, ((\text{metricsInfo}),(\text{ id, type, (component1, component2,...componentN}))), \text{nextFunctionState}) = \{ 2 \} \\ \delta_r(\text{sending request, } \sigma, ((\text{metricsInfo}),(\text{ id, type,}(\emptyset))), \text{nextFunctionState}) = \emptyset \\ \delta_r(\text{sending information, } \sigma, \text{actualRequest, nextFunctionState}) = \{ 1 \} \\ \delta_r(\text{sending fault, } \sigma, \text{actualRequest, nextFunctionState}) = \\ \delta_r(\text{sending failure, } \sigma, \text{actualRequest, nextFunctionState}) = \\ \delta_r(\text{processing PC2, } \sigma, \text{actualRequest, nextFunctionState}) = \\ \delta_r(\text{processing with faults PC2, } \sigma, \text{actualRequest, nextFunctionState }) = \\ \delta_r(\text{phase, } \sigma, \text{actualRequest, ok}) = \\ \delta_r(\text{sending state, } \sigma, \text{actualRequest, nextFunctionState}) = \\ \delta_r(\text{phase, } \sigma, \text{actualRequest, fault}) = \\ \delta_r(\text{phase, } \sigma, \text{actualRequest, failure}) = \emptyset \end{cases}$$

Example es el modelo esencial definido en el Ecuación 10.2,

Instance es el modelo que se ejecuta durante el proceso de ruteo (cuya definición sigue los lineamientos establecidos para la *definición de modelo de ruteo RDEVS*).

Manejo de Instancias de un Componente de Aplicación Definido (Modelo de Ruteo RDEVS)

Para la definición de instancias de los modelos *MessageQueue* y *LoadBalancer* se utilizan distintas estrategias de ruteo. Esto se debe a que cada caso mantiene una definición de estado diferente, por lo que es necesario detallar funciones de ruteo específicas aplicables a cada situación. Sin embargo, en ambos casos los eventos del tipo *component-state* son enviados hacia el exterior de la red (ya que únicamente se generan para informar que el modelo ha caído en estado inactivo).

En este punto, es importante destacar que los modelos esenciales envían eventos *request* hacia una réplica específica. Dicha réplica se encuentra indicada como parte del estado del modelo (no como parte del evento), por lo que la determinación del camino a seguir debe realizarse haciendo uso del estado actual. Luego, para determinar el modelo de simulación que representa la réplica a la cual debe enviarse un evento

request, se utiliza el estado del modelo junto con parámetros auxiliares que han sido definidos para cada caso.

En el caso del componente *LoadBalancer*, la formalización de los modelos de ruteo que representan instancias de este tipo de componente queda definida de forma similar al modelo *LoadBalancerInstance1*. En esta formalización se incluye el parámetro *replicasId* que corresponde a un arreglo de identificadores que mantiene una correspondencia uno a uno con el parámetro *replicas* del modelo esencial asociado. Luego, el modelo *LoadBalancerInstance1* queda formalizado por la estructura

$$LoadBalancerInstance1 = < (u, W, \delta_r) , LoadBalancer, Instance > \quad (10.6)$$

donde:

$u = 1$,

$W = \{ 30, 31, 32 \}$,

$$\delta_r(s) = \begin{cases} \delta_r(sending\ state, \sigma, lastReplicaUsed, replicasState, actualRequest) = \emptyset \\ \delta_r(sending\ request\ ,\sigma, lastReplicaUsed, replicasState, actualRequest) = \\ \quad \{ replicasId[lastReplicaUsed] \} \quad \text{con } replicasId = \{ 30, 31, 32 \} \end{cases}$$

LoadBalancer es el modelo esencial definido en el Ecuación 10.4,

Instance es el modelo que se ejecuta durante el proceso de ruteo (cuya definición sigue los lineamientos establecidos para la *definición de modelo de ruteo RDEVS*).

Por su parte, el modelo *MessageQueueInstance1* define a modo de ejemplo la formalización de un modelo de ruteo basado en el modelo esencial *MessageQueue*. Dicho modelo también utiliza el parámetro *replicasId* (el cual queda definido de la

misma forma que el parámetro utilizado en las instancias de *LoadBalancer*). Luego, el modelo de ruteo queda formalizado por la estructura

$$MessageQueueInstance1 = < (u, W, \delta_r) , MessageQueue, Instance > \quad (10.7)$$

donde:

$u = 2$,

$W = \{ 30, 31, 32, 40, 41 \}$,

$$\delta_r(s) = \begin{cases} \delta_r(sending\ state\ ,\sigma,\ queue,\ simulationTime,\ freeReplicas,\ requestsInProcess) = \emptyset \\ \delta_r(sending\ request\ ,\sigma,(startTime,\ ((executionTime,totalTime,incorrect),\ (id,type,componentsList)))\ \cup queue,\ simulationTime,\ freeReplicas,\ requestsInProcess) = \\ \{replicasId[freeReplicas.getFirstFree()]\} \quad \text{con } replicasId = \{ 40, 41 \} \end{cases}$$

MessageQueue es el modelo esencial definido en el Ecuación 10.3,

Instance es el modelo que se ejecuta durante el proceso de ruteo (cuya definición sigue los lineamientos establecidos para la *definición de modelo de ruteo RDEVS*).

Modelo de Red RDEVS

Una vez definidos los modelos de ruteo que componen la estructura de ejecución propuesta en la Figura 10.9, se define el modelo de red *Simple*. Asumiendo que los modelos *Data*, *DataInstance1* y *DataInstance2* han sido definidos siguiendo los lineamientos aplicados para *Example* y sus réplicas, el modelo *Simple* queda definido por la estructura

$$Simple = < X, Y, D, \{ R_d \}, \{ I_d \}, \{ Z_{i,d} \}, T_{in}, T_{out}, Select > \quad (10.8)$$

donde:

$X \equiv$ { *request* , *out-of-use* } con

request = (metricsInfo , typeInfo) = ((0, 0, false), (id, type, componentsList))

$Y \equiv$ { *request* , *component-state* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

component-state = (state, componentName) donde *state* $\in$ *{ (running, active, ok), (running, active, fault), (running, failure), expired }*

$D \equiv$ *{ model1, model30, model31, model32, model2, model40, model41 }*

R_{model1} = *LoadBalancerInstance1*, $R_{model30}$ = *ExampleInstance1*,

$R_{model31}$ = *ExampleInstance2*, $R_{model32}$ = *ExampleInstance3*,

R_{model2} = *MessageQueueInstance1*, $R_{model40}$ = *DataInstance1*, $R_{model41}$ = *DataInstance2*,

I_{model1} = *{ model30, model31, model32, model2, model40, model41, Simple }*

$I_{model30}$ = *{ model1, model31, model32, model2, model40, model41, Simple }*

$I_{model31}$ = *{ model1, model30, model32, model2, model40, model41, Simple }*

$I_{model32}$ = *{ model1, model30, model31, model2, model40, model41, Simple }*

I_{model2} = *{ model1, model30, model31, model32, model40, model41, Simple }*

$I_{model40}$ = *{ model1, model30, model31, model32, model2, model41, Simple }*

$I_{model41}$ = { *model1, model30, model31, model32, model2, model40, Simple* }

$$Z_{i,d} = \begin{cases} T_{in} & \forall\, i = Simple \\ T_{out} & \forall\, d = Simple \\ request & en\ otro\ caso \end{cases}$$

$$T_{in} = \begin{cases} T_{in}(request) = (request, 0, \{1\}) \\ T_{in}(out\text{-}of\text{-}use) = (out\text{-}of\text{-}use, 0, \{1, 30, 31, 32, 2, 40, 41\}) \end{cases}$$

$$T_{out} = \begin{cases} T_{out}(request, \emptyset, \emptyset) = (request) \\ T_{out}(component\text{-}state, \emptyset, \emptyset) = (component\text{-}state) \end{cases}$$

Como puede observase, todos los modelos se influencian entre sí a fin de que el intercambio de eventos se gestione únicamente por las decisiones vinculadas a la definición del proceso de ruteo.

10.5 Especificación del Modelo de Simulación MRM

El modelo MRM fue diseñado como un modelo DEVS atómico que, ante la llegada de un evento *request* o *component-state*, efectúa el cálculo de las métricas vinculadas a dicha información. Luego de esta acción, registra en *dos archivos de salida*[17] el relevamiento actual de todas las métricas que están siendo calculadas (Tabla 10.4).

[17] Se utilizan dos archivos a fin de diferenciar las métricas vinculadas al tiempo de ejecución (es decir, aquellas cuya evolución se relaciona con el estado del sistema en un contexto temporal) y las métricas referidas al tratamiento de las solicitudes (en las cuales, el registro de lo sucedido no se vincula con el tiempo de ejecución).

En este sentido, el estado del modelo queda definido como *(phase, σ, simulationTime, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, type*[18] *)*, donde los últimos diez elementos corresponden a las métricas relevadas. Luego, el estado inicial es *(waiting, ∞, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -)*.

Cuando arriba un evento de entrada, el modelo evoluciona hacia un nuevo estado en el que actualiza *simulationTime* (a fin de mantener registrado el tiempo de simulación actual). Si el evento de entrada corresponde a un *component-state* que informa un estado de desalojo (*expired*), el estado resultante será un estado pasivo con fase *stop* (ya que detecta que la aplicación web ha sido desalojada de su infraestructura y ya no se encuentra activa). En el caso de que el evento *component-state* no corresponda a un estado de desalojo, el modelo evoluciona hacia un nuevo estado con un valor actualizado de *TT*. Además, evalúa si el estado de componente reportado en el evento corresponde a una situación de defecto o falla. En el caso de que el evento reporte un estado defectuoso (*fault*), en el estado resultante se actualizan *FNF* y *TF*. Por su parte, si el estado corresponde a una falla (*failure*), el nuevo estado actualiza *TF* y luego, comprueba si (como resultado de la ocurrencia de esta falla) el servicio ha quedado inactivo[19]. Ante una eventual situación de inactividad en los componentes de aplicación, actualiza el valor de *FT*. En contraposición al arribo de eventos de entrada del tipo *component-state*, cuando arriba un evento *request* el modelo actualiza los valores de *ET, TSIT, IR, TR* y *type* como consecuencia del evento y evoluciona hacia un nuevo estado.

En todos estos casos en los que se actualizan los campos que relevan métricas, el modelo evoluciona hacia un nuevo estado transitorio con fase *savingMetrics*. En dicho

[18] Este campo refiere al tipo de solicitud sobre el cual se han obtenido los valores de ET y TSIT (ya que la métrica queda definida para cada solicitud que se procese).

[19] Para realizar esta comprobación utiliza el parámetro *modelStructure* que define la cantidad de réplicas activas existentes para cada componente de aplicación incluido en la arquitectura. Inicialmente, este parámetro queda configurado con la disposición de réplicas propuestas en la estructura bajo simulación. A medida que los componentes informan estados de falla, se actualiza la cantidad de réplicas activas. Si en algún momento se detecta que existe un componente que no posee réplicas activas, se deduce que el servicio ha quedado inactivo.

estado, el modelo intenta almacenar la información obtenida haciendo uso de los archivos propuestos. Si tiene éxito, regresa a un estado pasivo con fase *waiting*. En caso de que falle la operación de escritura, el modelo evoluciona a un estado pasivo con fase *file error*.

Una vez que el modelo entra en un estado pasivo con fase *stop* o *file error*, no acepta más eventos de entrada para la actualización de los valores de métricas. Como resultado de las transiciones internas, se genera un evento de salida vacío.

10.5.1 Especificación Formal

El modelo atómico MRM queda definido por la estructura

$$MRM = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau > \qquad (10.9)$$

donde:

$X \equiv$ { *request* , *component-state* } con

request = (metricsInfo , typeInfo) = ((executionTime, totalTime, incorrect), (id, type, componentsList))

component-state = (state, componentName) donde *state* $\in$ *{ (running, active, ok), (running, active, fault), (running, failure), expired }*

$Y \equiv \{ \emptyset \}$

$S \equiv$ *phase x σ x simulationTime x TR x IR x FT x TT x FNF x TF x RF x ET x TSIT x type* con

phase $\in$ *{ waiting, saving metrics stop, file error }*

$\{ \sigma, simualtionTime, FT, TT, ET, TSIT \} \in \mathbb{R}_0^+$

$\{ TR, IR, FNF, TF, RF \} \in N_0$

type es el tipo de solicitud que corresponde a los datos ET y TSIT

$$\delta_{int}(s) = \begin{cases} \delta_{int}(\text{saving metrics}, \sigma, \text{simulationTime}, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, \text{type}) = \\ (\text{waiting}, \infty, \text{simulationTime}, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, -) & \text{si savingMetrics()==true} \\ \delta_{int}(\text{saving metrics}, \sigma, \text{simulationTime}, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, \text{type}) = \\ (\text{file error}, \infty, \text{simulationTime}, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, -) & \text{en otro caso} \end{cases}$$

$$\delta_{ext}(s,e,x) = \begin{cases} \delta_{ext}((\text{phase}, \sigma, \text{simulationTime}, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, \text{type}), e, (\text{expired})) = \\ (\text{stop}, \infty, \text{simulationTime}+e, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, -) \\ \delta_{ext}((\text{waiting}, \sigma, \text{simulationTime}, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, \text{type}), e, (\text{running}, \text{active}, \text{fault})) = \\ (\text{saving metrics}, 0, \text{simulationTime}+e, TR, IR, FT, TT, FNF+1, TF+1, RF, ET, TSIT, -) \end{cases}$$

$$\delta_{ext}(s,e,x) = \begin{cases} \delta_{ext}((waiting, \sigma, simulationTime, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT,type),e,(running, failure)) = \\ (saving\ metrics,0,simulationTime+e,TR,IR,FT,TT,FNF,TF+1, RF,ET, TSIT,-) \quad si\ serviceInactive()=false \\ \\ \delta_{ext}((waiting, \sigma, simulationTime, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT,type),e,(running, failure)) = \\ (saving\ metrics,0,simulationTime+e,TR,IR,FT+e,TT,FNF,TF+1, RF,ET,TSIT,-) \quad si\ serviceInactive()=true \\ \\ \delta_{ext}((waiting, \sigma, simulationTime, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT,type),e,((executionTime, totalTime, incorrect), (id, type, componentsList))) = \\ (saving\ metrics,0,simulationTime+e,TR+1,IR+(incorrect==true), FT, TT, FNF, TF, RF,executionTime,totalTime,type) \\ \\ \delta_{ext}((phase, \sigma, simulationTime, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, type),e, x) = \\ (phase, \sigma-e, simulationTime+e, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT, -) \quad en\ otro\ caso \end{cases}$$

$$\lambda(s) = \emptyset$$

$$\tau(s) = \tau(phase, \sigma, simulationTime, TR, IR, FT, TT, FNF, TF, RF, ET, TSIT) = \sigma$$

10.6 Implementación en Java de los Modelos de Simulación

En el *Apéndice C* se presentan las clases Java implementadas para los modelos RDEVS propuestos haciendo uso del framework descripto en el apartado *"Implementación de Modelos RDEVS en Java"*. Específicamente se detallan las implementaciones de:

- El modelo *Example* formalizado en la Ecuación 10.2 como instancia de la clase *EssentialModel.java*.
- El modelo *MessageQueue* formalizado en la Ecuación 10.3 como instancia de la clase *EssentialModel.java*.

- El modelo *LoadBalancer* formalizado en la Ecuación 10.4 como instancia de la clase *EssentialModel.java*.
- Los modelos *ExampleInstance1, ExampleInstance2 y ExampleInstance3* definidos de acuerdo a la formalización propuesta en la Ecuación 10.5 como instancias de la clase *RoutingModel.java*.
- El modelo *LoadBalancerInstance1* formalizado en la Ecuación 10.6 como instancia de la clase *RoutingModel.java*.
- El modelo *MessageQueueInstance1* formalizado en la Ecuación 10.7 como instancia de la clase *RoutingModel.java*.

Además, el apéndice incluye la implementación del modelo MRM (formalizado en la Ecuación 10.9) como modelo atómico DEVS.

El proyecto Java completo de la implementación de los componentes que forman parte del modelo *Simple* se encuentra disponible en *SimpleProject*.

Conclusiones

En este capítulo se han utilizado los modelos de calidad de aplicaciones web con el objetivo de establecer el conjunto de atributos de calidad medibles en tiempo de ejecución y la información mínima requerida para su evaluación a partir del diseño arquitectónico. Los criterios empleados en la elaboración de la clasificación resultante complementan los contenidos presentados en los capítulos previos, brindando un esquema de clasificación aplicable a otros contextos en los cuales sea necesario determinar la capacidad de una característica para ser simulada.

En base a la información obtenida, se han especificado las salidas deseadas como resultado de los modelos de simulación (esto es, el conjunto de métricas directas básicas a relevar). De esta manera, se ha analizado el contexto de calidad web en relación a las arquitecturas de software de aplicaciones web en virtud de diseñar e implementar modelos de simulación RDEVS que faciliten su relevamiento.

En este contexto, dado que el modelo de simulación de cada componente de aplicación no definido debe ser diseñado de acuerdo a su especificación funcional (es decir, en base a los componentes funcionales incluidos en su diseño), se han establecido los lineamientos básicos que dan lugar a dicha transformación. Además, para los componentes de aplicación definidos se ha especificado un modelo propio que representa su comportamiento. Luego, en base a la construcción de una arquitectura de ejemplo, se han formalizado e implementado las instancias de ruteo requeridas para la simulación del proceso de solicitudes de usuario.

Los modelos RDEVS diseñados para la simulación arquitectónica son complementados con un modelo DEVS que facilita la obtención de las métricas requeridas (garantizando su cálculo y almacenamiento). Luego, el resultado es un conjunto de modelos de simulación de eventos discretos que brindan una primera aproximación de la evaluación de la

calidad de aplicaciones web haciendo uso del diseño arquitectónico.

En el siguiente capítulo, haciendo uso de un caso de ejemplo, se presenta la forma en la cual estos modelos deben ser combinados a fin de generar un esquema de simulación válido.

Capítulo 11. Evaluación de Arquitecturas Web en base a Modelos de Simulación

El objetivo de este trabajo de investigación consiste en la formulación de un modelo de evaluación integral basado en simulación que permita estimar la calidad de una aplicación web en base al análisis de la arquitectura de software en conjunto con atributos de calidad predefinidos. En el capítulo previo se han presentado los lineamientos aplicables para la especificación de modelos RDEVS partiendo de un diseño arquitectónico generado haciendo uso de componentes de aplicación y componentes funcionales. Tales modelos han sido diseñados en base al estudio de las métricas de calidad propuestas, a partir de las cuales se ha formulado un modelo DEVS que releva las mediciones obtenidas como parte del modelo de simulación de la arquitectura. En este capítulo se integran estos elementos como parte del diseño de un escenario de simulación genérico aplicable para el estudio de cualquier diseño arquitectónico. Se incluyen además dos ejemplos de ejecución formulados en base a dos estructuras arquitectónicas comúnmente utilizadas en el desarrollo de aplicaciones web.

11.1 Escenario de Simulación

Siguiendo los lineamientos del framework de Modelado y Simulación (detallado en la *Figura 9.2*), en la Figura 11.1 se esquematiza la forma en la cual debe configurarse un escenario de simulación útil para la evaluación del modelo de red

RDEVS que resulte del proceso de transformación de un diseño de arquitectura de software web a un modelo de simulación de eventos discretos.

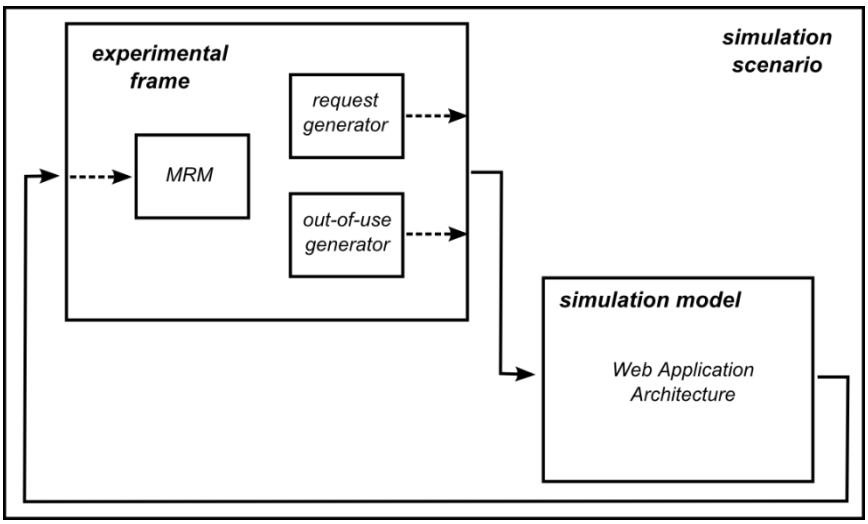


Figura 11.1. Escenario de simulación para arquitecturas de software web.

Como puede observarse, se identifican dos grandes componentes: *i)* un *experimental frame* que define el contexto en el cual se evalúa el modelo de simulación, y *ii)* un *simulation model* que representa el modelo de red RDEVS asociado a la arquitectura de software web bajo estudio.

A su vez, el *experimental frame* contiene tres elementos, a saber:

- Un *request generator* que se encarga de generar el conjunto de eventos de entrada del tipo *request* que serán enviados a la arquitectura para su procesamiento.
- Un *out-of-use generator* cuya responsabilidad es la generación de un único evento *out-of-use* que indica el fin de simulación y, por lo tanto, ingresa al modelo de la arquitectura a fin de desalojar todos los componentes activos.
- El modelo *MRM* (definido en la Ecuación 10.9) con el objetivo de relevar las métricas identificadas en la Tabla 10.4.

En base a este escenario, se definieron los modelos DEVS requeridos para la especificación del *experimental frame*. Tanto el modelo *request generator* como el modelo *out-of-use generator* fueron especificados como modelos atómicos DEVS (al igual que el modelo *MRM*). En base a estos modelos, el modelo *experimental frame* fue definido como un modelo DEVS acoplado que integra los tres componentes como parte de un único modelo de simulación.

11.1.1 Implementación en Java

Los modelos que definen la esquematización de un escenario de simulación válido para arquitecturas de software web fueron implementados haciendo uso de DEVSJAVA y del *framework de RDEVS*. El uso de estas herramientas habilita su simulación visual haciendo uso del entorno DEVS Suite. Las clases implementadas para dar lugar a la definición de escenarios de simulación incluyen *RequestGenerator.java, OutOfUseGenerator.java, ExperimentalFrame.java y SimulationScenario.java*. Estas clases pueden consultarse en *ExperimentalFrame&ScenarioModels*.

En este contexto, la generación de un nuevo escenario de simulación implica la extensión de la clase *SimulationScenario.java*. La nueva clase debe detallar los parámetros requeridos para la configuración de los modelos internos a fin de garantizar la ejecución de la simulación. Además debe indicar cuál es el modelo de red a ser utilizado como *simulation model*. A modo de ejemplo, la Figura 11.2 presenta una captura de pantalla basada en DEVS Suite en la cual se visualiza la clase *SimulationScenario1.java* que modela un escenario de simulación basado en el modelo *Simple* (propuesto en el capítulo previo) como arquitectura de software web bajo análisis.

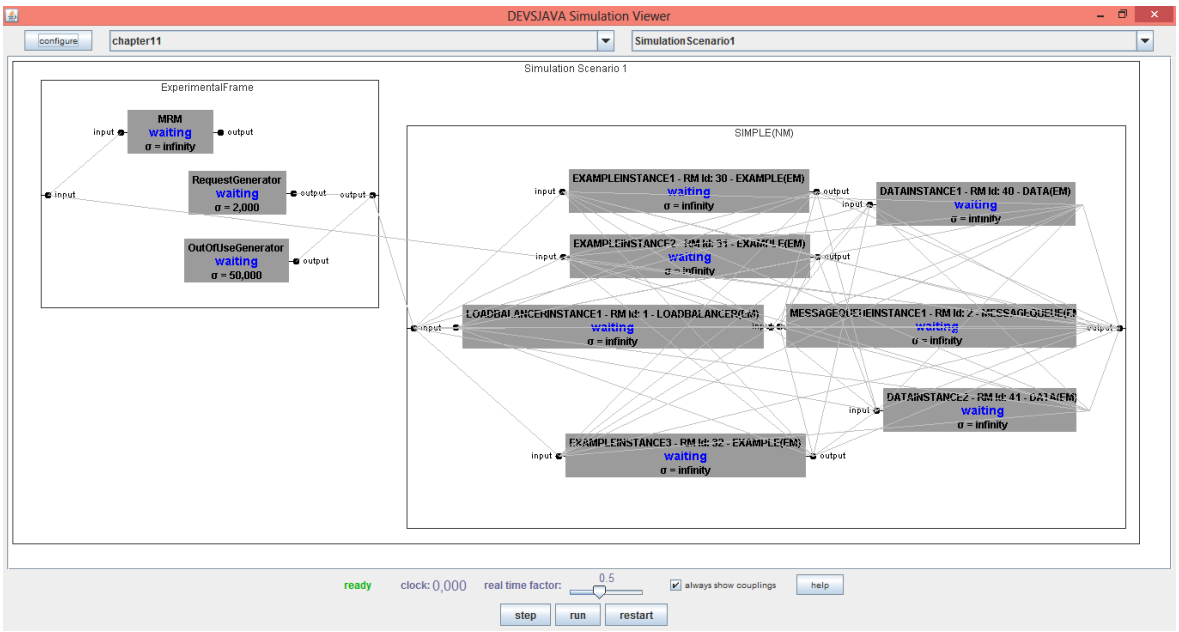


Figura 11.2. Escenario de simulación para la arquitectura "Simple".

11.2 Estudio de las Métricas de Calidad Relevadas

El conjunto de métricas a ser relevadas como resultado del proceso de simulación de los modelos RDEVS diseñados mide atributos de software que pertenecen al nivel más bajo de la jerarquía propuesta en el esquema de calidad SaaS (definido en el apartado *"Definición de un Esquema de Calidad para Servicios Web"*).

Tomando en consideración que todos los valores relevados varían en el intervalo [0,1] (ya que así lo han definido las fórmulas de cálculo asociadas a las métricas), es posible aplicar el criterio propuesto en el apartado *"Agrupamiento de Mediciones basadas en Métricas SaaS"* a fin de obtener valores representativos de las subcaracterísticas y características asociadas a cada caso. La aplicación de este criterio requiere de la asignación de pesos para cada medición de nivel inferior según su impacto sobre el elemento de nivel superior. En este contexto, la Tabla 11.1 presenta los pesos definidos para cada métrica relevada en relación a los atributos de software identificados como parte del esquema de calidad y la forma en la cual se estima un valor para el atributo. A

su vez, las Tablas 11.2 y 11.3 agrupan las mediciones obtenidas para los niveles de calidad referidos a subcaracterísticas y características, respectivamente.

Dada la reducida cantidad de métricas definidas para los atributos (ya que se ha restringido el conjunto original al considerar los criterios de simulación), en casi todos los casos los pesos han sido definidos con el valor 1. Únicamente en el caso del atributo *Service Stability* y de la característica *Reliability* se han utilizado pesos que indican la importancia de cada nivel componente sobre la medición de nivel superior. Sin embargo, es importante destacar que este agrupamiento, además de incluir las mediciones obtenidas como resultado del proceso de simulación, podría incorporar mediciones obtenidas por medio de la aplicación de otro tipo de estrategias de evaluación sobre las arquitecturas de software web.

Desde esta perspectiva, se complementaría el proceso de evaluación permitiendo generar mediciones de las propiedades de alto nivel (subcaracterísticas y características de calidad) incorporando información relevada por diversos medios sobre un mismo artefacto. Tales mediciones podrían considerar los pesos no sólo según su importancia, sino también según la exactitud del proceso de medición llevado a cabo para su obtención.

Tabla 11.1. Estrategia de medición de atributos de software.

MÉTRICA RELEVADA	PESO	ATRIBUTO	MÉTRICA
Time Behavior User Perspective (TBU)	1	*Invocation Time (IT)*	IT = TBU
Replies Accuracy (RA)	1	*Service Accuracy (SA)*	SA = RA
Service Robustness (SR)	1	*Robustness of Service (RS)*	RS = SR
Coverage of Fault Tolerance (CFT)	0.5	*Service Stability (SS)*	SS = 0.5 x CFT + 0.5 x CFR
Coverage of Failure Recovery	0.5		

(CFR)

Tabla 11.2. Estrategia de medición de subcaracterísticas de calidad.

ATRIBUTO	PESO	SUBCARACTERÍSTICA	MÉTRICA
Invocation Time (IT)	1	*Time Behavior (TB)*	TB = IT
Service Accuracy (SA)	1	*Maturity (M)*	M = SA
Robustness of Service (RS)	1	*Availability (A)*	A = RS
Service Stability (SS)	1	*Fault Tolerance (FT)*	FT = FT

Tabla 11.3. Estrategias medición de características de calidad.

SUBCARACTERÍSTICA	PESO	CARACTERÍSTICA	MÉTRICA
Time Behavior (TB)	1	*Performance Efficiency (PE)*	PE = TB
Maturity (M)	0.25	*Reliability (R)*	R = 0.25 x M + 0.5 x A + 0.25 x FT
Availability (A)	0.5		
Fault Tolerance (FT)	0.25		

Luego, tomando los archivos que resultan del proceso de simulación, es posible computar su contenido a fin de calcular las métricas propuestas para los atributos de software, las subcaracterísticas de calidad y las características de calidad.

11.3 Generación de Solicitudes de Usuario

En el contexto del escenario de simulación diseñado en la Figura 11.1, como parte del *experimental frame* se incluye un componente de simulación cuya responsabilidad

consiste en la generación de los eventos *request* que ingresan a la aplicación para su procesamiento. Aunque se ha enunciado que el *request generator* utilizado en los escenarios de simulación ha sido diseñado como un modelo DEVS atómico, como parte del trabajo del modelo integral de simulación desarrollado se propone un conjunto de modelos acoplados DEVS que simulan diferentes comportamientos de usuario. Tales modelos han sido presentados en (Blas, Gonnet y Leone 2017d) y basan su estructura en un esquema de simulación híbrida que combina señales continuas con eventos discretos, dando lugar a la definición de cinco tipos de usuarios. Estos modelos pueden ser utilizados como parte del *experimental frame* a fin de mantener una representación real en la generación temporal de solicitudes de usuario.

11.3.1 Carga de Trabajo y Tipos de Usuarios

El término *carga de trabajo* (*workload*) refiere al nivel de utilización de un recurso de tecnología de información (Fehling y colab. 2014). En cualquier sistema de software, la carga de trabajo puede ser vista como consecuencia del acceso de los usuarios al sistema (Hlavacs y Kotsis 1999; Hlavacs, Hotop y Kotsis 2000; Kurz, Hlavacs y Kotsis 2001). En este contexto, toma distintas formas de acuerdo al tipo de recurso sobre el cual esté siendo analizada. Por ejemplo: los servidores experimentan cargas de procesamiento, los medios de almacenamiento experimentan un aumento o disminución de la cantidad de datos a almacenar y/o del volumen de consultas a realizar, los recursos de comunicación experimentan variaciones de tráfico, entre otros.

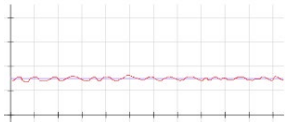
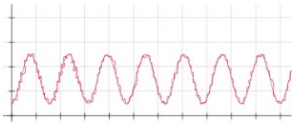
En términos abstractos, la carga de trabajo puede ser vista como la petición a un recurso o servicio de la ejecución de una tarea predefinida, la cual se repite y varía en función de un patrón temporal. Bajo esta perspectiva, puede interpretarse como el patrón de utilización del recurso o servicio a lo largo del tiempo (el cual queda definido por la cantidad de peticiones realizadas en dicho período). De esta manera, las solicitudes que realizan los usuarios al recurso /servicio (que se traducen en cargas de

procesamiento, tráfico de comunicación y/o volumen de información) son el resultado de un patrón temporal asociado a la carga de trabajo.

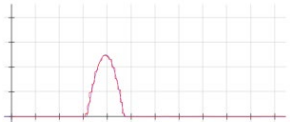
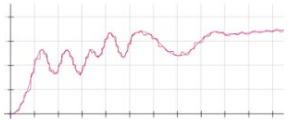
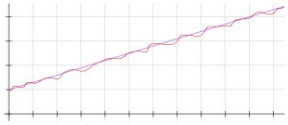
Luego, los usuarios de cualquier tipo de sistema de software experimentan, de acuerdo al tipo de carga de trabajo, un comportamiento temporal (esto es, un patrón de envío de solicitudes que varía con el tiempo). En base a este comportamiento, un usuario genera una secuencia de solicitudes que ingresan al sistema (a fin de ser procesadas) en un período de tiempo específico.

En este sentido, los modelos de simulación propuestos utilizan como marco de referencia los patrones de carga de trabajo propuestos en (Fehling y colab. 2014). Teniendo en cuenta que es común utilizar abstracciones para simplificar la construcción de modelos de simulación (Law y Kelton 1991), cada una de las representaciones requeridas fue diseñada en una función continua *f(t)* asociada a un tipo de carga de trabajo específica. La Tabla 11.4 resume estas asociaciones.

Tabla 11.4. Cargas de trabajo utilizadas como base de los modelos de usuario.

CARGA DE TRABAJO[1]	DESCRIPCIÓN	FUNCIÓN
 Estática *(Static)*	Perfil de utilización más o menos homogéneo a lo largo del período de tiempo en estudio.	Función constante *f(t)=c* cuyo valor de salida *c* se mantiene para cualquier valor de *t*.
 Periódica *(Periodic)*	Perfil de utilización recurrente en horas pico a intervalos de tiempo regulares.	Función periódica y positiva *f(t)* que representa un comportamiento sinusoidal.

[1] Debido a que cada una de estas representaciones evidencia una conducta real, existen ligeras variaciones (propias del comportamiento humano) sobre el patrón temporal predominante.

 Única *(Once in a lifetime)*	Caso especial de carga periódica donde el pico de utilización se da una única vez en el tiempo.	Función *f(t)* definida en tramos en la cual existe un instante de tiempo *tMax* en el que la curva toma un valor máximo *vMax*.
 Impredecible *(Unpredictable)*	Generalización de carga periódica en la que se dan picos/valles de forma no predecible.	Función continua *f(t)* que define un comportamiento aleatorio acotado en el extremo superior.
 De cambio continuo *(Continuously changing)*	El perfil de utilización aumenta o disminuye constantemente en el período de tiempo.	Función *f(t)* que posee una evolución temporal de incremento/decremento lineal a partir de un valor de referencia inicial.

11.3.2 Diseño de los Modelos de Simulación

La Figura 11.3 esquematiza la estructura utilizada como base para la especificación de los modelos de simulación requeridos. Esta estructura plantea que es posible simular el comportamiento de distintos tipos de usuarios por medio de la utilización de técnicas de muestreo sobre funciones continuas. En base a esta conceptualización, los modelos desarrollados utilizan modelos de simulación continuos en combinación con modelos de eventos discretos con el fin de definir un único modelo de simulación aplicable a distintos tipos de usuarios.

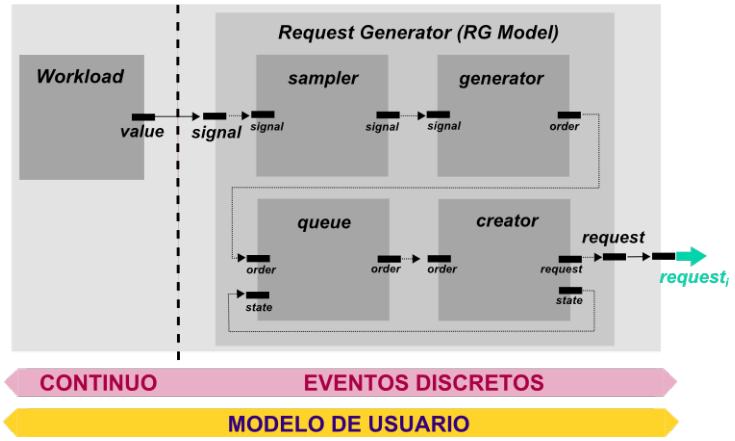


Figura 11.3. Esquema base del modelo de simulación de comportamiento de usuario.

Como puede observarse, un modelo de comportamiento de usuario queda definido haciendo uso de dos modelos interconectados, a saber:

- *Workload:* Modelo continuo que brinda soporte a la función *f(t)* asociada a cada uno de los patrones de carga de trabajo sobre los cuales se generan las solicitudes. En este caso, se especificó un modelo *Workload* para cada una de las funciones identificadas en la Tabla 11.4.
- *Request generator*: Modelo de eventos discretos (diseñado en base a un modelo DEVS acoplado) que lleva a cabo el proceso de muestreo sobre el modelo *Workload* y, posteriormente, define el conjunto de solicitudes a generar. Para llevar a cabo esta responsabilidad, se compone de cuatro modelos DEVS internos que realizan tareas específicas, a saber:

  i) *Sampler*: Cada T unidades de tiempo emite un único evento de salida que contiene el valor entero más próximo al valor relevado en sus eventos de entrada (el cual proviene del muestreo de *Workload*).

  ii) *Order generator*: Al recibir un evento de entrada, emite una cantidad de eventos de salida equivalente al valor entero indicado en dicho evento

(que proviene del muestreo realizado por el modelo *Sampler*). Cada evento de salida incluye el identificador de un tipo de solicitud de usuario.

iii) *Order queue*: Cola de trabajos que administra la entrega de los identificadores de solicitudes de usuario al modelo *Request creator*.

iv) *Request creator*: Recibe el identificador de la solicitud de usuario a crear y, en consecuencia, genera un evento de salida que encapsula la solicitud de usuario que será enviada hacia el modelo de simulación del sistema de software bajo estudio. El formato de los diferentes tipos de solicitudes es recuperado desde un archivo externo de configuración.

Luego, con el objetivo de representar distintos tipos de usuarios, se diseñaron e implementaron cinco modelos acoplados (haciendo uso de los diferentes modelos *Workload* planteados) que mantienen la estructura esquematizada en la Figura 11.3. Tales modelos corresponden a:

- *Static user*: Usuarios que, en todo momento, envían una cantidad fija de solicitudes.
- *Periodic user*: Usuarios que, a lo largo del tiempo, envían al sistema de software una cantidad de solicitudes que sigue un comportamiento periódico.
- *Once in a lifetime user*: Usuarios que, en un único período temporal, envían solicitudes al sistema de software.
- *Unpredictable user*: Usuarios que no siguen un patrón de comportamiento definido en relación a la cantidad de solicitudes que realizan al sistema.
- *Continuously changing user*: Usuarios que presentan un aumento/decremento continuo en la cantidad de solicitudes que envían al sistema.

Para mayores detalles de la especificación de los modelos y las pruebas llevadas a cabo sobre sus implementaciones, se sugiere recurrir a (Blas, Gonnet y Leone 2017d).

11.4 Prueba de Conceptos: Evaluación de Arquitecturas Web Genéricas

En este apartado se presentan dos arquitecturas de aplicaciones web genéricas (basadas en diseños arquitectónicos definidos en términos de componentes de aplicación y componentes funcionales), las cuales han sido evaluadas por medio de los modelos de simulación propuestos. Las Figuras 11.4 y 11.5 presentan estos diseños en términos del conjunto de elementos básicos que definen su estructura global (es decir, componentes de software y componentes de infraestructura).

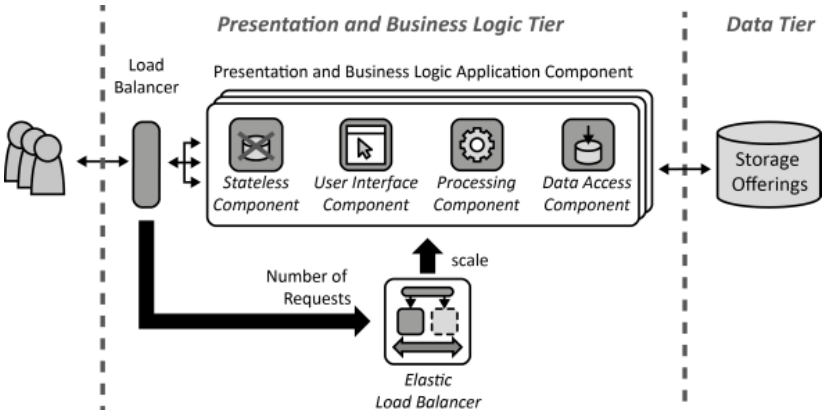


Figura 11.4. Arquitectura de una aplicación web genérica de dos bandas[2].

[2] Ambos ejemplos han sido tomados de (Fehling y colab. 2014).

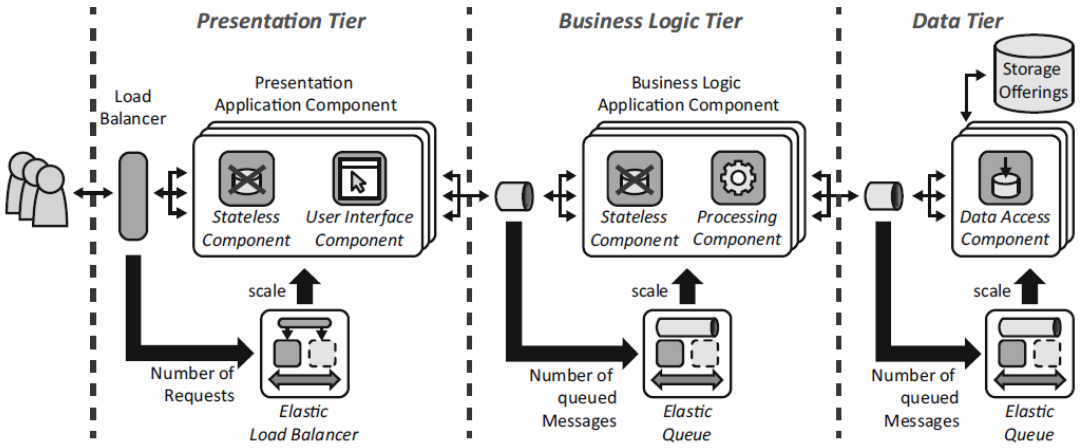


Figura 11.5. Arquitectura de una aplicación web genérica de tres bandas.

Una *arquitectura de dos bandas* agrupa la presentación y la lógica de negocios en un único nivel (esto es, un único componente de aplicación) que facilita el escalado de todas las funcionalidades requeridas como parte de su funcionamiento. Este nivel de aplicación es separado del nivel de datos que, no sólo es más difícil de escalar, sino que usualmente es gestionado por ofertas de almacenamiento provistas por proveedores específicos. En contraposición, una *arquitectura de tres bandas* separa la lógica de presentación, la lógica de negocios y el manejo de datos en diferentes niveles de aplicación (componentes) a fin de dar lugar a un procesamiento intensivo de cómputo de forma independiente al resto de las funcionalidades. Al igual que en el caso de dos bandas, el nivel de datos queda separado del nivel de aplicación (facilitando su administración por parte de un proveedor externo).

La elección de estos patrones se fundamenta en la necesidad de verificar que el comportamiento de los modelos de simulación diseñados guarda relación con los beneficios propios de los estilos de diseño arquitectónicos utilizados como base durante la construcción de la arquitectura. Esto es, los arquitectos de software deciden utilizar un determinado estilo arquitectónico como respaldo de sus diseños con el objetivo de garantizar un conjunto de beneficios propios del esquema de distribución

elegido. En este contexto, las principales características derivadas de un aumento en la cantidad de bandas utilizadas como parte de un diseño arquitectónico incluyen:

- *La posibilidad de replicar en diferente escala cada uno de los componentes de aplicación definidos*: Teniendo en cuenta que cada función tiene asociada una complejidad, no todas las funciones cargan el sistema de software de la misma manera. Luego, en una arquitectura de dos bandas, la única forma de procesar una solicitud es en una instancia del único nivel de aplicación definido. Una vez saturada la cantidad de réplicas instanciables, no podrán procesarse nuevas solicitudes hasta que los componentes se liberen del trabajo asignado. En cambio, en una arquitectura de tres bandas la distribución de responsabilidades posibilita la liberación de componentes a medida que se ejecutan las diferentes funciones. De esta manera, deben saturarse las réplicas disponibles de cada componente de un mismo nivel para que una solicitud quede a la espera por ser atendida. Además, sobre aquellas funciones que conlleven retrasos (ya sea por su complejidad o por el tiempo de procesamiento), es posible generar un esquema de instanciación prioritario a fin de no demorar el tratamiento de las solicitudes por un período de tiempo prolongado.
- *El impacto de los defectos/fallas de las réplicas en el nivel de aplicación difiere según la cantidad de responsabilidades asignadas a los componentes de aplicación definidos*: En una arquitectura de dos bandas, la réplica del nivel de aplicación implica que cada instancia tiene la responsabilidad de ejecutar todas las funciones incluidas en su comportamiento. Un eventual defecto/fallo en alguna de sus funciones, conlleva a un defecto/fallo en componente de alto nivel. En el caso de una arquitectura de tres bandas se desagregan las funciones brindando la posibilidad de cada componente ejecute un conjunto limitado de funcionalidades. Luego, un defecto/falla en una funcionalidad no conlleva necesariamente un defecto/falla en la totalidad del nivel de aplicación.

Tal como se muestra en los diseños arquitectónicos propuestos (Figura 11.4 y 11.5), se han seleccionado dos distribuciones basadas en diferente cantidad de bandas que incluyen el mismo conjunto de componentes funcionales. En este sentido, el comportamiento de los modelos de simulación básicos de ambas arquitecturas puede configurarse de forma semejante a fin de comparar su funcionamiento en términos de los beneficios esperados en cada caso.

11.4.1 Definición de la Arquitectura de Software

Dado que las arquitecturas propuestas incluyen elementos de infraestructura, con el objetivo de construir los modelos de simulación requeridos se generaron las abstracciones necesarias para la representación de la *arquitectura de software*. Tales representaciones fueron diseñadas y verificadas por medio del uso de la herramienta de modelado propuesta en el *Capítulo 8*.

Las Figuras 11.6 y 11.7 presentan los diseños resultantes (incluyendo los componentes de administración que gestionan la asignación de recursos a los componentes de aplicación no definidos). A fin de mantener la correspondencia con los diseños originales, se utilizan las mismas denominaciones.

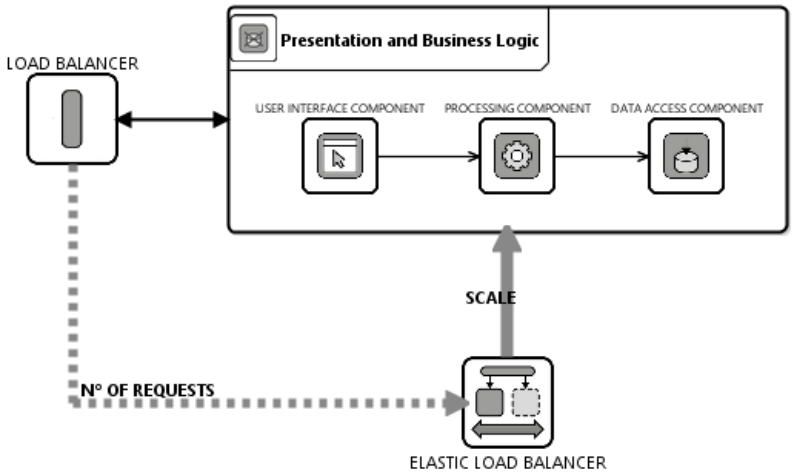


Figura 11.6. Arquitectura de software de una aplicación web genérica de dos bandas.

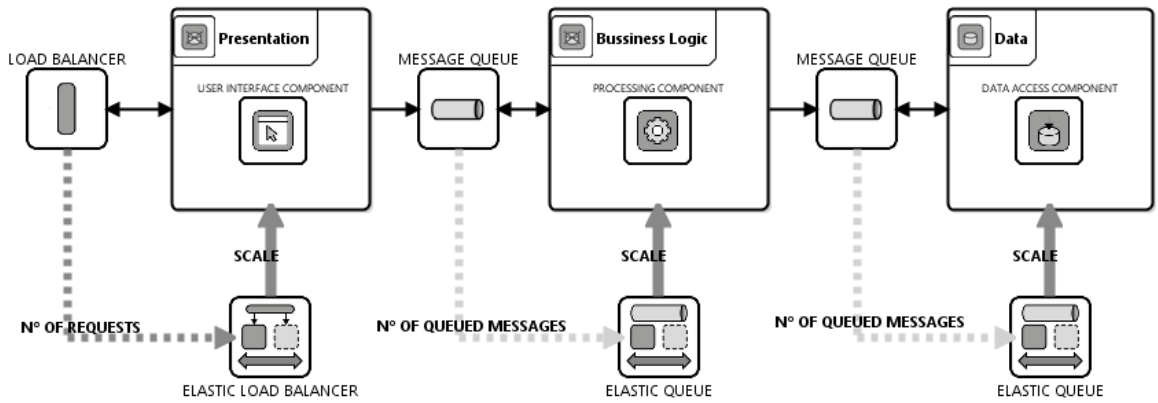


Figura 11.7. Arquitectura de software de una aplicación web genérica de tres bandas.

11.4.2 Implementación de los Modelos de Red RDEVS

Tomando como base los diseños detallados en la sección precedente, se especificaron e implementaron los modelos RDEVS requeridos para la evaluación de ambas arquitecturas. La Tabla 11.5 resume las denominaciones de los modelos desarrollados junto con sus relaciones de composición.

Los modelos *TwoTiers* y *ThreeTiers* fueron diseñados como modelos de red RDEVS compuestos de entidades de ruteo definidas haciendo uso de los componentes de aplicación incluidos en cada caso. Los modelos *MessageQueue* y *LoadBalancer* corresponden, respectivamente, a los modelos esenciales especificados en las Ecuaciones 10.3 y 10.4. A su vez, para la formulación de los modelos esenciales *PresentationBusinessLogic*, *Presentation*, *BusinessLogic* y *Data* se aplicaron los lineamientos indicados en el apartado *"Especificación de Componente de Aplicación No Definido"*.

Tabla 11.5. Denominación de los modelos diseñados para la evaluación.

ARQUITECTURA	COMPONENTES DE APLICACIÓN	
	No Definidos	Definidos
TwoTiers	*PresentationBusinessLogic*	*LoadBalancer*
	Presentation	*LoadBalancer*
ThreeTiers	*BusinessLogic*	*MessageQueue*
	Data	

La estructura de ejecución elegida para las pruebas fue formulada en base a una cantidad fija de réplicas de los *componentes de aplicación no definidos*. Esta cantidad fue definida como *numberOfReplicas*. Por ejemplo, un valor de *numberOfReplicas* = *100*, implica que los modelos de red *TwoTiers* y *ThreeTiers* se encuentran formados por 100 modelos de ruteo asociados a cada uno de los modelos esenciales que representan *componentes de aplicación no definidos* como parte de su definición. Siguiendo este ejemplo, el modelo *TwoTiers* incluirá 100 modelos de ruteo asociados a *PresentationBusinessLogic* mientras que el modelo *ThreeTiers* se compondrá de 300 modelos de ruteo (100 modelos asociados a *Presentation*, 100 modelos asociados a *BusinessLogic* y 100 modelos asociados a *Data*).

Todas las réplicas generadas utilizan la misma configuración de parámetros a fin de garantizar una semejanza de comportamiento en los componentes internos (esto es, el comportamiento de las responsabilidades definidas en base a los componentes funcionales). En todos los casos, como elemento auxiliar de la función *getNextFunctionState()* para la obtención del valor aleatorio a ser comparado con la ocurrencia de falla/defecto, se optó por aplicar una distribución uniforme en el rango [0; 1] (ver *Figura 10.6*).

A fin de no complejizar la carga de simulación, la generación de eventos de entrada fue diseñada en base a una distribución exponencial (con media de 5

milisegundos) que define la forma en la cual arriban las solicitudes de usuario a la aplicación web. Esta decisión se fundamenta en el objetivo de la prueba (*verificar la validez de los modelos de simulación asociados a la arquitectura de software*), dando lugar a un esquema de validación más simple pero igualmente válido para la comparación de los dos patrones de diseño bajo análisis. Todos los modelos fueron ejecutados durante un tiempo de simulación equivalente a 900000 milisegundos (15 minutos de actividad) con tiempos expresados en fracciones de milisegundos para los componentes funcionales.

El proyecto Java en el cual se implementaron todos los modelos requeridos para la prueba de conceptos se encuentra disponible en *WebApplicationPatterns*. A modo de ejemplo, algunos de los archivos resultantes de la ejecución de las simulaciones realizadas se encuentran disponibles junto con el proyecto. Este conjunto de archivos incluye un ejemplo por cada caso representativo analizado en el apartado *"Resultados"*.

Complemento de los Modelos de Simulación

Teniendo en cuenta que el objetivo de la prueba de conceptos consiste en *verificar la validez de los modelos de simulación desarrollados para las arquitecturas de software web*, es necesario introducir nuevas métricas que contribuyan al estudio de las propiedades a analizar. Luego, a fin de complementar el análisis de las mediciones obtenidas, se incorpora al conjunto de métricas relevadas un nuevo indicador asociado a la cantidad de solicitudes que (en un instante de tiempo dado) se encuentran bajo procesamiento en la aplicación[3]. Esta medición permite estudiar el impacto de la cantidad de réplicas sobre el desempeño del sistema web en relación a las solicitudes de usuario.

En este sentido, la incorporación de esta métrica facilita la interpretación de los valores obtenidos como resultado de la ejecución de las simulaciones planteadas. Esto

[3] La estructura que define el conjunto de datos que se relevan como parte del modelo MRM fue modificada para la incorporación de esta métrica.

se debe a que las solicitudes que se prolongan en el tiempo (dentro del sistema) han ingresado a la aplicación web pero nunca han sido respondidas. Luego, en base a esta información es posible analizar el desempeño de los niveles de aplicación en relación a la cantidad de bandas definidas. Esta ha sido la estrategia utilizada en el apartado *"Resultados Obtenidos según la Cantidad de Réplicas Disponibles"*.

11.4.3 Resultados

Resultados Obtenidos según la Cantidad de Réplicas Disponibles

Se ejecutaron múltiples simulaciones configuradas según *numberOfReplicas = 5, numberOfReplicas = 10*, *numberOfReplicas = 20* y *numberOfReplicas = 50*. En todos los casos se mantuvo fija la configuración de comportamiento de los componentes funcionales a fin de dar lugar a una adecuada comparación de resultados.

En una primera instancia se estudia la cantidad de solicitudes que son atendidas por la aplicación. Del conjunto de simulaciones ejecutadas, es importante observar la forma en la cual ha evolucionado en el tiempo la cantidad de solicitudes de usuario que se encuentran siendo procesadas dentro del sistema de software en un momento dado. En este contexto, la Figura 11.8 presenta (a modo de ejemplo generalizador) los resultados obtenidos en las distintas ejecuciones de las configuraciones planteadas en la *arquitectura de dos bandas*.

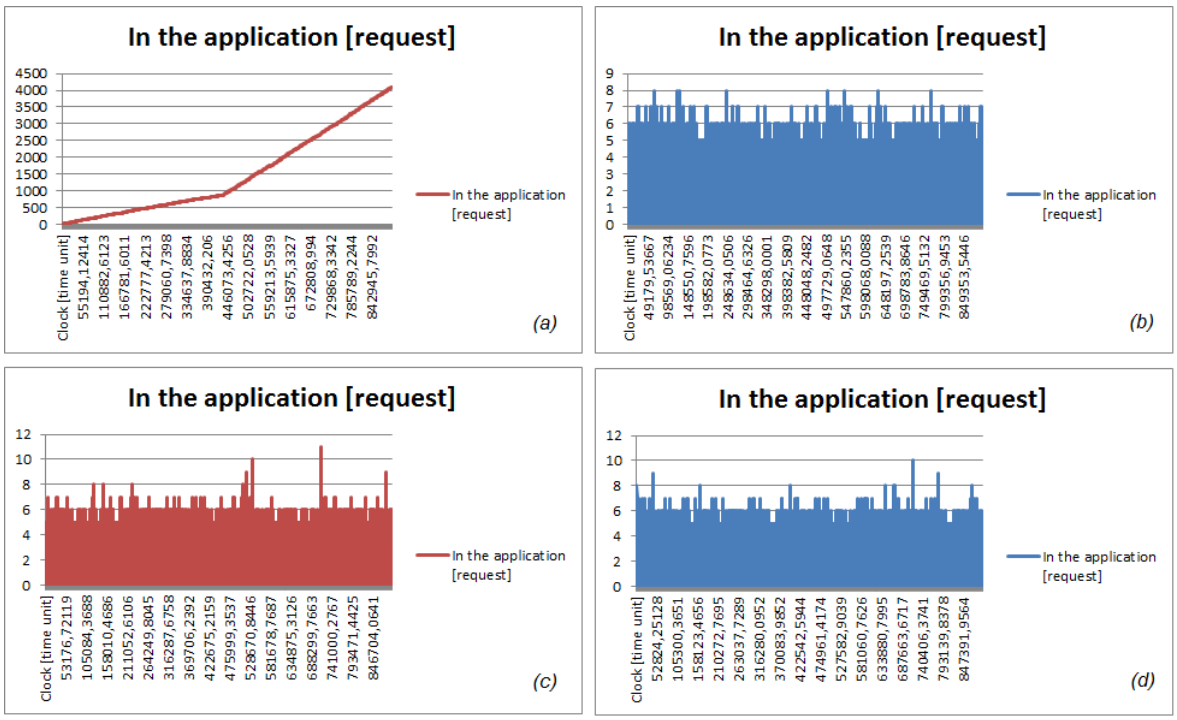


Figura 11.8. Análisis de las solicitudes en proceso en la arquitectura de dos bandas[4].

Como puede observarse, el caso *11.8 (a)* se diferencia de los casos restantes por la existencia de una pendiente pronunciada. Dicha pendiente representa que, a lo largo del tiempo, el sistema continúa recibiendo solicitudes pero no es capaz de atender todos los pedidos de procesamiento. Luego, existen solicitudes que ingresan al sistema de software pero nunca son atendidas. Esto puede deberse a dos motivos:

- *Falta de disponibilidad de los componentes de aplicación*: Dado que no se admite el almacenamiento de aquellas solicitudes que no pueden ser inmediatamente procesadas (por el comportamiento definido en el *Load Balancer*), un subconjunto de las solicitudes entrantes es descartado como consecuencia de la falta de réplicas del componente de aplicación que debe procesarlas.
- *Estados de falla en los componentes de aplicación no definidos*: Cuando un componente de aplicación entra en estado de falla, las solicitudes que se le

[4] Los gráficos *(a), (b), (c)* y *(d)* corresponden, respectivamente, a configuraciones con *numberOfReplicas = 5*, *numberOfReplicas = 10, numberOfReplicas = 20* y *numberOfReplicas = 50*.

envíen para procesamiento son descartadas. Luego, estas solicitudes se pierden dentro de la aplicación.

Analizando la métrica *Tf* (*total number of failures*) en distintas ejecuciones de la misma configuración se observa que la existencia de la pendiente no se deriva únicamente de la presencia/ausencia de estados de falla. La Figura 11.9 presenta una comparativa entre tres ejecuciones de la misma configuración en las cuales se han producido diferente cantidad de fallas. Como puede observarse en el caso *11.9 (a)*, al no presentarse fallas durante la ejecución, la gráfica mantiene una única pendiente a lo largo de todo el horizonte de simulación. Esto indica que dicha pendiente proviene de la *falta de disponibilidad de componentes de aplicación*. En los casos *11.9 (b)* y *11.9 (c)* se observa que por cada falla detectada, la gráfica original modifica su pendiente. Esto implica que, a las solicitudes que se descartan por la *falta de disponibilidad de componentes de aplicación*, se le suman las solicitudes que se descartan como consecuencia de los *componentes que han entrado en estado de falla*.

Si se comparan los casos restantes de la Figura 11.8 con el caso *11.8 (a)*, se observa que al aumentar la cantidad de réplicas disponibles la pendiente deja de existir. Luego, aunque se presenten fallas en los componentes, la no disponibilidad de réplicas para el procesamiento inmediato de las solicitudes tiene un fuerte impacto en el comportamiento del patrón.

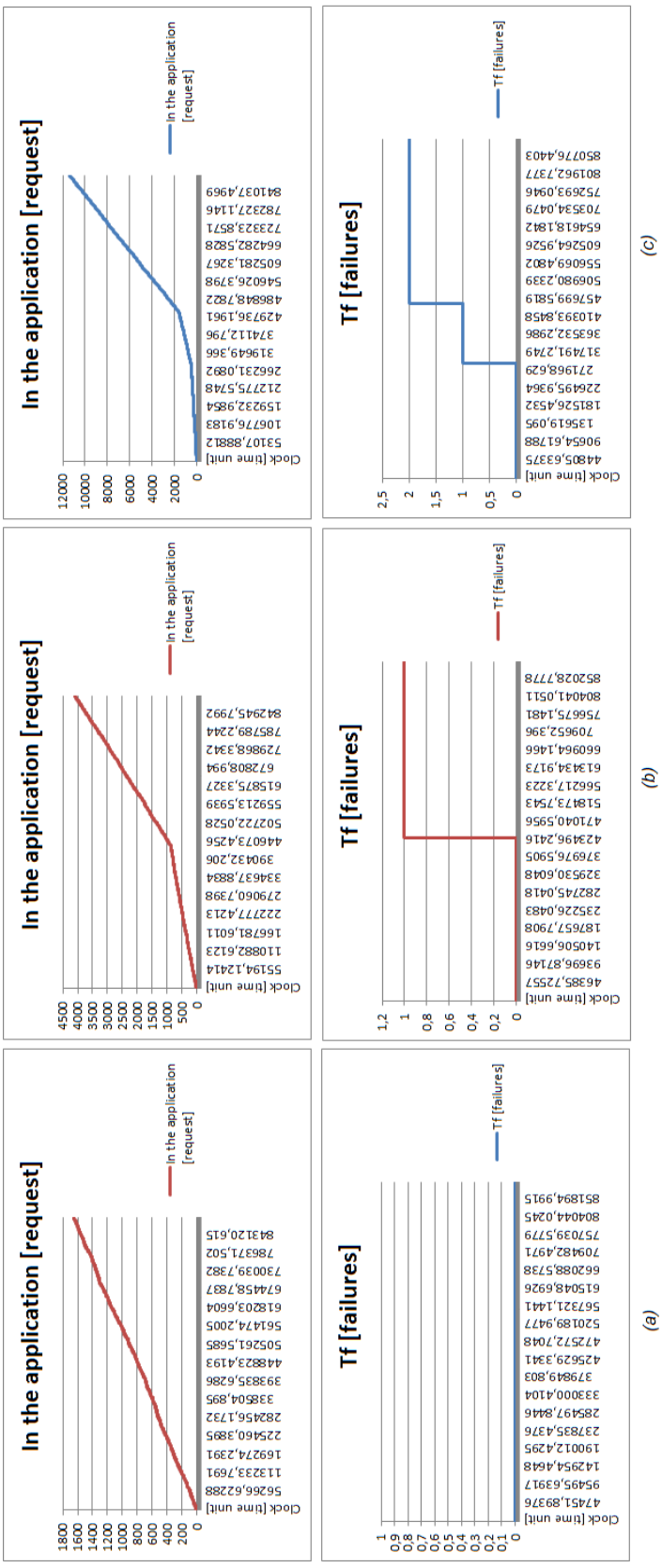


Figura 11.9. Impacto de las falla en las solicitudes que se procesan en la aplicación.

En base a lo enunciado con anterioridad, la Figura 11.10 presenta un promedio de la cantidad de solicitudes respondidas *TR* (*total number of requests*) en las ejecuciones realizadas conforme se aumenta la cantidad de réplicas disponibles. Como puede observarse, cuando se incrementa la cantidad de réplicas de 5 a 10 instancias, se presenta un aumento considerable del valor de TR. Esto se debe a que las solicitudes descartadas en una configuración con *numberOfReplicas* = *5*, son procesadas en una configuración con *numberOfReplicas* = 10. Tal hecho coincide con el análisis formulado en relación a las Figuras 11.8 y 11.9.

Sin embargo, al comparar los valores de TR en las configuraciones con 10, 20 y 50 réplicas no se observan grandes variaciones. Lo mismo ocurre cuando se comparan los casos *11.8 (b), 11.8 (c)* y *11.8 (d)*. Esto se debe a que la capacidad propia determinada por la configuración temporal de las funciones de entrada y procesamiento, no admite una mayor cantidad de solicitudes de usuario (no se debe al comportamiento propio de los componentes involucrados en el procesamiento de la solicitud).

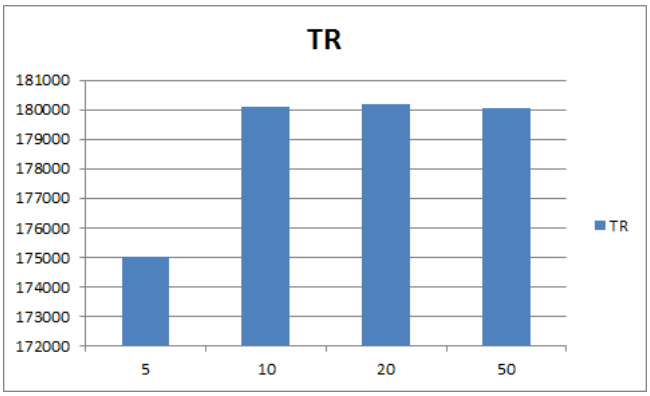


Figura 11.10. Promedio de solicitudes resueltas según cantidad de réplicas.

En relación al comportamiento de la *arquitectura de tres bandas*, la Figura 11.11 presenta un resumen de los resultados generales obtenidos en las distintas ejecuciones

de las configuraciones planteadas. Tal como puede observase, en el caso *11.11 (a)* se presenta una leve pendiente que desaparece al aumentar la cantidad de réplicas disponibles. Si se realiza una comparación de la cantidad de solicitudes que se descartan en este caso en comparación con el caso *11.8 (a)* (es decir, se compara la pendiente de las gráficas), se observa que el rendimiento de la arquitectura de tres bandas es mejor que el de la arquitectura de dos bandas. Esto se debe a que, al desagregar las responsabilidades de la banda de presentación, las solicitudes de usuario que arriban al sistema deben ser atendidas por una única funcionalidad (pudiendo, luego, quedar a la espera de que se desocupen instancias de la banda de lógica de negocios). Este comportamiento, no es admitido en un diseño de dos bandas (por lo que se descarta una mayor cantidad de solicitudes).

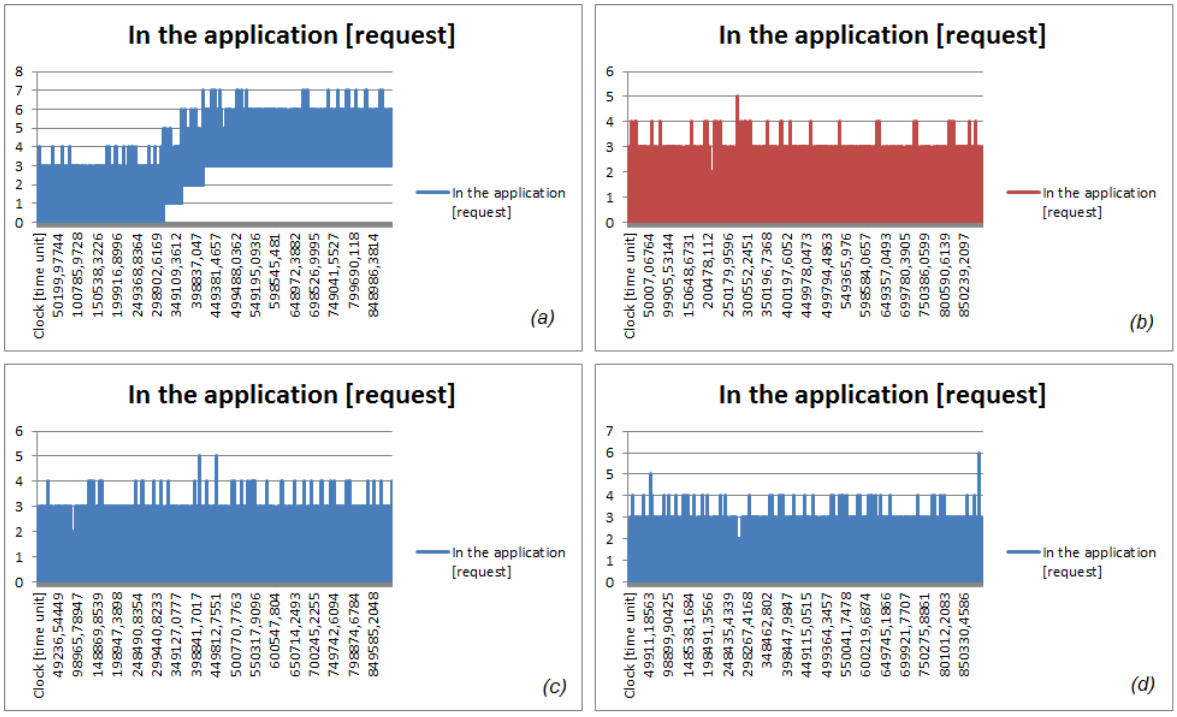


Figura 11.11. Análisis de las solicitudes en proceso en la arquitectura de dos bandas[5].

[5] Los gráficos *(a), (b), (c)* y *(d)* corresponden, respectivamente, a configuraciones con *numberOfReplicas = 5, numberOfReplicas = 10, numberOfReplicas = 20* y *numberOfReplicas = 50*.

Al analizar la cantidad de solicitudes que han sido procesadas por el la aplicación web, se obtiene un gráfico similar al presentado en la Figura 11.10. Con el objetivo de compararlos, la Figura 11.12 presenta ambos resultados. Como puede observase, en el caso de 5 réplicas de componentes es evidente que el modelo de la arquitectura de tres bandas tiene un mejor desempeño para el procesamiento general de las solicitudes. Esto se condice con lo analizado en términos de las solicitudes procesadas/descartadas en ambos estructuras de diseño. Sin embargo, en los casos de 10, 20 y 50 réplicas el valor de TR es similar para ambos modelos.

A fin de comprobar que esta similitud se debe a que la configuración de los modelos no permite una mayor escalabilidad de las solicitudes, se ejecutaron escenarios alternativos modificando el tiempo medio de generación de solicitudes. La Figura 11.13 presenta una nueva comparativa basada en una configuración distinta a la original. En este caso, se observa que al aumentar la frecuencia con la cual arriban las solicitudes, el modelo de la arquitectura de dos bandas presenta diferencias significativas con respecto al modelo de tres bandas cuando existen 5 y 10 réplicas de los componentes de aplicación. En los casos de 20 y 50 réplicas, la cantidad de instancias implementadas debe ser suficiente para procesar todas las solicitudes que ingresan a la aplicación (independientemente de la cantidad de funciones asociadas a los componentes de aplicación). Luego, no se visualiza una diferencia en los valores de TR.

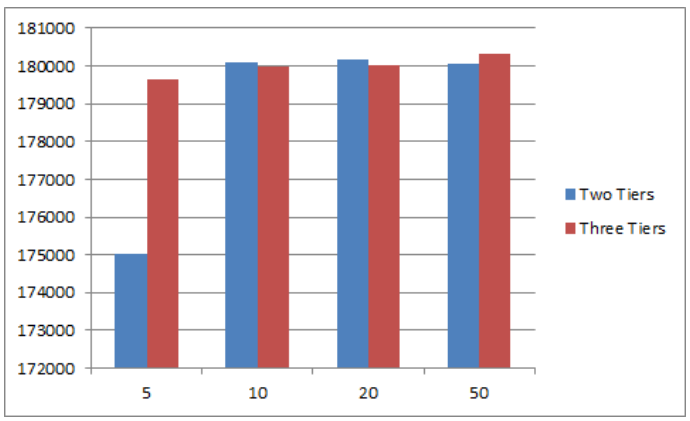


Figura 11.12. Promedio de solicitudes resueltas en ambas arquitecturas.

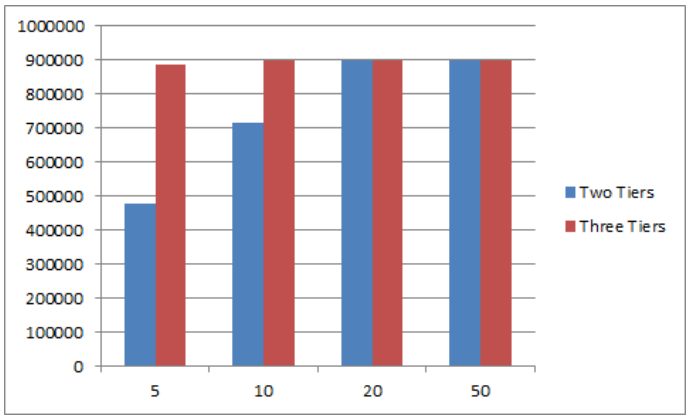


Figura 11.13. Promedio de solicitudes resueltas (nueva configuración).

Luego, en relación a la cantidad de réplicas de componentes de aplicación no definidos, los modelos de simulación diseñados han reflejado un comportamiento consistente con lo esperado en los patrones de diseño que representan.

Resultados Obtenidos cuando todo Defecto Provoca una Falla

A fin de probar el manejo de los estados de falla, se ejecutaron múltiples simulaciones manteniendo la configuración temporal y la probabilidad de defectos según los parámetros utilizados para el caso previo. Para producir estados de falla, se ejecutaron simulaciones modificando el valor de los parámetros *failureProbability* asociados a los distintos componentes funcionales (*failureProbability = 0.5* y *failureProbability = 1*). En todos los casos, la cantidad de réplicas disponibles se mantuvo fija en *numberOfReplicas = 25*.

Tal como se ha definido en el apartado *"Robustez del Servicio (Service Robustness)"*, la métrica SR (*Service Robustness)* refiere a la relación entre el tiempo total de operación

y el tiempo que el servicio ha estado disponible para ser invocado. Cuando entran en estado de fallas todas las réplicas de un mismo componente de aplicación, se entiende que el sistema no se encuentra en servicio ya que no será capaz de procesar nuevas solicitudes (SR = 0). Luego, un valor cercano a 1 indica que es altamente probable que, en un instante de tiempo dado, el sistema se encuentre en servicio.

A modo de ejemplo, las Figuras 11.14 y 11.15 muestran los resultados obtenidos para la métrica SR en cuatro de las corridas de simulación ejecutadas sobre los modelos de dos y tres bandas, respectivamente, con una configuración de *failureProbability = 0.5*. Como puede observarse, la arquitectura de tres bandas mantiene un nivel de SR alto (es decir *SR = 1*) durante un período de tiempo mayor al evidenciado en la arquitectura de dos bandas. Esto se debe a que, independientemente del conjunto de componentes funcionales que entren en falla, la separación de las responsabilidades diseñada en el patrón de tres bandas permite independizar los estados de falla de los componentes de aplicación. En el caso de dos bandas, una falla en cualquiera de los componentes funcionales involucrados en el diseño de un componente de aplicación, da como consecuencia la falla de este último. Luego, ambos modelos han respondido coherentemente en relación a su desempaño cuando se presentan estados de falla.

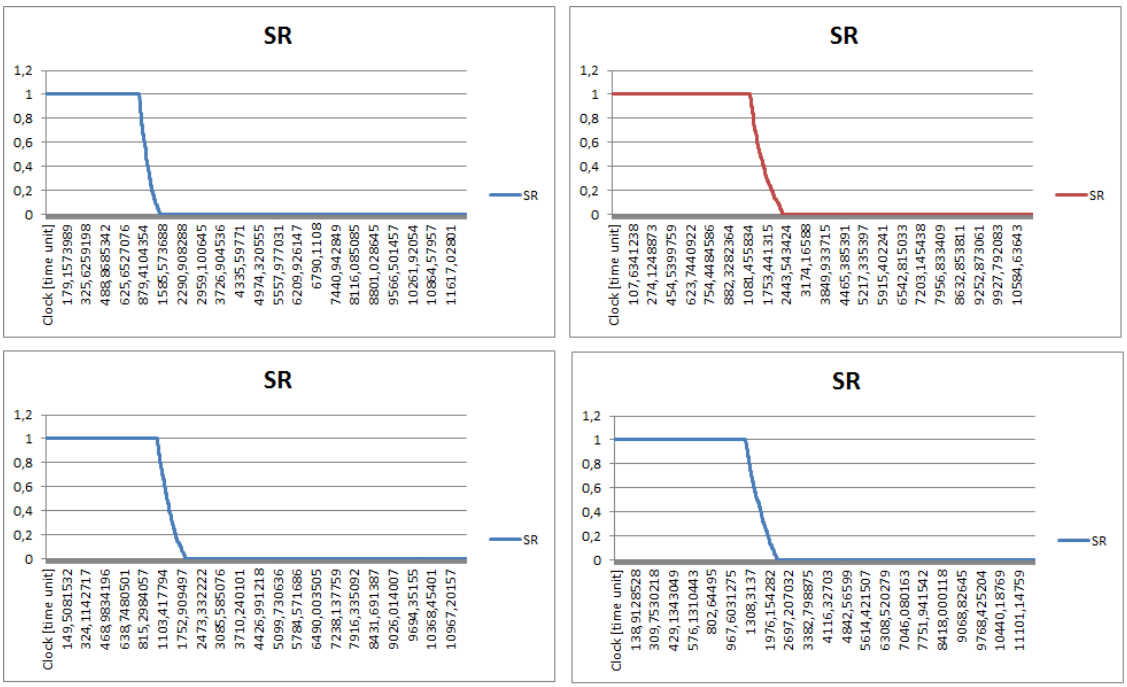


Figura 11.14. SR en el modelo de dos bandas con failureProbability = 0.5.

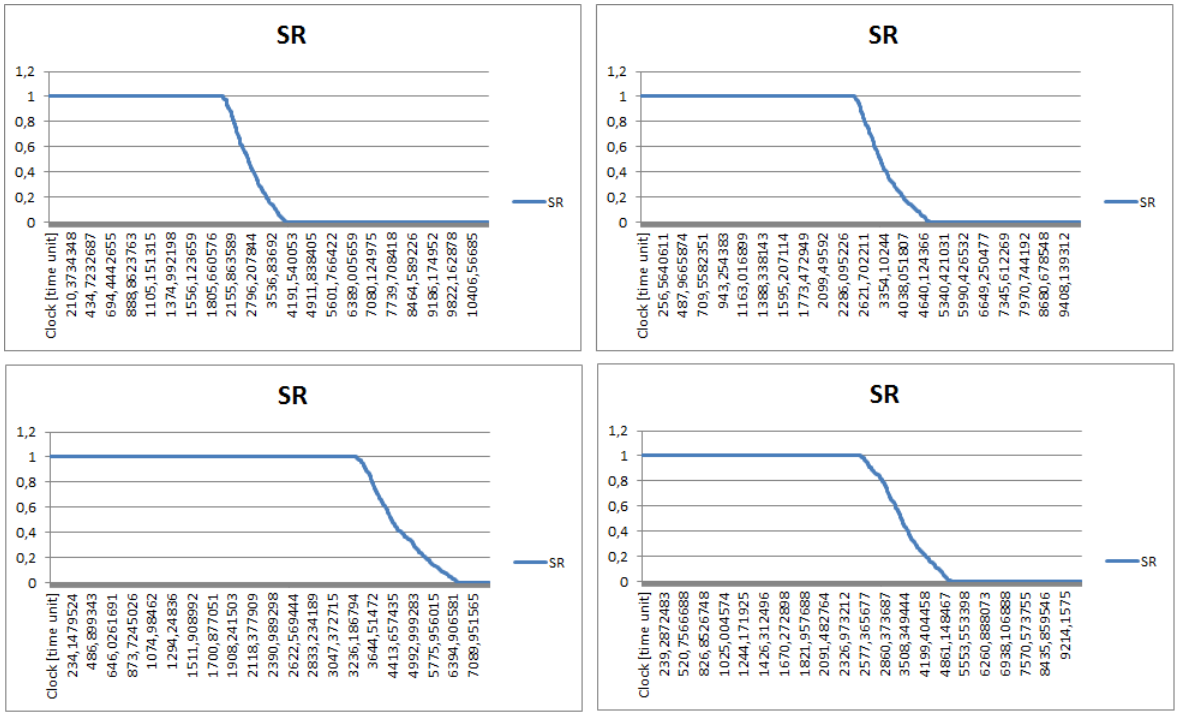


Figura 11.15. SR en el modelo de tres bandas con failureProbability = 0.5.

Realizando la misma comparación con *failureProbability = 1* se obtienen los resultados visualizados en las Figuras 11.16 y 11.17. Al igual que en el caso de *failureProbability = 0.5*, el modelo de tres bandas presenta un mejor desempeño que el de dos bandas. Además, se observa que al aumentar la probabilidad de que un defecto se convierta en una falla (es decir, modificar el valor de *failureProbability* de 0.5 a 1), disminuye el tiempo en el cual el sistema permanece con un alto nivel de SR. Este comportamiento se justifica en el hecho de que, con valor *failureProbability = 1*, la presencia de cualquier defecto en un componente funcional conlleva a un estado de falla del componente de aplicación asociado. Luego, los modelos de simulación reflejan la sensibilidad de la estructura arquitectónica en relación a la presencia de defectos/fallas.

En relación con los estados de falla, la Figura 11.18 presenta una comparativa de la cantidad promedio de solicitudes resuelta con *failureProbability* en 0.5 y 1 en ambos patrones arquitectónicos. Como puede observarse, en los dos casos el modelo de tres bandas ha resuelto una mayor cantidad de solicitudes (ya que, de acuerdo a lo analizado con anterioridad, ha permanecido con un buen nivel de servicio durante un mayor período).

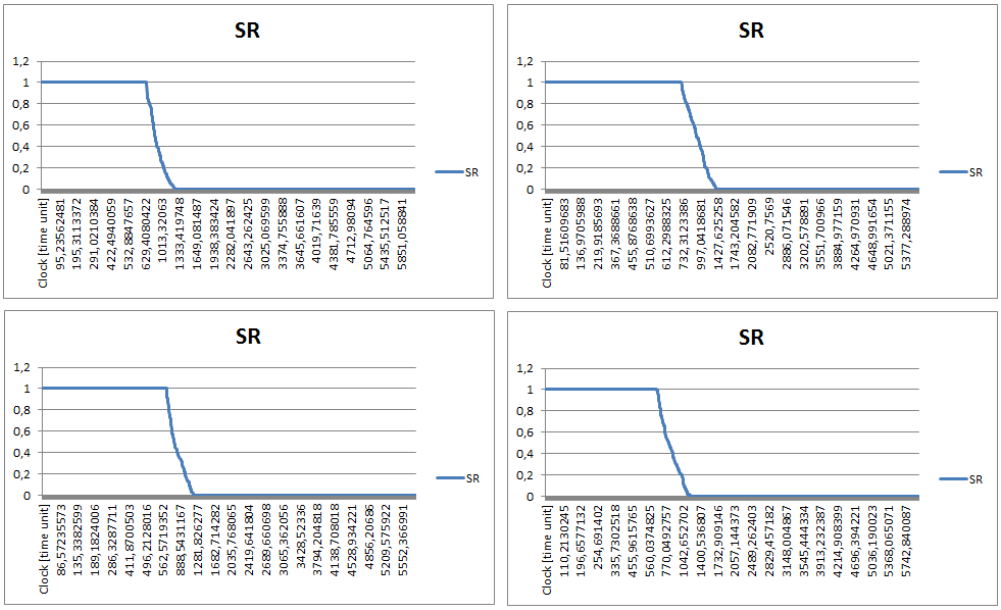


Figura 11.16. SR en el modelo de dos bandas con failureProbability = 1.

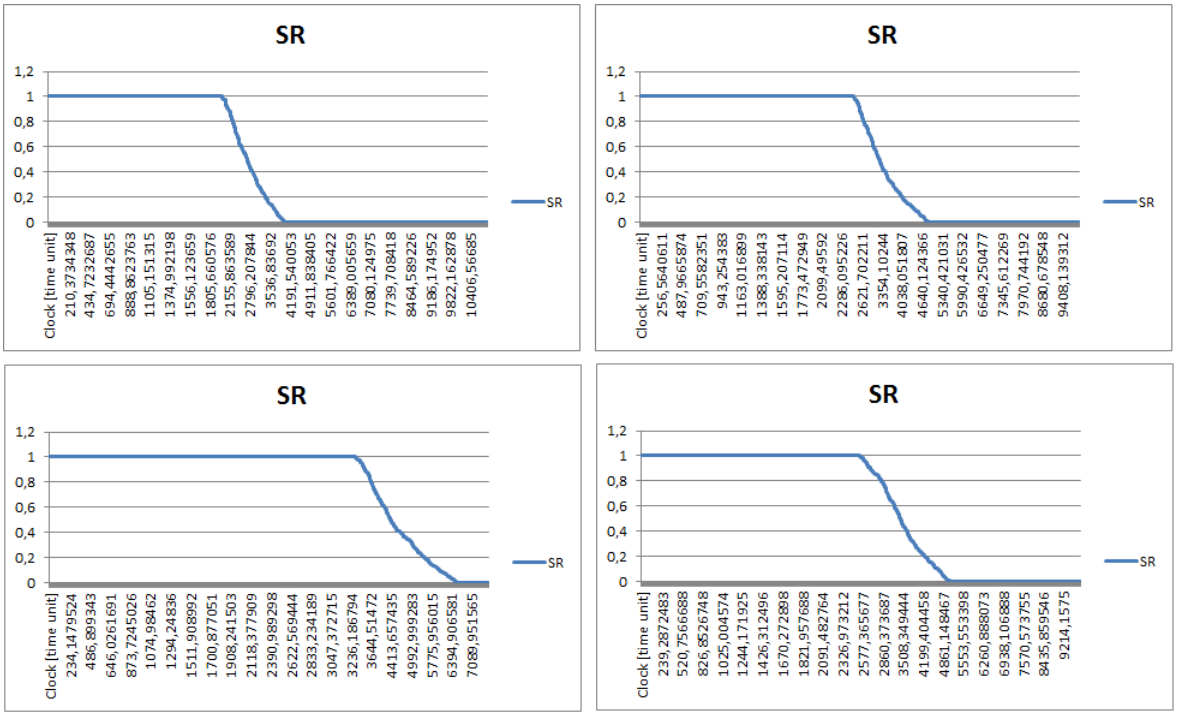


Figura 11.17. SR en el modelo de tres bandas con failureProbability = 1.

Además se observa que al aumentar la probabilidad de falla ante un defecto, la cantidad de solicitudes se reduce en ambos casos. Luego, se entiende que al presentarse estados de falla en períodos de tiempo cortos, el sistema resuelve una menor cantidad de solicitudes de usuario. Este comportamiento también puede evaluarse en sentido inverso, analizando la cantidad de solicitudes que se encuentran dentro de la aplicación para su procesamiento (Figuras 11.19 y 11.20).

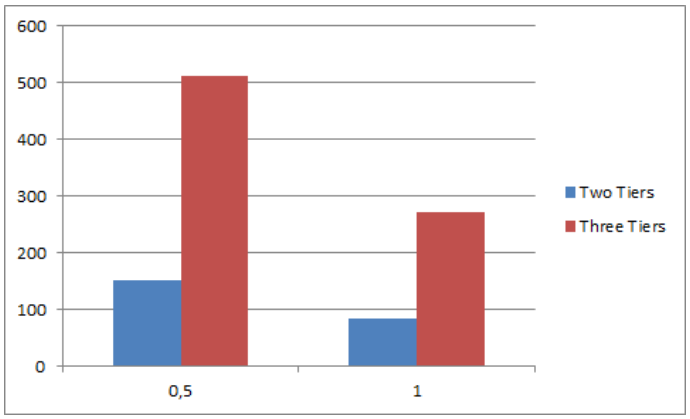


Figura 11.18. Promedio de solicitudes resueltas en estados con fallas.

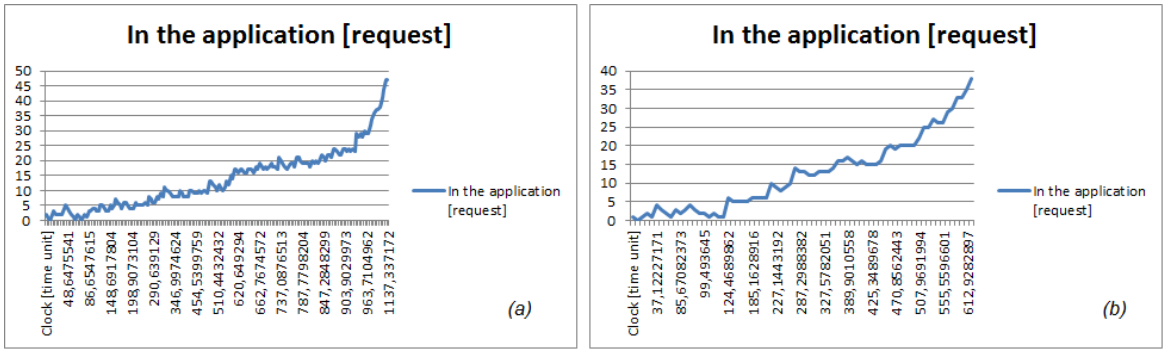


Figura 11.19. Solicitudes en proceso en la arquitectura de dos bandas[6].

[6] Los gráficos *(a) y (b)* corresponden, respectivamente, a configuraciones con *failureProbability = 0.5* y *failureProbability = 1*.

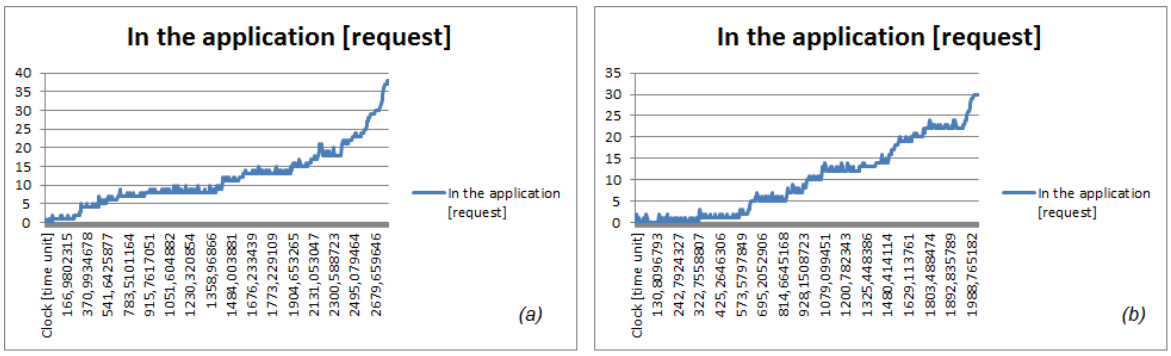


Figura 11.20. Solicitudes en proceso en la arquitectura de tres bandas[7].

Desde esta perspectiva, la cantidad de solicitudes que están siendo procesadas aumenta conforme los distintos componentes van presentando estados de falla. Luego, en los casos *11.19 (a)* y *11.20 (a)* se observa un incremento temporal con menor pendiente que en los casos *11.19 (b)* y *11.20 (b)* - ya que la presencia de estados de falla se prolonga un período de tiempo mayor con *failureProbability = 0.5*-.

Luego, en relación a la presencia de diferentes situaciones asociadas a defectos/fallas en los componentes de aplicación no definidos (como consecuencia de problemas en el comportamiento de los componentes funcionales), los modelos de simulación diseñados han reflejado un comportamiento consistente con lo esperado en los patrones de diseño que representan.

[7] Los gráficos *(a) y (b)* corresponden, respectivamente, a configuraciones con *failureProbability = 0.5* y *failureProbability = 1*.

Conclusiones

En este capítulo se ha finalizado la presentación del modelo de evaluación integral para la estimación de la calidad basado en la construcción de modelos de simulación de eventos discretos asociados a los diseños arquitectónicos de entornos de CC. Se ha diseñado e implementado un escenario de simulación genérico sobre el cual es posible ejecutar los modelos de simulación detallados en el capítulo precedente. Además, se ha descripto brevemente la estructura de un modelo de simulación que permite generar diferentes tipos de comportamiento de usuarios utilizando modelos basados en patrones de carga de trabajo.

Como parte de la validación de la propuesta, se ha presentado una prueba de conceptos en la cual se construyen (haciendo uso de todos los modelos propuestos en esta tesis) dos arquitecturas de software web genéricas en virtud de comparar los resultados obtenidos con los beneficios propios de cada diseño. Los resultados de simulación obtenidos de esta comparativa no sólo dan lugar a una validación de los modelos diseñados sino que, además, actúan como mecanismo de verificación del formalismo RDEVS. En este sentido, todos los escenarios de simulación ejecutados como parte de la prueba de conceptos combinan una gran cantidad de modelos DEVS y RDEVS. En ningún escenario ejecutado se han presentado inconvenientes relacionados a la ejecución de modelos RDEVS haciendo uso de un simulador DEVS.

De esta manera se realiza un primer aporte a las etapas iniciales del desarrollo de software web, las cuales son claves para la calidad del producto final. En este sentido, se brinda un contexto de trabajo en el cual es posible obtener información fiable de la calidad del producto (para la toma de decisiones y la prevención de defectos) a partir de la simulación del diseño arquitectónico asociado a una aplicación de software web.

- Parte V -

Conclusiones y Trabajos Futuros

Capítulo 12. Conclusiones

La evaluación de la calidad esperada en un producto de software específico se ha convertido en un tema de interés en el área de la ingeniería de software. En etapas de desarrollo tempranas, alcanzar un nivel de calidad apropiado implica lograr un diseño arquitectónico adecuado a la naturaleza de los atributos de calidad relevantes. Para esto, es necesario seleccionar los principios de trabajo adecuados buscando un balance entre aquellas propiedades de calidad que compiten entre sí. En los apartados previos se ha presentado un modelo de evaluación integral basado en simulación que permite analizar la calidad esperada en una aplicación de software web en base a su diseño arquitectónico. En este contexto, en este capítulo se exponen las conclusiones generales del trabajo de investigación llevado a cabo en el marco de esta tesis, detallándose sus principales contribuciones y líneas de trabajo futuras.

12.1 Resumen de la Propuesta

La evaluación de arquitecturas de software es una actividad clave del proceso de desarrollo de los sistemas de software en relación a la calidad esperada en el producto final. Los atributos de calidad que guían el proceso de diseño de una arquitectura específica, usualmente se ven reflejados en patrones de diseño que ayudan al arquitecto a resolver problemas frecuentes en términos de composiciones que son utilizadas como plantillas de diseño. Sin embargo, cuando se trabaja con aplicaciones basadas en entornos de computación en la nube, no se disponen de mecanismos

específicos que posibiliten la realización de una evaluación completa en base al dinamismo con el cual operan este tipo de aplicaciones de software.

Teniendo en cuenta que el paradigma de computación en la nube se ha convertido rápidamente en una de las estrategias de solución tecnológica más populares e influyentes del mundo actual, en esta tesis se ha propuesto un enfoque de evaluación integral que permite estimar la calidad de productos de software web por medio de la aplicación de técnicas de simulación de eventos discretos sobre el diseño arquitectónico. Dado que este tipo de servicios de software es diseñado e implementado de forma similar a un sistema de software tradicional, el objetivo de los modelos de simulación propuestos consiste en el relevamiento de las propiedades de calidad requeridas conforme una especificación de requerimientos de software genérica.

Los principales elementos requeridos para comprender el dominio del problema bajo estudio han sido detallados en la *Parte I*. En este apartado se han definido brevemente los problemas identificados en relación a la construcción de aplicaciones web sobre entornos de computación en la nube, dando lugar al estudio de los principales aspectos asociados a la evaluación del diseño arquitectónico y de la calidad esperada en relación al producto de software final. Además, se ha presentado el esquema general de la solución propuesta como base de esta tesis, identificando los modelos componentes que se describen en los apartados subsiguientes.

A fin de estudiar el dominio de calidad en relación al tipo de productos de software bajo análisis, en la *Parte II* se desarrolla el modelo de *esquema de calidad* diseñado para la definición de los atributos de calidad a ser relevados para evaluar la arquitectura. La estrategia de documentación propuesta en este apartado utiliza ontologías de dominio para la formalización del conjunto de elementos requeridos (y sus relaciones) conforme un único esquema semántico. Dicha definición se complementa con un conjunto de reglas SWRL y consultas SPARQL que contribuyen a

su aplicación en un contexto específico. En este sentido, el documento propuesto se basa en la combinación de tres dominios: *modelo semántico de calidad*, *modelo semántico de métricas de software* y *modelo semántico de producto de software*. Luego, dada la especificación de un producto de software, un *esquema de calidad* queda definido como el conjunto de tripletas del tipo *{atributo de software, métrica de software, subcaracterística de calidad}* que debe ser utilizado para especificar los aspectos de calidad asociados al producto bajo estudio, dentro del cual:

- El *atributo de software* referencia a una parte de una entidad de software sobre la cual se requiere evaluar una propiedad de calidad específica.
- La *métrica de software* es el mecanismo de medición que debe ser utilizado para relevar la calidad del *atributo de software*.
- La *subcaracterística de calidad* es la propiedad de calidad que debe ser evaluada haciendo uso de la *métrica de software* sobre el *atributo de calidad* especificado.

De esta manera, la especificación de un *esquema de calidad* implica: *i)* la adopción de un conjunto coherente de propiedades de calidad que deben estar presentes en los distintos componentes del producto de software, y *ii)* la definición de la forma en la cual deben medirse las propiedades identificadas.

Con el objetivo de especificar un *esquema de calidad aplicable a entornos de computación en la nube*, se ha desarrollado una instancia genérica que define la forma en la cual han sido incorporados al modelo de calidad ISO/IEC 25010 los aspectos de calidad propios de este tipo de entornos. Tal incorporación dio como resultado un nuevo modelo de calidad específico denominado *Modelo de Calidad de Software-as-a-Service*. Tomando en consideración que la existencia de atributos de calidad conlleva a la necesidad de especificar métricas que faciliten su determinación, junto con el modelo de calidad se ha identificado un conjunto de métricas de software que facilitan el relevamiento de un subconjunto de propiedades deseadas. Estos elementos convergen

en un único *esquema de calidad* (genérico y extensible) *para entornos de computación en la nube* que sienta las bases requeridas sobre los aspectos de calidad que pueden/deben ser analizados en un servicio de software junto con la forma en la cual pueden ser relevados, no excluyendo a futuro la incorporación de nuevos elementos componentes.

Para la representación de los diseños arquitectónicos de aplicaciones web, en la *Parte III* se han analizado los principales componentes de las arquitecturas de computación en la nube a fin de definir un conjunto de *actividades, pautas y lineamientos* (respaldados por el enfoque de sistema-de-sistemas) que contribuyen a la tarea de diseño arquitectónico. En base a una *estrategia de co-diseño*, se ha definido la forma en la cual es posible diseñar dos sistemas de forma independiente (*sistema de software y sistema de hardware*), a fin de dar lugar a un sistema mayor (*sistema de computación en la nube*) cuyas capacidades son mayores que la mera suma de las capacidades de los sistemas componentes. De acuerdo a esta descomposición, se especifica una estrategia de representación para el sistema de software basada en patrones de diseño que facilita la tarea de modelado en base a un conjunto de componentes y conexiones predefinidos. Esta estrategia se basa en la formulación de un *metamodelo* UML (junto con restricciones OCL) que permite instanciar patrones de diseño como parte de la construcción de un diseño específico asociado a la arquitectura de software de una aplicación web. La base de este modelo consiste en un conjunto de elementos obtenidos a partir del estudio de patrones de diseño que definen componentes y relaciones que ayudan al arquitecto en la especificación de este tipo de arquitecturas. Teniendo en cuenta que un ambiente de computación en la nube puede diseñarse en base a diferentes esquemas de distribución, la descomposición básica aplicada como lineamiento fundamental corresponde a un modelo de capas (en el cual los componentes residen sobre capas funcionales separadas y el acceso es permitido únicamente entre elementos de la misma capa o con su inmediata inferior). En este

contexto, el *metamodelo* trata únicamente con el conjunto de elementos asociados al diseño de la capa de aplicación.

Tomando como base el conjunto de elementos especificados en el metamodelo, se ha presentado una *herramienta de modelado gráfico* que facilita la aplicación de los conceptos identificados como parte de la tarea de diseño arquitectónico. Esta herramienta propuesta permite generar y validar diseños de acuerdo al conjunto de patrones originalmente analizados. De esta manera, la herramienta de software que posibilita la instanciación del metamodelo puede ser utilizada como herramienta de soporte al proceso de diseño definido en términos de la estrategia de co-diseño.

Siguiendo los lineamientos de representación definidos, en la *Parte IV* se describe el principal aporte de esta tesis: el *formalismo de simulación Routed DEVS*. Esta extensión del formalismo original DEVS, evita la generación de una sobrecarga de composición de modelos de simulación cuando se debe resolver un problema de ruteo en modelos de eventos discretos. En este sentido, el formalismo RDEVS debe ser visto como una subclase de DEVS destinada a manejar la complejidad inherente de un proceso de ruteo de eventos en base a los acoplamientos establecidos entre componentes. Por medio de la utilización de una función de ruteo, los modelos que componen una red de simulación establecen un conjunto de destinos específicos para sus eventos de salida incorporando, además, la posibilidad de que cada componente determine la aceptación / denegación de eventos de entrada de acuerdo a su configuración interna. Dado que RDEVS es una subclase de DEVS, los modelos de simulación formalizados en términos de esta nueva extensión son compatibles con modelos de simulación DEVS. Aún más, un simulador DEVS puede ser utilizado para su ejecución. En este contexto, la especificación formal de RDEVS es acompañada con un framework de software que facilita la construcción de modelos RDEVS posibilitando su ejecución en base a un simulador DEVS.

En virtud de formalizar los modelos de simulación requeridos para el estudio de las arquitecturas de software web, para cada componente arquitectónico identificado como parte de la representación propuesta como base de diseño, un modelo RDEVS fue especificado e implementado. En el caso de los *componentes de aplicación no definidos*, dado que la formulación del modelo de simulación asociado depende de su estructura, se establecieron los lineamientos requeridos para dar lugar a esta transformación. Por su parte, la especificación de los modelos de simulación asociados a cada *componente de aplicación definido* fue formulada teniendo en cuenta su comportamiento característico. Finalmente, la definición de un proceso de ruteo de solicitudes como problema de simulación básico a resolver, dio lugar a un modelo RDEVS asociado a múltiples entidades de ruteo que corresponden a réplicas de un mismo componente de aplicación en combinación con un mecanismo de relevamiento de métricas de calidad. El esquema de simulación genérico ha sido complementado con un conjunto de modelos que representan comportamientos de usuario en base a distintos patrones de carga de trabajo, dando como resultado un conjunto configurable de modelos de simulación que brindan la posibilidad de analizar el patrón de eventos de entrada como parte de la carga de trabajo de la aplicación web. Como respaldo del formalismo y de los modelos diseñados, se propuso una prueba de conceptos basada en la comparación de dos patrones de diseño arquitectónico de alto nivel.

Actualmente, es muy alto el porcentaje de errores y defectos que se encuentran en los sistemas de software. Ya sea en productos de software tradicionales o en productos de software basados en servicios de computación en la nube, la existencia de estos errores suele estar relacionada con el cambio continuo de escenario que conlleva a un conjunto de requerimientos poco claros. En ambos contextos, las etapas iniciales del proceso de desarrollo de software son claves para la calidad del producto final. Luego, por medio de la ejecución de los modelos de simulación planteados a partir de la transformación de los diferentes componentes de diseño arquitectónico en estados, transiciones y eventos, se da lugar a una evaluación cuantitativa de indicadores de

calidad esperados como resultado del producto final en términos de propiedades de calidad relevantes en el entorno de computación en la nube. Estos modelos pueden ser utilizados en configuraciones reales a fin de estimar, con un adecuado nivel de detalle, una evaluación de la calidad esperada en una aplicación web en etapas tempranas de desarrollo.

12.2 Principales Contribuciones

En términos generales, el trabajo de investigación realizado impacta en aspectos tecnológicos y económicos vinculados al campo de la industria del software, específicamente en el área de computación en la nube. La aplicación de los modelos y herramientas propuestas contribuyen a mejorar los resultados obtenidos durante la ejecución del proceso de desarrollo de software, permitiendo evaluar la calidad en etapas tempranas de trabajo. En este punto, es importante destacar que el impacto de la propuesta trazada no sólo beneficia la etapa de diseño de la arquitectura sino que además brinda una estimación de la calidad esperada sobre el producto final a lo largo del proceso de desarrollo. Luego, si el diseño arquitectónico es reformulado, es posible evaluar el impacto de los cambios introducidos en relación a la estimación previa.

Sin embargo, la principal contribución desarrollada como parte de la resolución del problema general trabajado a lo largo de esta tesis, ha sido la especificación del formalismo *Routed DEVS*. Aunque en este trabajo la formalización de RDEVS se ha utilizado para la resolución de los modelos de simulación de eventos discretos asociados a arquitecturas de software web, la especificación de los modelos no se encuentra restringida a este dominio.

Luego, el formalismo RDEVS definido como subclase de DEVS constituye un nuevo enfoque para la definición de modelos de simulación de eventos discretos con una menor complejidad estructural que los modelos DEVS equivalentes. Esta particularidad da lugar a estructuras de simulación en las cuales el diseñador puede centrar su

atención en el desarrollo de los comportamientos de dominio (propios de elementos conocidos) y no en la resolución del direccionamiento de eventos. En este sentido, RDEVS puede aplicarse a nuevos dominios en los cuales se presenten problemas cuya resolución posea las siguientes características:

- Existe un conjunto definido de elementos que presentan un mismo comportamiento.
- El accionar de las diferentes instancias de un mismo elemento depende de un conjunto de vínculos externos (pero no así su comportamiento).
- Debe resolverse un problema de ruteo de eventos definido en base a información externa a los estados y/o los parámetros definidos en los modelos de simulación.

Una de las principales ventajas del formalismo RDEVS es la posibilidad de combinar sus modelos con modelos DEVS (tal como se ha hecho en el caso de las arquitecturas de software de aplicaciones web). Por ser una subclase, todo modelo RDEVS es un modelo DEVS que mejora algún aspecto propio de la clase de la cual depende. Luego, la flexibilidad de combinación entre ambos formalismos facilita la construcción de modelos de simulación en escenarios que requieran resolver un problema de ruteo sobre un subconjunto de elementos definidos como parte de la solución del problema.

En este contexto, el estudio de las propiedades inherentes al formalismo RDEVS en comparación con otras extensiones existentes, así como también su aplicación en nuevos dominios de trabajo, constituye una de las principales líneas de trabajo futuras a ser llevadas a cabo a partir de los resultados obtenidos en esta tesis.

12.3 Líneas de Trabajo Futuras

12.3.1 Simulación de Arquietcturas de Software Web

En Relación al Estudio de las Propiedades de Calidad

Herramienta de Software para la Especificación de Esquemas de Calidad

Uno de los principales trabajos futuros a desarrollar en relación a la ontología QSO, es el diseño e implementación de una herramienta de software que automatice no sólo la instanciación de esquemas de calidad basados en los modelos semánticos, sino también su estudio (es decir, tanto la ejecución de preguntas de competencia como también el análisis de cobertura).

En (Carvallo y colab. 2004) se presenta una primera aproximación a este tipo de herramienta en la cual se le permite al usuario especificar una herencia de características y subcaracterísticas basadas en el modelo de calidad de producto definido en el estándar (ISO/IEC 9126-1 2001). En este caso, para cada factor identificado, el usuario tiene la posibilidad de asociar métricas relacionadas. Sin embargo, dado que el estándar ISO/IEC 9126-1 ha sido reemplazado por el estándar ISO/IEC 25010, el documento generado en esta herramienta no contempla las propiedades de calidad asociadas al modelo de calidad de producto vigente. Aún más, la herramienta no provee mecanismos de consulta ni inferencia, por lo que aunque sienta bases similares a las requeridas para la automatización de la ontología QSO, su definición no es lo suficientemente sólida como para tomarla como base de trabajo.

En este contexto, los modelos semánticos que componen la ontología QSO junto con sus reglas SWRL y consultas SPARQL pueden ser tomados como base para construir una herramienta de software basada en ontologías cuyo principal objetivo sea la creación, almacenamiento, modificación y consulta de esquemas de calidad. Como funcionalidad adicional, se puede incorporar el análisis de cobertura.

Con este soporte tecnológico el equipo de desarrollo involucrado en la construcción de un producto de software específico podría, de forma sencilla y flexible, trabajar los distintos aspectos de calidad vinculados a un artefacto particular o al producto general, permitiéndoles identificar de forma automática los aspectos de calidad que impactan sobre un determinado elemento y la información requerida y disponible para su medición. Además, por medio del uso de gráficos estadísticos se podría representar la relevancia de cada propiedad de calidad (característica y/o subcaracterística) en relación a los distintos niveles de descomposición del producto de software (producto, artefacto y entidad) a fin de analizar su nivel de cumplimiento a medida que se avance en el desarrollo. De esta forma, el usuario podría visualizar gráficamente el impacto de los aspectos de calidad sobre el producto de software final.

Ampliación del Conjunto de Métricas y Propiedades de Calidad incluidas en el Esquema de Calidad SaaS

Teniendo en cuenta que el documento de calidad definido en el apartado *"Esquema de Calidad SaaS: Instanciación de la Ontología QSO*"* brinda la posibilidad de ampliar el conjunto de elementos incluido en la definición inicial, un trabajo futuro factible de ser abordado consiste en la incorporación de nuevas características, subcaracterísticas y/o atributos de calidad que amplíen la estructura del modelo de calidad subyacente. Además, el conjunto de métricas definido en el apartado *"Métricas para Evaluar la Calidad en Servicios de Software"* puede ser ampliado para la incorporación de nuevas métricas de software que den lugar a la realización de una definición más detallada de los indicadores de calidad relevantes en contexto de computación en la nube.

Luego, la adición de nuevas propiedades como parte del documento propuesto para la formalización de la calidad en los entornos de computación en la nube enriquecería la propuesta. De esta manera, se brindaría un marco de trabajo genérico aplicable, en principio, a cualquier situación de desarrollo referida tanto a las

aplicaciones web como así también a la infraestructura de hardware utilizada como mecanismo de soporte de este tipo de productos de software.

Utilización de Esquemas de Calidad para el Estudio y Análisis de Otros Artefactos de Software

El uso de ontologías como parte de la estructuración de los esquemas de calidad brinda múltiples beneficios, no solo para el equipo de desarrollo sino también para el cliente final. En este sentido, la definición de los aspectos de calidad asociados a un producto de software específico en un único documento ayuda a gestionar la calidad desde etapas tempranas del proceso de desarrollo.

Aunque en el contexto de esta tesis los esquemas de calidad han sido utilizados como base para la identificación de los objetivos de simulación en el marco de aplicaciones web, este tipo de documentos puede ser utilizado para analizar otro tipo de características referidas a diferentes artefactos de software (no necesariamente vinculados a entornos de computación en la nube).

En Relación a la Representación de Arquitecturas Basadas en Entornos de Computación en la Nube

El modelo formulado en el apartado *"Metamodelo de Diseño para Arquitecturas Web"* no incluye elementos referidos a la arquitectura de infraestructura subyacente a la aplicación de software analizada (es decir, la capa de hardware). En este sentido, un trabajo a futuro consiste en la incorporación del conjunto de conceptos, relaciones y restricciones que permitan instanciar arquitecturas orientadas a la totalidad de capas involucradas en los entornos de computación en la nube. Esto es, incluir tanto la lógica de la aplicación web como la disposición de los componentes de la infraestructura que le brindan soporte.

Dado que la herramienta de software que brinda soporte al metamodelo puede ser utilizada como herramienta auxiliar del proceso de diseño arquitectónico basado en co-diseño (definido en el apartado *"Proceso de Diseño: Pasos, Pautas y Lineamientos*

Generales"), un trabajo futuro inmediato relacionado con estos elementos consiste en la incorporación de los lineamientos definidos como guía durante el modelado de una arquitectura específica. De esta manera, se contribuye a la elaboración de un diseño arquitectónico que aplique los elementos, composiciones y relaciones de forma apropiada conforme lo analizado en los patrones de diseño. Aunque estos lineamientos no reemplazan el proceso de verificación definido en términos de las restricciones OCL, el arquitecto será menos propenso a la formulación de diseños inapropiados ya que los lineamientos lo ayudarán durante su elaboración.

En Relación a los Modelos de Simulación Asociados a los Componentes Arquitectónicos

Dado que el enfoque de simulación presentado se basa en los componentes definidos en el metamodelo propuesto, la automatización del proceso de transformación *diseño de arquitectura-a-modelo de simulación* es uno de los principales trabajos futuros a abordar para completar la propuesta.

En este contexto, la especificación del comportamiento de los usuarios también podría ser incluida como parte del proceso de mapeo a fin de generar un escenario apropiado para la evaluación arquitectónica. Eventualmente, si se incorporan nuevos componentes referidos a la infraestructura como parte del metamodelo, los modelos de simulación asociados podrían ser diseñados haciendo uso de RDEVS dirigiendo los eventos desde la capa de aplicación hacia la capa de infraestructura sin requerir de otro tipo de mecanismos para la resolución de esta comunicación.

12.3.2 Arquitecturas de Sistemas Auto-Adaptativos combinando Teoría de Control y Simulación

Los sistemas de software modernos están sujetos a incertidumbres, como la dinámica en la disponibilidad de recursos o los cambios en los objetivos del sistema. Desde una perspectiva de Ingeniería de Software, un sistema es adaptativo cuando modifica su estructura o comportamiento en tiempo de ejecución (es decir, sin

interrumpir su servicio). Un *sistema adaptativo* puede ser combinado con un "administrador de adaptación" a fin de garantizar la satisfacción de sus requerimientos aun en aquellos casos en los que se produzca una violación de los mismos (ya sea debido a cambios en el entorno de ejecución, interacciones indebidas con el usuario, nuevo comportamiento de componentes de terceros o por modificación en los requisitos). El acoplamiento de un sistema de software con un "administrador de adaptación" da lugar a un *sistema de software auto-adaptativo*.

En el área de Teoría de Control, este "administrador de adaptación" es de forma genérica denominado "controlador". En este contexto, dado el alto costo que tiene la manipulación de los sistemas físicos, los estudios referidos al diseño de controladores son frecuentemente llevados a cabo haciendo uso de modelos de simulación. Esta técnica no sólo permite realizar múltiples pruebas de diseño en etapas previas a la incorporación del controlador como mecanismo complementario al sistema bajo análisis, sino que además posibilita la experimentación de múltiples situaciones sin afectar el funcionamiento del sistema físico bajo estudio.

Teniendo en cuenta que en la actualidad los sistemas de software auto-adaptativos han comenzado a emerger debido a la rapidez con la cual los productos de software deben incorporar nuevos requerimientos referidos a plataformas y modelos de ejecución; es importante establecer mecanismos y estrategias de trabajo flexibles que faciliten tanto la especificación como así también el diseño e implementación de este tipo de sistemas. Puede afirmarse que, al igual que en cualquier tipo de sistema de software, las etapas iniciales del desarrollo son claves para el éxito del producto final. Una mala decisión en las estrategias de control a implementar en un sistema auto-adaptativo tendrá como consecuencia un incremento en los costos de desarrollo y mantenimiento y, en muchos casos, llevará a que el sistema quede obsoleto por no cumplir con los requerimientos y expectativas de los usuarios/clientes. En estas instancias, el diseño de la arquitectura del sistema de software a desarrollar constituye

una base de análisis para estudiar distintas estrategias de control sobre una arquitectura modelo.

A partir de esta situación, se plantea como línea de trabajo futura (derivada del contenido desarrollado en esta tesis), el estudio de la dinámica de sistemas de software auto-adaptativos haciendo uso de su especificación arquitectónica en base a la aplicación de modelos DEVS y RDEVS. Desde este punto de partida, resulta conveniente evaluar la flexibilidad de los modelos de simulación para alterar la estructura definida conforme se adapta el diseño arquitectónico subyacente a cambios propios del sistema asociado.

12.3.3 Extensiones del Formalismo DEVS: Caracterización y Comparación de Propiedades

Tal como se ha enunciado en esta tesis, la especificación de *Routed DEVS* como extensión del formalismo DEVS fue desarrollada bajo la forma de subclase. Sin embargo, en la actualidad no existen mecanismos de soporte para la comparativa de diferentes modelos de simulación desarrollados en DEVS.

Normalmente, las extensiones del formalismo surgen en respuesta a la necesidad de modelar un nuevo tipo de problema que, en una etapa previa, no podía ser resuelto. En este contexto, no todas las extensiones del formalismo son comparables en términos de las mismas características. Luego, la definición de *tipos de extensiones DEVS* posibilita el establecimiento de un conjunto de propiedades básicas aplicables para la determinación del nivel de adecuación de los formalismos ante distintas situaciones de modelado y/o simulación. Además, una caracterización de este tipo posibilitaría la determinación de posibles compatibilidades entre modelos expresados en múltiples formalismos, brindando la posibilidad de definir una estructura de simulación similar a la propuesta como *DEVS Bus* (Zeigler, Praehofer y Kim 2000).

Un primer trabajo en esta línea de investigación ha sido desarrollado en (Blas y colab. 2018). En este contexto, con el objetivo de facilitar la comparación de modelos (en principio) y formalismos (como consecuencia de la comparación de modelos), una de los principales problemas a abordar como parte de la línea de investigación es la ausencia de métricas e indicadores universales tendientes a relevar la complejidad de modelos de simulación de eventos discretos. La comparación desde un punto de vista conceptual referido a las diferencias entre extensiones derivadas de un único formalismo, es un área de trabajo prioritaria para la caracterización de tipos de extensiones DEVS.

Al mismo tiempo, siguiendo esta línea de investigación, específicamente sobre RDEVS surgen múltiples áreas de trabajo relacionadas con el estudio del formalismo en relación a otras extensiones DEVS existentes. Una de áreas más prometedoras plantea la combinación de RDEVS con Dynamic DEVS (Barros 1997) con el objetivo de aprovechar las ventajas de una configuración dinámica de modelos de simulación de acuerdo al ruteo de eventos.

- Parte VI -

Bibliografía y Apéndices

Bibliografía

(Abebe y Tonella 2015) Abebe, S. L., Tonella, P. (2015). Extraction of domain concepts from the source code. *Science of Computer Programming*, 98, 680-706.

(Albin 2003) Albin, S. T. (2003). *The art of software architecture: design methods and techniques*. John Wiley & Sons.

(Albin-Amiot y Guéhéneuc 2001) Albin-Amiot, H., & Guéhéneuc, Y. G. (2001). Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*.

(Al-Azzani y Bahsoon 2012) Al-Azzani, S., & Bahsoon, R. (2012). SecArch: Architecture-level evaluation and testing for security. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture* (pp. 51-60). IEEE.

(Al-Badareen y colab. 2011) Al-Badareen, A. B., Selamat, M. H., Jabar, M. A., Din, J., & Turaev, S. (2011). Software quality models: A comparative study. In *International Conference on Software Engineering and Computer Systems* (pp. 46-55). Springer, Berlin, Heidelberg.

(Ampatzoglou, Frantzeskou y Stamelos 2012) Ampatzoglou, A., Frantzeskou, G., & Stamelos, I. (2012). A methodology to assess the impact of design patterns on software quality. *Information and Software Technology*, 54(4), 331-346.

(Armbrust y colab. 2010) Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski & Gunho Lee (2010). A view of cloud computing. *Communications of the ACM* 53 (4): 50-58.

(Atkinson y Kuhne 2003) Atkinson, C., & Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE software*, 20(5), 36-41.

(Aydiner 2017) Aydiner, A. S. Linking Information System Capabilities with Firm Performance: A Review of Theoretical Perspectives and New Research Agenda.

(Bachmann y colab. 2004) Bachmann, F., Bass, L., Klein, M., & Shelton, C. (2004). Experience using an expert system to assist an architect in designing for modifiability. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture* (pp. 281-284). IEEE.

(Balushi, Sampaio y Loucopoulos 2013) Balushi, T. H., Sampaio, P. R. F., & Loucopoulos, P. (2013). Eliciting and prioritizing quality requirements supported by ontologies: a case study using the ElicitO framework and tool. *Expert systems*, 30(2), 129-151.

(Barbacci y colab. 1995) Barbacci, M., Klein, M. H., Longstaff, T. A., & Weinstock, C. B. (1995). Quality Attributes (No. CMU/SEI-95-TR-021). Carnegie-Mellon University.

(Barros 1997) Barros, F. J. (1997). Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(4), 501-515.

(Bass, Clements y Kazman 2012) Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

(Bayer y Muthig 2006) Bayer, J., & Muthig, D. (2006). A view-based approach for improving software documentation practices. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems* (pp. 10-pp). IEEE.

(Bergero y Kofman 2014) Bergero, F., & Kofman, E. (2014). A vectorial DEVS extension for large scale system modeling and parallel simulation. *Simulation*, 90(5), 522-546.

(Bertoa, Vallecillo y García 2006) Bertoa, M. F., Vallecillo, A., & García, F. (2006). An ontology for software measurement. In *Ontologies for Software Engineering and Software Technology* (pp. 175-196). Springer, Berlin, Heidelberg.

(Blas, Gonnet y Leone 2014) Blas, M. J., Gonnet, S., & Leone, H. (2014). Una Taxonomía de Atributos de Calidad para la Evaluación de Arquitecturas de Software por medio de Simulación. En *Actas del 2º Congreso Nacional de Ingeniería Informática y Sistemas de Información (CoNaIISI)*, 936-944.

(Blas, Gonnet y Leone 2015) Blas, M. J., Gonnet, S., & Leone, H. (2015). Un Modelo para la Representación de Arquitecturas Cloud basadas en Capas por medio de la Utilización de Patrones de Diseño: Especificación de la Capa de Aplicación. En *Actas del 3º Congreso Nacional de Ingeniería Informática y Sistemas de Información (CoNaIISI)*.

(Blas, Gonnet y Leone 2016a) Blas, M., Gonnet, S., & Leone, H. (2016). Especificación de la Calidad en Software-as-a-Service: Definición de un Esquema de Calidad basado en el Estándar ISO/IEC 25010. In *Simposio Argentino de Ingeniería de Software (ASSE 2016)-JAIIO 45*.

(Blas, Gonnet y Leone 2016b) Blas, M. J., Gonnet, S., & Leone, H. (2016). Building simulation models to evaluate web application architectures.» En *Computing Conference (CLEI), 2016 XLII Latin American*, 1-12. IEEE.

(Blas, Gonnet y Leone 2017a) Blas, M. J., Gonnet, S., & Leone, H. (2017). Routing Structure Over Discrete Event System Specification: A DEVS Adaptation To Develop Smart Routing In Simulation Models. In *2017 Winter Simulation Conference* (pp. 774-785). IEEE.

(Blas, Gonnet y Leone 2017b) Blas, M. J., Gonnet, S., & Leone, H. (2017). An ontology to document a quality scheme specification of a software product. *Expert Systems*, 34(5), e12213.

(Blas, Gonnet y Leone 2017c) Blas, M. J., Gonnet, S., & Leone, H. (2017). Definición de un Proceso de Diseño basado en un Metamodelo para la Especificación de Arquitecturas de Software de Entornos Cloud Computing Utilizando el Enfoque de "Sistema-de-Sistemas". En Actas del 5º Congreso Nacional de Ingeniería Informática y Sistemas de Información (CoNaIISI).

(Blas, Gonnet y Leone 2017d) Blas, M., Gonnet, S., & Leone, H. (2017). Modeling user temporal behaviors using hybrid simulation models. *IEEE Latin America Transactions*, 15(2), 341-348.

(Blas y colab. 2018) Blas, M. J., Gonnet, S. M., Leone, H. P., & Zeigler, B. P. (2018). A Conceptual Framework to Classify the Extensions of DEVS Formalism as Variants and Subclasses. In *2018 Winter Simulation Conference* (pp. 560-571). IEEE.

(Boehm y colab. 1978) Boehm, B. W., Brown, J. R., & Lipow, M. (1978). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software Engineering* (pp. 592-605). IEEE Computer Society Press.

(Boer y colab. 2009) De Boer, R. C., Lago, P., Telea, A., & Van Vliet, H. (2009). Ontology-driven visualization of architectural design decisions. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture* (pp. 51-60). IEEE.

(Bogado, Gonnet y Leone 2014) Bogado, V., Gonnet, S., & Leone, H. (2014). Modeling and simulation of software architecture in discrete event system specification for quality evaluation. *Simulation*, 90(3), 290-319.

(Bouanan y colab. 2016) Bouanan, Y., Zacharewicz, G., Vallespir, B., Ribault, J., & Diallo, S. Y. (2016). DEVS based network: modeling and simulation of propagation processes in a multi-layers network. In *Proceedings of the Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems 2016 and Space Simulation for Planetary Space Exploration (SPACE 2016)* (p. 8). Society for Computer Simulation International.

(Breu, Kuntzmann-Combelles, y Felderer 2014) Breu, R., Kuntzmann-Combelles, A., & Felderer, M. (2014). New Perspectives on Software Quality. *IEEE software*, 31(1), 32-38.

(Butler 1994) Butler, J. M. (1994). Quantum modeling of distributed object computing. In *Proceedings of Simulation Symposium* (pp. 175-184). IEEE.

(Callejas-Cuervo, Alarcón-Aldana y Álvarez-Carreño 2017) Callejas-Cuervo, M., Alarcón-Aldana, A. C., & Álvarez-Carreño, A. M. (2017). Modelos de calidad del software, un estado del arte. *Entramado* 13(1), 236-250.

(Card y Glass 1990) Card, D. N., & Glass, R. L. (1990). *Measuring software design quality*. Prentice-Hall, Inc.

(Carvallo y colab. 2004) Carvallo, J. P., Franch, X., Grau, G., & Quer, C. (2004,). QM: a tool for building software quality models. In *Proceedings. 12th IEEE International Requirements Engineering Conference*, 2004. (pp. 358-359). IEEE.

(Castro, Rico y Castro 1995) Castro, E., Rico, L., & Castro, E. (1995). *Estructuras aritméticas elementales y su modelización*.

(Chezzi, Tymoschuk y Lerman 2013) Chezzi, C. M., Tymoschuk, A. R., & Lerman, R. (2013). A method for DEVS simulation of e-commerce processes for integrated business and technology evaluation (WIP). In *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium* (p. 13). Society for Computer Simulation International.

(Chow y Zeigler 1994) Chow, A. C. H., & Zeigler, B. P. (1994). Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In Proceedings of *Winter Simulation Conference* (pp. 716-722). IEEE.

(Clarke y colab. 2003) Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M. & Shepperd, M. (2003). Reformulating software engineering as a search problem. *IEEE Proceedings-software*, 150(3), 161-175.

(CMMI Institute 2018) CMMI Institute. (2018). https://cmmiinstitute.com/.

(Cooke 2010) Cooke, J. (2010). The Shift to Cloud Computing: Forget the Technology, It's About Economics. *White Paper, Cisco Internet Business Solutions Group (IBSG)*.

(Couto, Ribeiro y Campos 2014) Couto, R., Ribeiro, A. N., & Campos, J. C. (2014). Application of ontologies in identifying requirements patterns in use cases.

(Dargan y colab. 2014) Dargan, J. L., Campos-Nanez, E., Fomin, P., & Wasek, J. (2014). Predicting systems performance through requirements quality attributes model. *Procedia Computer Science*, 28, 347-353.

(Deissenboeck y colab. 2009) Deissenboeck, F., Juergens, E., Lochmann, K., & Wagner, S. (2009). Software quality models: Purposes, usage scenarios and requirements. In *2009 ICSE Workshop on Software Quality* (pp. 9-14). IEEE.

(Department of Defense 2004) Department of Defense. (2004). Systems of Systems Engineering. US Department of Defense. Washington. DC.

(Department of Defense 2008) Department of Defense. (2008). Systems Engineering Guide for Systems of Systems. US Department of Defense. Washington. DC.

(Department of Defense 2012) Department of Defense. (2012). Manual for the operation of the joint capabilities integration and development system. US Department of Defense. Washington. DC.

(Dillon, Wu y Chang 2010) Dillon, T., Wu, C., & Chang, E. (2010). Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications* (pp. 27-33). IEEE.

(Dowell y colab. 2011) Dowell, S., Barreto, A., Michael, J. B., & Shing, M. T. (2011). Cloud to cloud interoperability. In *2011 6th International Conference on System of Systems Engineering* (pp. 258-263). IEEE.

(Dromey 1995) Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on software engineering*, 21(2), 146-162.

(Edwards y colab. 1997) Edwards, S., Lavagno, L., Lee, E. A., & Sangiovanni-Vincentelli, A. (1997). Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3), 366-390.

(El-Haik y Shaout 2010) El-Haik, B. S., & Shaout, A. (2011). *Software Design for Six Sigma: A roadmap for excellence.* John Wiley & Sons.

(Fan y colab. 2015) Fan, H., Hussain, F. K., Younas, M., & Hussain, O. K. (2015). An integrated personalization framework for SaaS-based cloud services. *Future Generation Computer Systems*, 53, 157-173.

(Fang y DeLaurentis 2014) Fang, Z., & DeLaurentis, D. (2014). Dynamic planning of system of systems architecture evolution. *Procedia Computer Science*, 28, 449-456.

(Fehling y colab. 2014) Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). *Cloud computing patterns: fundamentals to design, build, and manage cloud applications.* Springer Science & Business Media.

(Fenton y Bieman 2014) Fenton, N., & Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC press.

(Fonseca 2008) Fonseca, B. (2008). SaaS benefits starting to outweigh risks. *Computerworld*, 42(19), 12.

(Foster y colab. 2008) Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008). Cloud computing and grid computing 360-degree compared.

(Franch y colab. 2010) Franch, X., Palomares, C., Quer, C., Renault, S., & De Lazzer, F. (2010). A metamodel for software requirement patterns. In *International Working Conference on Requirements Engineering: Foundation for Software Quality* (pp. 85-90). Springer, Berlin, Heidelberg.

(Fukuzawa y Saeki 2002) Fukuzawa, K., & Saeki, M. (2002). Evaluating software architectures by coloured petri nets. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering* (pp. 263-270). ACM.

(Gao y colab. 2011) Gao, J., Pattabhiraman, P., Bai, X., & Tsai, W. T. (2011). SaaS performance and scalability evaluation in clouds. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)* (pp. 61-71). IEEE.

(García y colab. 2006) García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruiz, F., Piattini, M., & Genero, M. (2006). Towards a consistent terminology for software measurement. *Information and Software Technology*, 48(8), 631-644.

(García-Peñalvo y colab. 2012) García-Peñalvo, F. J., Colomo-Palacios, R., García, J., & Therón, R. (2012). Towards an ontology modeling tool. A validation in software engineering scenarios. *Expert Systems with Applications*, 39(13), 11468-11478.

(Garlan y Shaw 1994) Garlan, D., & Shaw, M. (1993). An introduction to software architecture. In *Advances in software engineering and knowledge engineering* (pp. 1-39).

(Gholami y colab. 2014) Gholami, S., Sarjoughian, H. S., Godding, G. W., Peters, D. R., & Chang, V. (2014). Developing composed simulation and optimization models using actual supply-demand network datasets. In Proceedings of the *Winter Simulation Conference* (pp. 2510-2521). IEEE.

(Gómez-Pérez, Fernandez-Lopez y Corcho 2010) Gomez-Perez, A., Fernández-López, M., & Corcho, O. (2006). *Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer Science & Business Media.

(Gonnet, Henning y Leone 2007) Gonnet, S., Henning, G., & Leone, H. (2007). A model for capturing and representing the engineering design process. *Expert Systems with Applications*, 33(4), 881-902.

(Gonzalez y colab. 2012) Gonzalez, N., Miers, C., Redigolo, F., Simplicio, M., Carvalho, T., Näslund, M., & Pourzandi, M. (2012). A quantitative analysis of current security concerns and solutions for cloud computing. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1), 11.

(Gonçalves 2016) Gonçalves, M. B. (2016). Supporting architectural design of acknowledged Software-intensive Systems-of-Systems (Université de Bretagne Sud).

(Graaf y colab. 2014) Graaf, K. A., Liang, P., Tang, A., van Hage, W. R., & van Vliet, H. (2014). An exploratory study on ontology engineering for software architecture documentation. *Computers in Industry*, 65(7), 1053-1064.

(Gronback 2009) Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education.

(Gruber 1993) Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2), 199-220.

(Grüninger y Fox 1995) Grüninger, M., & Fox, M. S. (1995). Methodology for the design and evaluation of ontologies.

(Gupta, Seetharaman y Raj 2013) Gupta, P., Seetharaman, A., & Raj, J. R. (2013). The usage and adoption of cloud computing by small and medium businesses. *International Journal of Information Management*, 33(5), 861-874.

(Hadzic y colab. 2009) Hadzic, M., Chang, E., Dillon, T., & Wongthongtham, P. (2009). *Ontology-based multi-agent systems*. Germany: Springer Berlin Heidelberg.

(Hamri, Giambiasi y Frydman 2006) Hamri, M. E. A., Giambiasi, N., & Frydman, C. (2006). Min–Max-DEVS modeling and simulation. *Simulation Modelling Practice and Theory*, 14(7), 909-929.

(Harter, Krishnan y Slaughter 2000) Harter, D. E., Krishnan, M. S., & Slaughter, S. A. (2000). Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4), 451-466.

(Henderson-Sellers 2011) Henderson-Sellers, B. (2011). Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software*, 84(2), 301-313.

(Hild, Sarjoughian y Zeigler 2002) Hild, D. R., Sarjoughian, H. S., & Zeigler, B. P. (2002). DEVS-DOC: a modeling and simulation environment enabling distributed codesign. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 32(1), 78-92.

(Hlavacs y Kotsis 1999) Hlavacs, H., & Kotsis, G. (1999). Modeling user behavior: A layered approach. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (pp. 218-225). IEEE.

(Hlavacs, Hotop y Kotsis 2000) Hlavacs, H., Hotop, E., & Kotsis, G. (2000). Workload generation by modeling user behavior. *Proceedings of OPNETWORKS 2000*.

(Hong y colab. 1997) Hong, J. S., Song, H. S., Kim, T. G., & Park, K. H. (1997). A real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems*, 7(4), 355-375.

(Hu y colab. 2011) Hu, F., Qiu, M., Li, J., Grant, T., Taylor, D., McCaleb, S. & Hamner, R. (2011). A review on cloud computing: Design challenges in architecture and security. *Journal of computing and information technology*, 19(1), 25-55.

(IEEE 1061 1998) IEEE 1061. (1998). Software Quality Metrics Methodology: Description. Institute of Electrical Electronic Engineering.

(Immonen y Niemelä 2008) Immonen, A., & Niemelä, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software & Systems Modeling*, 7(1), 49.

(ISO/IEC 9126-1 2001) ISO/IEC 9126-1. (2001). Software Engineering - Product Quality - Part 1: Quality Model. Institute of Electrical Electronic Engineering.

(ISO/IEC TR 9126-2 2003) ISO/IEC TR 9126-2. (2003). Software Engineering - Product Quality - Part 2: External Metrics. Institute of Electrical Electronic Engineering.

(ISO/IEC TR 9126-3 2003) ISO/IEC TR 9126-3. (2003). Software Engineering - Product Quality - Part 3: Internal Metrics. Institute of Electrical Electronic Engineering.

(ISO/IEC 14598 1999) ISO/IEC 14598. (1999). Software Engineering - Product Evaluation. Institute of Electrical Electronic Engineering.

(ISO/IEC 15504 2004) ISO/IEC 15504. (2004). Information Technology: Process Assessment. Institute of Electrical Electronic Engineering.

(ISO/IEC 25000 2014) ISO/IEC 25000. (2014). Systems and Software Engineering; Systems and software Quality Requirements and Evaluation (SQuaRE): Guide to SQuaRE. Institute of Electrical Electronic Engineering.

(ISO/IEC 25001 2014) ISO/IEC 25001. (2014). Systems and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Planning and Management. Institute of Electrical Electronic Engineering.

(ISO/IEC 25010 2011) ISO/IEC 25010. (2011). Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Institute of Electrical Electronic Engineering.

(ISO/IEC TS 25011 2017) ISO/IEC TS 25011. (2017). Information Technology - Systems and software Quality Requirements and Evaluation (SQuaRE) - Service Quality Models. Institute of Electrical Electronic Engineering.

(ISO/IEC 25012 2008) ISO/IEC 25012. (2008). Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) -- Data Quality Model.

(ISO/IEC 25020 2007) ISO/IEC 25020. (2007). Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) - Measurement Reference Model and Guide. Institute of Electrical Electronic Engineering.

(ISO/IEC 25021 2012) ISO/IEC 25021. (2012). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Quality Measure Elements. Institute of Electrical Electronic Engineering.

(ISO/IEC 25022 2016) ISO/IEC 25022. (2016). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Measurement of Quality in Use. Institute of Electrical Electronic Engineering.

(ISO/IEC 25023 2016) ISO/IEC 25023. (2016). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Measurement of System and Software Product Quality. Institute of Electrical Electronic Engineering.

(ISO/IEC 25024 2015) ISO/IEC 25024. (2015). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Measurement of Data Quality. Institute of Electrical Electronic Engineering.

(ISO/IEC 25030 2007) ISO/IEC 25030. (2007). Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) - Quality Requirements.

(ISO/IEC 25040 2011) ISO/IEC 25040. (2011). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Evaluation Process. Institute of Electrical Electronic Engineering.

(ISO/IEC 25041 2012) ISO/IEC 25041. (2012). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Evaluation Guide for Developers, Acquirers and Independent Evaluators.

(ISO/IEC 25045 2010) ISO/IEC 25045. (2010). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Evaluation Module for Recoverability. Institute of Electrical Electronic Engineering.

(ISO/IEC 25051 2014) ISO/IEC 25051. (2014). Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Requirements for Quality of Ready to Use Software Product (RUSP) and Instructions for Testing. Institute of Electrical Electronic Engineering.

(ISO/IEC 25062 2006) ISO/IEC 25062. (2006). Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) - Common Industry Format (CIF) for Usability Test Reports. Institute of Electrical Electronic Engineering.

(ISO/IEC 25063 2014) ISO/IEC 25063. (2014). Systems and Software Engineering - Systems and Software Product Quality Requirements and Evaluation (SQuaRE) - Common Industry Format (CIF) for Usability: Context of Use Description.

(ISO/IEC 25064 2013) ISO/IEC 25064. (2013). Systems and Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) - Common Industry Format (CIF) for Usability: User Needs Report. Institute of Electrical Electronic Engineering.

(ISO/IEC 25066 2016) ISO/IEC 25066. (2016). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Common Industry Format (CIF) for Usability: Evaluation Report. Institute of Electrical Electronic Engineering.

(ISO/IEC TR 25060 2010) ISO/IEC TR 25060. (2010). Systems and Software Engineering - Systems and Software Product Quality Requirements and Evaluation (SQuaRE) - Common Industry Format (CIF) for Usability: General Framework for Usability-Related information. Institute of Electrical Electronic Engineering.

(ISO/IEC 33000 2015) ISO/IEC 33000. (2015). Information Technology: Process Assessment. Institute of Electrical Electronic Engineering.

(ISO/IEC/IEEE 24765 2010) ISO/IEC/IEEE 24765. (2010). ISO/IEC/IEEE 24765:2010 - Systems and software engineering -- Vocabulary. Institute of Electrical Electronic Engineering.

(Jacobson 1999) Jacobson, I. (1999). *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc.

(Jansen y Bosch 2005) Jansen, A., & Bosch, J. (2005). Software architecture as a set of architectural design decisions. In *5th Working IEEE/IFIP Conference on Software Architecture* (pp. 109-120). IEEE.

(Jwo y Cheng 2010) Jwo, J., & Cheng, Y. (2010). Pseudo software: A mediating instrument for modeling software requirements. *Journal of Systems and Software*, 83(4), 599-608.

(Kan 2003) Kan, S. H. (2002). Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc.

(Karnouskos y Colombo 2011) Karnouskos, S., & Colombo, A. W. (2011). Architecting the next generation of service-based SCADA/DCS system of systems. In *IECON 2011-37th Annual Conference of the IEEE Industrial Electronics Society* (pp. 359-364). IEEE.

(Kashkoush y ElMaraghy 2017) Kashkoush, M., & ElMaraghy, H. (2017). An integer programming model for discovering associations between manufacturing system capabilities and product features. *Journal of Intelligent Manufacturing*, 28(4), 1031-1044.

(Kazman y colab. 1996) Kazman, R., Abowd, G., Bass, L., & Clements, P. (1996). Scenario-based analysis of software architecture. *IEEE software*, 13(6), 47-55.

(Kazman y colab. 1998) Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998). The architecture tradeoff analysis method. In *Proceedings of the 4º IEEE International Conference on Engineering of Complex Computer Systems* (pp. 68-78). IEEE.

(Kazman, Asundi y Klien 2002) Kazman, R., Asundi, J., & Klien, M. (2002). *Making architecture design decisions: An economic approach* (No. CMU/SEI-2002-TR-035). Carnegie-Mellon University.

(Kazman, Gagliardi y Wood 2012) Kazman, R., Gagliardi, M., & Wood, W. (2012). Scaling up software architecture analysis. *Journal Systems and Software*, 85(7), 1511-1519.

(Kazman, Klein y Clements 2000) Kazman, R., Klein, M., & Clements, P. (2000). *ATAM: Method for architecture evaluation* (No. CMU/SEI-2000-TR-004). Carnegie-Mellon Univ.

(Khan y Singh 2012) Khan, I. A., & Singh, R. (2012). Quality assurance and integration testing aspects in web based applications.

(Khan y colab. 2013) Khan, A. N., Kiah, M. M., Khan, S. U., & Madani, S. A. (2013). Towards secure mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(5), 1278-1299.

(Khoshgoftaar, Liu y Seliya 2004) Khoshgoftaar, T. M., Liu, Y., & Seliya, N. (2004). A multiobjective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation*, 8(6), 593-608.

(Kim, Kim y Park 1998) Kim, K. H., Kim, T. G., & Park, K. H. (1998). Hierarchical partitioning algorithm for optimistic distributed simulation of DEVS models. *Journal of Systems Architecture*, 44(6-7), 433-455.

(Kim y colab. 2015) Kim, Y. J., Yang, J. Y., Kim, Y. M., Lee, J., & Choi, C. (2015). Modeling Behavior of Mobile Application Using Discrete Event System Formalism. In *Asian Simulation Conference* (pp. 40-48). Springer, Singapore.

(Kitchenham y Pfleeger 1996) Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: the elusive target [special issues section]. *IEEE software*, 13(1), 12-21.

(Kitchenham, Hughes y Linkman 2001) Kitchenham, B. A., Hughes, R. T., & Linkman, S. G. (2001). Modeling software measurement data. *IEEE Transactions on Software Engineering*, 27(9), 788-804.

(Kläs, Lampasona y Münch 2013) Klas, M., Lampasona, C., & Munch, J. (2011). Adapting software quality models: Practical challenges, approach, and first empirical results. In *37th Conference on Software Engineering and Advanced Applications* (pp. 341-348). IEEE.

(Knublauch y colab. 2005) Knublauch, H., Horridge, M., Musen, M. A., Rector, A. L., Stevens, R., Drummond, N. & Wang, H. (2005). The Protege OWL Experience.

(Koziolek y colab. 2008) Koziolek, H., Becker, S., Reussner, R., & Happe, J. (2009). Evaluating performance of software architecture models with the Palladio component model. In *Software Applications: Concepts, Methodologies, Tools, and Applications* (pp. 1111-1134).

(Kruchten 2004) Kruchten, P. (2004). An ontology of architectural design decisions in software intensive systems. In 2nd Groningen workshop on software variability (pp. 54-61).

(Kruchten, Capilla y Dueas 2009) Kruchten, P., Capilla, R., & Dueñas, J. C. (2009). The decision view's role in software architecture practice. *IEEE software*, 26(2), 36-42.

(Kurz, Hlavacs y Kotsis 2001) Kurz, C., Hlavacs, H., & Kotsis, G. (2001). Workload Generation by Modelling User Behaviour in an ISP Subnet.

(Kwon y colab. 1996) Kwon, Y., Park, H., Jung, S., & Kim, T. (1996). Fuzzy-DEVS formalism: concepts, realization and applications. In *Proceedings AIS* (pp. 227-234).

(Law y Kelton 1991) Law, A. M., & Kelton, W. D. (1991). *Simulation modeling and analysis*. New York: McGraw-Hill.

(Lee, Lee y Kim 2009) Lee, J. Y., Lee, J. W., & Kim, S. D. (2009). A quality model for evaluating software-as-a-service in cloud computing. In *7º ACIS international conference on software engineering research, management and applications* (pp. 261-266). IEEE.

(Lewis 2010) Lewis, G. (2010). Basics about cloud computing. Carniege-Mellon University.

(Liu 2010) Liu, Q. (2010). *Algorithms for parallel simulation of large-scale DEVS and Cell-DEVS models* (Doctoral dissertation, Carleton University).

(Liu y Wainer 2010) Liu, Q., & Wainer, G. (2010). Accelerating large-scale DEVS-based simulation on the cell processor. In *Proceedings of the 2010 Spring Simulation Multiconference* (p. 124). Society for Computer Simulation International.

(Lynch 2008) Lynch, M. (2008). The Cloud Wars: $100+ billion at stake.

(Low, Chen y Wu 2011) Low, C., Chen, Y., & Wu, M. (2011). Understanding the determinants of cloud computing adoption. *Industrial management & data systems*, 111(7), 1006-1023.

(McCall, Richards, y Walters 1977) McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality. Concept and definitions of software quality: final technical report*. National Technical Information Service (Report nr. RADC-TR-77-369).

(Magalhães y colab. 2015) Magalhães, D., Calheiros, R. N., Buyya, R., & Gomes, D. G. (2015). Workload modeling for resource usage analysis and simulation in cloud computing. *Computers & Electrical Engineering*, 47, 69-81.

(Maier 1996) Maier, M. W. (1998). Architecting principles for systems-of-systems. *Systems Engineering: Journal of the International Council on Systems Engineering*, 1(4), 267-284.

(Mather, Kumaraswamy y Latif 2009) Mather, T., Kumaraswamy, S., & Latif, S. (2009). *Cloud security and privacy: an enterprise perspective on risks and compliance*. O'Reilly Media, Inc.

(Meiappane, Chithra y Venkataesan 2013) Meiappane, A., Chithra, B., & Venkataesan, P. (2013). Evaluation of software architecture quality attribute for an internet banking system.

(Mell y Grance 2011) Mell, P., & Grance, T. (2011). The NIST definition of cloud computing.

(Modi y colab. 2012) Modi, C., Patel, D., Borisaniya, B., Patel, A., & Rajarajan, M. (2013). A survey on security issues and solutions at different layers of Cloud computing. *The journal of supercomputing*, 63(2), 561-592.

(Moallemi y colab. 2010) Moallemi, M., Wainer, G., & Awad, A. (2010). Application of RT-DEVS in Military. In *Proceedings of the 2010 Spring Simulation Multiconference* (p. 29). Society for Computer Simulation International.

(Monroe y colab. 1997) Monroe, R. T., Kompanek, A., Melton, R., & Garlan, D. (1997). Architectural styles, design patterns, and objects. *IEEE software*, 14(1), 43-52.

(Muñoz, Velthuis y Moraga de la Rubia 2010) Muñoz, C. C., Velthuis, M. G. P., & de la Rubia, M. Á. M. (2010). *Calidad del producto y proceso software*. Editorial Ra-Ma.

(Musa y Alkhateeb 2013) Musa, K., & Alkhateeb, J. (2013). Quality model based on cots quality attributes. *International Journal of Software Engineering & Applications*, 4(1), 1.

(Muzy y Nutaro 2005) Muzy, A., & Nutaro, J. J. (2005). Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling & Simulation* (pp. 273-279).

(Myllymäki, Koskimies y Mikkonen 2002) Myllymäki, T., Koskimies, K., & Mikkonen, T. (2002). Structuring product-lines: A layered architectural style. In *International Conference on Object-Oriented Information Systems* (pp. 482-487). Springer, Berlin, Heidelberg.

(Nakagawa y colab. 2013) Nakagawa, E. Y., Gonçalves, M., Guessi, M., Oliveira, L. B., & Oquendo, F. (2013). The state of the art and future perspectives in systems of systems software architectures. In *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems* (pp. 13-20). ACM.

(Naone 2009) Naone, E. (2009). Technology overview conjuring clouds.

(Nielsen y colab. 2015) Nielsen, C. B., Larsen, P. G., Fitzgerald, J., Woodcock, J., & Peleska, J. (2015). Systems of systems engineering: basic concepts, model-based techniques, and research directions. *ACM Computing Surveys*, 48(2), 18.

(Nissen y colab. 1996) Nissen, H., Jeusfeld, M., Jarke, M., Zemanek, G., & Huber, H. (1996). *Managing applications requirements perspectives with metamodels*. IEEE Software, 13(2), 37-48.

(Noy y McGuinness 2001) Noy, N. F., & McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.

(Olsina y Martín 2003) Olsina, L., & Martin, M. (2004). Ontology for software metrics and indicators. *Journal of Web Engineering*, 2(4), 262-281.

(Orgun y Meyer 2008) Orgun, M. A., & Meyer, T. (2008). Introduction to the special issue on advances in ontologies. *Expert Systems*, 25(3), 175-178.

(Pahl, Giesecke y Hasselbring 2009) Pahl, C., Giesecke, S., & Hasselbring, W. (2009). Ontology-based modelling of architectural styles. *Information and Software Technology*, 51(12), 1739-1749.

(Pires y colab. 2011) Pires, P. F., Delicato, F. C., Cóbe, R., Batista, T., Davis, J. G., & Song, J. H. (2011). Integrating ontologies, model driven, and CNL in a multi-viewed approach for requirements engineering. *Requirements Engineering*, 16(2), 133-160.

(Pressman 2010) Pressman, R. S. (2010). *Software engineering: a practitioner's approach*. McGraw-Hill.

(Reese 2009) Reese, G. (2009). *Cloud application architectures: building applications and infrastructure in the cloud.* O'Reilly Media, Inc.

(Reinhartz-Berger, Sturm y Wand 2013) Reinhartz-Berger, I., Sturm, A., & Wand, Y. (2013). Comparing functionality of software systems: An ontological approach. *Data & Knowledge Engineering*, 87, 320-338.

(Rijgersberg, Wigham y Top 2011) Rijgersberg, H., Wigham, M., & Top, J. L. (2011). How semantics can improve engineering processes: A case of units of measure and quantities. *Advanced Engineering Informatics*, 25(2), 276-287.

(Rimal, Choi y Lumb 2009) Rimal, B. P., Choi, E., & Lumb, I. (2009). A taxonomy and survey of cloud computing systems. In *Fifth International Joint Conference on INC, IMS and IDC* (pp. 44-51). IEEE.

(Roldán, Gonnet y Leone 2013) Roldán, M. L., Gonnet, S., & Leone, H. (2013). Knowledge representation of the software architecture design process based on situation calculus. *Expert Systems*, 30(1), 34-53.

(Roldán, Gonnet y Leone 2016) Roldán, M. L., Gonnet, S., & Leone, H. (2016). Operation-based approach for documenting software architecture knowledge. *Expert Systems*, 33(4), 313-348.

(Roshandel, Medvidovic y Golubchik 2007) Roshandel, R., Medvidovic, N., & Golubchik, L. (2007). A Bayesian model for predicting reliability of software systems at the architectural level. In *International Conference on the Quality of Software Architectures* (pp. 108-126). Springer, Berlin, Heidelberg.

(Roussey y colab. 2011) Roussey, C., Pinet, F., Kang, M. A., & Corcho, O. (2011). An introduction to ontologies and ontology engineering. In *Ontologies in urban development projects* (pp. 9-38). Springer, London.

(Scalone 2006) Scalone, F. (2006). Estudio comparativo de los modelos y estándares de calidad del software. Universidad de Buenos Aires.

(Shang y Wainer 2006) Shang, H., & Wainer, G. (2006). A simulation algorithm for dynamic structure DEVS modeling. In *Proceedings of the 2006 Winter Simulation Conference* (pp. 815-822). IEEE.

(Silva y colab. 2015) Silva, A. C., Verdúm, J. C., Espinoza, M. A., Hurtado, D. J., & Poma, A. (2015). Modelo de calidad de servicio QoS en entornos Cloud. *International Journal of Information Systems and Software Engineering for Big Companies*, 2(2), 70-80.

(Singh, Bhagat y Kumar 2012) Singh, R., Bhagat, A., & Kumar, N. (2012). Generalization of Software Metrics on Software as a Service (SaaS). In *2012 International Conference on Computing Sciences* (pp. 267-270). IEEE.

(Singh, Tripathi y Vinod 2011) Singh, L. K., Tripathi, A. K., & Vinod, G. (2011). Software reliability early prediction in architectural design phase: Overview and Limitations. *Journal of Software Engineering and Applications*, 4(03), 181.

(Sommerville 2005) Sommerville, I. (2005). *Ingeniería del software*. Pearson Educación.

(Spitznagel y Garlan 1998) Spitznagel, B., & Garlan, D. (1998). Architecture-based performance analysis. En *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, 146-51. San Francisco, California.

(Steinberg y colab. 2008) Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

(Steiniger y Uhrmacher 2016) Steiniger, A., & Uhrmacher, A. M. (2016). Intensional couplings in variable-structure models: An exploration based on multilevel-DEVS. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(2), 9.

(Strowd y Lewis 2010) Strowd, H. D., & Lewis, G. A. (2010). *T-check in system-of-systems technologies: Cloud computing* (No. CMU/SEI-2010-TN-009). Carnegie-Mellon University.

(Suman y Rohtak 2014) Suman, M. W., & Rohtak, M. D. U. (2014). A comparative study of software quality models. *International Journal of Computer Science and Information Technologies*, 5(4), 5634-5638.

(Tang y Han 2005) Tang, A., & Han, J. (2005). Architecture rationalization: a methodology for architecture verifiability, traceability and completeness. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (pp. 135-144). IEEE.

(Tolvanen y Rossi 2003) Tolvanen, J. P., & Rossi, M. (2003). MetaEdit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 92-93). ACM.

(Tyree y Akerman 2005) Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE software*, 22(2), 19-27.

(Uhrmacher y Kuttler 2006) Uhrmacher, A., & Kuttler, C. (2006). Multi-Level Modeling in Systems Biology by Discrete Event Approaches *It-Information Technology*, 48(3), 148-153.

(Uhrmacher y colab. 2007) Uhrmacher, A. M., Ewald, R., John, M., Maus, C., Jeschke, M., & Biermann, S. (2007). Combining micro and macro-modeling in DEVS for computational biology. In *Proceedings of the 39th Winter Simulation Conference* (pp. 871-880). IEEE Press.

(Vegetti, Leone y Henning 2011) Vegetti, M., Leone, H., & Henning, G. (2011). PRONTO: An ontology for comprehensive and consistent representation of product information. *Engineering Applications of Artificial Intelligence*, 24(8), 1305-1327.

(Vegetti y colab. 2016) Vegetti, M., Roldán, L., Gonnet, S., Leone, H., & Henning, G. (2016). A framework to represent, capture, and trace ontology development processes. *Engineering Applications of Artificial Intelligence*, 56, 230-249.

(W3C 2004) W3C (2004). SWRL: A Semantic Web Rule Language.

(W3C 2012) W3C (2012). OWL 2 Web Ontology Language.

(W3C 2013) W3C (2013). SPARQL 1.1 Query Language.

(Wagner 2013) Wagner, S. (2013). *Software product quality control*. Heidelberg: Springer.

(Wainer 2004) Wainer, G. A. (2004). Modeling and simulation of complex systems with Cell-DEVS. In *Proceedings of the 36th Winter Simulation Conference* (pp. 49-60).

(Wainer y Madhoun 2005) Wainer, G., & Madhoun, R. (2005). Creating spatially-shaped defense models using DEVS and Cell-DEVS. *The Journal of Defense Modeling and Simulation,* 2(3), 121-143.

(Wainer y Mosterman 2010) Wainer, G., & Mosterman, P. (2010). *Discrete-Event Modeling and Simulation: Theory and Applications*. CRC Press.

(Wang, Wu y Chen 1999) Wang, W. L., Wu, Y., & Chen, M. H. (1999). An architecture-based software reliability model. In *Proceedings 1999 Pacific Rim International Symposium on Dependable Computing* (pp. 143-150). IEEE.

(Wen y Dong 2013) Wen, P. X., & Dong, L. (2013, September). Quality model for evaluating SaaS service. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies* (pp. 83-87). IEEE.

(Wolf 1994) Wolf, W. H. (1994). Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7), 967-989.

(Yeh, Lee y Pai 2012) Yeh, C. H., Lee, G. G., & Pai, J. C. (2012). How information system capability affects e-business information technology strategy implementation: An empirical study in Taiwan. *Business Process Management Journal*, 18(2), 197-218.

(Zalewski y Kijas 2013) Zalewski, A., & Kijas, S. (2013). Beyond ATAM: Early architecture evaluation method for large-scale distributed systems. *Journal of Systems and Software*, 86(3), 683-697.

(Zehe y colab. 2015) Zehe, D., Cai, W., Knoll, A., & Aydt, H. (2015). Tutorial on a modeling and simulation cloud service. In *2015 Winter Simulation Conference* (pp. 103-114).

(Zeigler 1976) Zeigler, Bernard P. (1976). Theory of Modelling and Simulation.

(Zeigler 2018) Zeigler, B. P. (2018). Closure under coupling: concept, proofs, DEVS recent examples (wip). In *Proceedings of the Theory of Modeling and Simulation Symposium* (p. 7). Society for Computer Simulation International.

(Zeigler y Sarjoughian 2013) Zeigler, B. P., Sarjoughian, H. S., Duboz, R., & Soulie, J. (2012). *Guide to modeling and simulation of systems of systems (simulation foundations, methods and applications)*. London: Spring-Verlag.

(Zeigler, Muzy y Kofman 2018) Zeigler, B. P., Muzy, A., & Kofman, E. (2018). *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations.* Academic Press.

(Zeigler, Praehofer y Kim 2000) Zeigler, B. P., Kim, T. G., & Praehofer, H. (2000). *Theory of modeling and simulation: Integrating Discrete Event and Continous Complex Dynamic Systems*. Academic press.

(Zeist y Hendriks 1996) Zeist, R. H. J., & Hendriks, P. R. H. (1996). Specifying software quality with the extended ISO model. *Software Quality Journal*, 5(4), 273-284.

(Zhang y Zhou 2009) Zhang, L. J., & Zhou, Q. (2009). CCOA: Cloud computing open architecture. In *2009 IEEE International Conference on Web Services* (pp. 607-616). IEEE.

(Zhang, Cheng y Boutaba 2010) Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1), 7-18.

(Zhao y Zhu 2014) Zhao, B., & Zhu, Y. (2014). Formalizing and validating the web quality model for web source quality evaluation. *Expert Systems with Applications*, 41(7), 3306-3312.

(Zissis y Lekkas 2012) Zissis, D., & Lekkas, D. (2012). Addressing cloud computing security issues. *Future Generation computer systems*, 28(3), 583-592.

(Zio y Sansavini 2013) Zio, E., & Sansavini, G. (2013). Vulnerability of smart grids with variable generation and consumption: A system of systems perspective. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 43(3), 477-487.

Apéndice A. Consultas sobre la Ontología QSO

En este apéndice se detallan las consultas SPARQL especificadas a fin de implementar cada una de las preguntas de competencia planteadas para la ontología de esquemas de calidad. Los identificadores utilizados se corresponden a los detallados en el capítulo donde se describe la ontología QSO (*Capítulo 5*).

Tabla A.1. Documentación de la pregunta de competencia Q1.

ID	Q1
PREGUNTA	¿Qué métricas son útiles para evaluar la característica de calidad 'X'?
RESPUESTA	Un conjunto de instancias del concepto "Métrica" identificadas por su nombre y propósito.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX s: <http://www.finalontology.org#> PREFIX q: <http://www.qualitymodel.org#> PREFIX m: <http://www.metric.org#> PREFIX sw: <http://www.softwaremodel.org#> SELECT DISTINCT **?name ?purpose ?metricindividual** WHERE { ?characteristic rdf:type ?type . ?characteristic q:contains ?subcharacteristic . ?attribute sw:isDescribedBy ?subcharacteristic . ?attribute sw:isMeasuredBy ?metricindividual . **?metricindividual** m:metricName **?name** . **?metricindividual** m:purpose **?purpose** . FILTER(regex(str(?type),"X")) }
VARIABLES DE SALIDA	"?name", "?purpose" y "?metricindividual".

Tabla A.2. Documentación de la pregunta de competencia Q2.

ID	Q2
PREGUNTA	¿Qué métricas son útiles para evaluar la subcaracterística de calidad 'Y'?
RESPUESTA	Un conjunto de instancias del concepto "Métrica" identificadas por su nombre y propósito.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX s: <http://www.finalontology.org#> PREFIX q: <http://www.qualitymodel.org#> PREFIX m: <http://www.metric.org#> PREFIX sw: <http://www.softwaremodel.org#> SELECT DISTINCT **?name ?purpose ?metricindividual** WHERE { ?subcharacteristic rdf:type ?type . ?attribute sw:isDescribedBy ?subcharacteristic . ?attribute sw:isMeasuredBy ?metricindividual . **?metricindividual** m:metricName **?name** . **?metricindividual** m:purpose **?purpose** . FILTER(regex(str(?type),"Y")) }
VARIABLES DE SALIDA	"?name", "?purpose" y "?metricindividual".

Tabla A.3. Documentación de la pregunta de competencia Q3.

ID	Q3
PREGUNTA	¿Cuántas métricas distintas se asocian a la característica de calidad 'X'?

RESPUESTA	Un valor entero que indica la cantidad de métricas asociadas al elemento.
CONSULTA SPARQL	
VARIABLES DE SALIDA	-

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://www.finalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT (COUNT(*) AS ?quantity)
{
      SELECT DISTINCT ?metricindividual
      WHERE
      {
            ?characteristic rdf:type ?type .
            ?characteristic q:contains ?subcharacteristic .
            ?attribute sw:isDescribedBy ?subcharacteristic .
            ?attribute sw:isMeasuredBy ?metricindividual .
            FILTER(regex(str(?type),"X"))
      }
}
```

Tabla A.4. Documentación de la pregunta de competencia Q4.

ID	Q4

PREGUNTA	¿Cuántas métricas distintas se asocian a la subcaracterística de calidad 'Y'?
RESPUESTA	Un valor entero que indica la cantidad de métricas asociadas al elemento.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX owl: <http://www.w3.org/2002/07/owl#>
	PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
	PREFIX s: <http://www.finalontology.org#>
	PREFIX q: <http://www.qualitymodel.org#>
	PREFIX m: <http://www.metric.org#>
	PREFIX sw: <http://www.softwaremodel.org#>
	SELECT (COUNT(*) AS ?quantity)
	{
	SELECT DISTINCT ?metricindividual
	WHERE
	{
	?subcharacteristic rdf:type ?type .
	?attribute sw:isDescribedBy ?subcharacteristic .
	?attribute sw:isMeasuredBy ?metricindividual .
	FILTER(regex(str(?type),"Y"))
	}
	}

VARIABLES DE SALIDA	-

Tabla A.5. Documentación de la pregunta de competencia Q5.

ID	Q5
PREGUNTA	¿A cuál característica de calidad se encuentra asociada la métrica 'Z'?
RESPUESTA	El nombre de una o más instancias de los conceptos que representan características.
CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX s: <http://wwwfinalontology.org#> PREFIX q: <http://www.qualitymodel.org#> PREFIX m: <http://www.metric.org#> PREFIX sw: <http://www.softwaremodel.org#> SELECT DISTINCT **?characteristic** WHERE { ?metricindividual rdf:type m:Metric . ?metricindividual m:metricName ?metricname . ?attribute sw:isDescribedBy ?subcharacteristic . ?attribute sw:isMeasuredBy ?metricindividual . ?subcharacteristic rdf:type q:Subcharacteristic . ?subcharacteristic q:belongs ?characteristicindividual . ?characteristicindividual rdf:type q:Characteristic . ?characteristicindividual rdf:type **?characteristic** . FILTER (?metricname = "Z" && (regex(str(**?characteristic**),"*Compatibility*") || regex(str(**?characteristic**),"*FunctionalSuitability*") || regex(str(**?characteristic**),"*Maintainability*") ||

	regex(str(**?characteristic**),"*PerformanceEfficiency*") || regex(str(**?characteristic**),"*Portability*") || regex(str(**?characteristic**),"*Reliability*") || regex(str(**?characteristic**),"*Security*") || regex(str(**?characteristic**),"*Usability*"))) }
VARIABLES DE SALIDA	"?characteristic".

Tabla A.6. Documentación de la pregunta de competencia Q6.

ID	Q6
PREGUNTA	¿Cuántas veces se asocia la métrica 'Z' con la característica de calidad 'X'?

RESPUESTA	Un valor entero que indica la cantidad de veces que se han asociado los elementos.
CONSULTA SPARQL	
VARIABLES DE SALIDA	-

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://wwwfinalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT (COUNT(*) AS ?quantity)
{
 SELECT ?individual
 WHERE
 {
     ?metricindividual rdf:type m:Metric .
     ?metricindividual m:metricName ?metricname .
     ?attribute sw:isDescribedBy ?subcharacteristic .
     ?attribute sw:isMeasuredBy ?metricindividual .
     ?subcharacteristic rdf:type q:Subcharacteristic .
     ?subcharacteristic q:belongs ?individual .
     ?individual rdf:type q:Characteristic .
     ?individual rdf:type ?characteristic .
     FILTER (?metricname = "Z" &&
     regex(str(?characteristic),"X") )
 }
}
```

Tabla A.7. Documentación de la pregunta de competencia Q7.

ID	Q7
PREGUNTA	¿A qué característica de calidad es usualmente asociada la métrica 'Z'?
RESPUESTA	El nombre de la instancia del concepto que representa la característica buscada.
CONSULTA SPARQL	

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://wwwfinalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT ?characteristic {
   SELECT ?characteristic
   (COUNT(?characteristicindividual) AS ?quantity)
   {
      SELECT ?characteristic ?individual
      WHERE
      {
        ?metricindividual rdf:type m:Metric .
        ?metricindividual m:metricName ?metricname .
        ?attribute sw:isDescribedBy ?subcharacteristic .
        ?attribute sw:isMeasuredBy ?metricindividual .
        ?subcharacteristic rdf:type q:Subcharacteristic .
        ?subcharacteristic q:belongs ?individual.
        ?individual rdf:type q:Characteristic .
        ?individual rdf:type ?characteristic .
        FILTER (?metricname = "Z" &&
        (regex(str(?characteristic),"Compatibility") ||
        regex(str(?characteristic),"FunctionalSuitability")||
```

	regex(str(**?characteristic**),"*Maintainability*") || regex(str(**?characteristic**),"*PerformanceEfficiency*")|| regex(str(**?characteristic**),"*Portability*") || regex(str(**?characteristic**),"*Reliability*") || regex(str(**?characteristic**),"*Security*") || regex(str(**?characteristic**),"*Usability*"))) } } GROUP BY **?characteristic** ORDER BY DESC(?quantity) LIMIT 1 }
VARIABLES DE SALIDA	"?characteristic".

Tabla A.8. Documentación de la pregunta de competencia Q8.

ID	Q8
PREGUNTA	¿A cuál subcaracterística de calidad se encuentra asociada la métrica 'Z'?
RESPUESTA	El nombre de las instancias de los conceptos que representan subcaracterísticas.

CONSULTA SPARQL

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://wwwfinalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT DISTINCT ?s
WHERE {
 ?metricindividual rdf:type m:Metric .
 ?metricindividual m:metricName ?metricname .
 ?attribute sw:isDescribedBy ?individual .
 ?attribute sw:isMeasuredBy ?metricindividual .
 ?individual rdf:type q:Subcharacteristic .
 ?individual rdf:type ?s .
 FILTER(?metricname="Z"&&(regex(str(?s),"Accessibility")||
 regex(str(?s),"Accountability") ||
 regex(str(?s),"Adaptability") ||
 regex(str(?s),"Analysability") ||
 regex(str(?s),"Appropriateness") ||
 regex(str(?s),"Authenticity") ||
 regex(str(?s),"Availability")||
 regex(str(?s),"Capacity")||
 regex(str(?s),"Coexistence")||
 regex(str(?s),"Confidentiality") ||
 regex(str(?s),"FaultTolerance") ||
 regex(str(?s),"FunctionalCompleteness") ||
 regex(str(?s),"FunctionalCorrectness") ||
 regex(str(?s),"Installability") ||
 regex(str(?s),"Integrity")||
```

	regex(str(**?s**),"*Interoperability*") || regex(str(**?s**),"*Learnability*") || regex(str(**?s**),"*Maturity*") || regex(str(**?s**),"*Modifiability*") || regex(str(**?s**),"*Modularity*") || regex(str(**?s**),"*NonRepudiation*") || regex(str(**?s**),"*Operability*") || regex(str(?**s**),"*Recoverability*")|| regex(str(**?s**),"*Replaceability*") || regex(str(**?s**),"*ResourceUtilisation*") || regex(str(**?s**),"*Reusability*")|| regex(str(**?s**),"*Testability*") || regex(str(**?s**),"*TimeBehaviour*") || regex(str(**?s**),"*UserErrorProtection*")|| regex(str(?s),"*UserInterfaceAesthetics*"))) }
VAR. DE S.	"?s".

Tabla A.9. Documentación de la pregunta de competencia Q9.

ID	Q9
PREGUNTA	¿Cuántas veces se asocia la métrica 'Z' con la subcaracterística de calidad 'Y'?
RESPUESTA	Un valor entero que indica la cantidad de veces que se han asociado los elementos.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX owl: <http://www.w3.org/2002/07/owl#>
	PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
	PREFIX s: <http://wwwfinalontology.org#>
	PREFIX q: <http://www.qualitymodel.org#>
	PREFIX m: <http://www.metric.org#>
	PREFIX sw: <http://www.softwaremodel.org#>
	SELECT (COUNT(*) AS ?quantity) {
	SELECT ?individual
	WHERE
	{
	?metricindividual rdf:type m:Metric .
	?metricindividual m:metricName ?metricname .
	?attribute sw:isDescribedBy ?individual .
	?attribute sw:isMeasuredBy ?metricindividual .
	?individual rdf:type q:Subcharacteristic .
	?individual rdf:type ?subcharacteristic .
	FILTER (?metricname = "Z" &&
	regex(str(?subcharacteristic),"Y"))
	}
	}

VARIABLES DE SALIDA	-

Tabla A.10. Documentación de la pregunta de competencia Q10.

ID	Q10
PREGUNTA	¿A qué subcaracterística de calidad es usualmente asociada la métrica 'Z'?
RESPUESTA	El nombre de la instancia del concepto que representa la subcaracterística buscada.

CONSULTA SPARQL

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://wwwfinalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT ?s
{
   SELECT ?s (COUNT(?individual) AS ?quantity)
   {
   SELECT ?s ?individual
   WHERE
   {
    ?metricindividual rdf:type m:Metric .
    ?metricindividual m:metricName ?metricname .
    ?attribute sw:isDescribedBy ?individual .
    ?attribute sw:isMeasuredBy ?metricindividual .
    ?individual rdf:type q:Subcharacteristic .
    ?individual rdf:type ?s .
    FILTER(?metricname="Z" &&
    (regex(str(?s),"Accessibility")||
    regex(str(?s),"Accountability") ||
    regex(str(?s),"Adaptability") ||
    regex(str(?s),"Analysability") ||
    regex(str(?s),"Appropriateness") ||
    regex(str(?s),"Authenticity") ||
    regex(str(?s),"Availability")||
    regex(str(?s),"Capacity")||
```

```
    regex(str(?s),"Coexistence")||
    regex(str(?s),"Confidentiality") ||
    regex(str(?s),"FaultTolerance") ||
    regex(str(?s),"FunctionalCompleteness") ||
    regex(str(?s),"FunctionalCorrectness") ||
    regex(str(?s),"Installability") ||
    regex(str(?s),"Integrity")||
    regex(str(?s),"Interoperability") ||
    regex(str(?s),"Learnability") ||
    regex(str(?s),"Maturity") ||
    regex(str(?s),"Modifiability") || regex(str(?s),"Modularity")
    ||
    regex(str(?s),"NonRepudiation") ||
    regex(str(?s),"Operability") ||
    regex(str(?s),"Recoverability")||
    regex(str(?s),"Replaceability") ||
    regex(str(?s),"ResourceUtilisation") ||
    regex(str(?s),"Reusability")||
```

```
    regex(str(?s),"Testability") ||
    regex(str(?s),"TimeBehaviour") ||
    regex(str(?s),"UserErrorProtection")||
    regex(str(?s),"UserInterfaceAesthetics")))
    }
  }
```

	GROUP BY **?s** ORDER BY DESC(?quantity) LIMIT 1 }
VARIABLE DE SALIDA	"?s".

Tabla A.11. Documentación de la pregunta de competencia Q11.

ID	Q11
PREGUNTA	¿A qué esquema de calidad se asocia el artefacto 'V' del producto de software 'W'?
RESPUESTA	Los nombres de las tripletas de elementos que definen la estructura del esquema de calidad asociado a los elementos indicados.
CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX s: <http://wwwfinalontology.org#> PREFIX q: <http://www.qualitymodel.org#> PREFIX m: <http://www.metric.org#> PREFIX sw: <http://www.softwaremodel.org#> SELECT **?characteristicname ?subcharacname ?metric** WHERE { ?spindividual rdf:type sw:SoftwareProduct . ?spindividual sw:softwareProductName ?softwareproduct . ?spindividual sw:include ?softwaresystem . ?softwaresystem rdf:type sw:SoftwareSystem . ?softwaresystem sw:isDefinedBy ?artifactindividual . ?artifactindividual rdf:type sw:Artifact . ?artifactindividual sw:artifactName ?artifact . ?artifactindividual sw:isDividedIn ?entity . ?entity rdf:type sw:Entity . ?entity sw:hasAsProperty ?attribute . ?attribute rdf:type sw:Attribute . ?attribute sw:isDescribedBy ?s .

```
?s rdf:type q:Subcharacteristic .
?s rdf:type ?subcharacname.
?s q:belongs ?c .
?c rdf:type q:Characteristic .
?c rdf:type ?characteristicname .
?attribute sw:isMeasuredBy ?m .
?m m:metricName ?metric
FILTER (?softwareproduct = "W" && ?artifact = "V" &&
(regex(str(?characteristicname),"Compatibility") ||
regex(str(?characteristicname),"FunctionalSuitability")||
regex(str(?characteristicname),"Maintainability")||
regex(str(?characteristicname),"PerformanceEfficiency")||
regex(str(?characteristicname),"Portability") ||
regex(str(?characteristicname),"Reliability") ||
regex(str(?characteristicname),"Security") ||
regex(str(?characteristicname),"Usability")) &&
(regex(str(?subcharacname),"Accessibility") ||
regex(str(?subcharacname),"Accountability") ||
regex(str(?subcharacname),"Adaptability") ||
regex(str(?subcharacname),"Analysability") ||
regex(str(?subcharacname),"Appropriateness") ||
regex(str(?subcharacname),"Authenticity") ||
regex(str(?subcharacname),"Availability") ||
regex(str(?subcharacname),"Capacity")||
```

```
regex(str(?subcharacname),"Coexistence") ||
regex(str(?subcharacname),"Confidentiality") ||
regex(str(?subcharacname),"FaultTolerance") ||
regex(str(?subcharacname),"FunctionalAppropriateness")||
regex(str(?subcharacname),"FunctionalCompleteness") ||
```

	regex(str(**?subcharacname**),"*FunctionalCorrectness*") || regex(str(**?subcharacname**),"*Installability*") || regex(str(**?subcharacname**),"*Integrity*")|| regex(str(**?subcharacname**),"*Interoperability*") || regex(str(**?subcharacname**),"*Learnability*") || regex(str(**?subcharacname**),"*Maturity*") || regex(str(**?subcharacname**),"*Modifiability*") || regex(str(**?subcharacname**),"*Modularity*") || regex(str(**?subcharacname**),"*NonRepudiation*") || regex(str(**?subcharacname**),"*Operability*") || regex(str(**?subcharacname**),"*Recoverability*")|| regex(str(**?subcharacname**),"*Replaceability*") || regex(str(**?subcharacname**),"*ResourceUtilisation*") || regex(str(**?subcharacname**),"*Reusability*") || regex(str(**?subcharacname**),"*Testability*") || regex(str(**?subcharacname**),"*TimeBehaviour*") || regex(str(**?subcharacname**),"*UserErrorProtection*")|| regex(str(**?subcharacname**),"*UserInterfaceAesthetics*"))) }
VARIABLE DE SALIDA	"?characteristicname", "?subcharacname" y "?metric".

Tabla A.12. Documentación de la pregunta de competencia Q12.

ID	Q12
PREGUNTA	¿Cuál es el esquema de calidad asociado al producto de software 'W'?
RESPUESTA	Los nombres de las tripletas de elementos que definen la estructura del esquema de calidad asociado a los elementos indicados para cada artefacto encontrado.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX s: <http://wwwfinalontology.org#> PREFIX q: <http://www.qualitymodel.org#> PREFIX m: <http://www.metric.org#> PREFIX sw: <http://www.softwaremodel.org#> SELECT **?artifact ?characteristname ?subcharacname ?metric** WHERE { ?spindividual rdf:type sw:SoftwareProduct . ?spindividual sw:softwareProductName ?softwareproduct . ?spindividual sw:include ?softwaresystem . ?softwaresystem rdf:type sw:SoftwareSystem . ?softwaresystem sw:isDefinedBy ?artifactindividual . ?artifactindividual rdf:type sw:Artifact . ?artifactindividual sw:artifactName ?artifact . ?artifactindividual sw:isDividedIn ?entity . ?entity rdf:type sw:Entity . ?entity sw:hasAsProperty ?attribute . ?attribute rdf:type sw:Attribute . ?attribute sw:isDescribedBy ?subcharacteristic . ?subcharacteristic rdf:type q:Subcharacteristic . ?subcharacteristic rdf:type **?subcharacname** . ?subcharacteristic q:belongs ?characteristic . ?characteristic rdf:type q:Characteristic .

	?characteristic rdf:type **?characteristname** . ?attribute sw:isMeasuredBy ?metricindividual . ?metricindividual m:metricName **?metric** FILTER(?softwareproduct = "W" && (regex(str(**?characteristname**),"*Compatibility*") || regex(str(**?characteristname**),"*FunctionalSuitability*") || regex(str(**?characteristname**),"*Maintainability*") || regex(str(**?characteristname**),"*PerformanceEfficiency*") || regex(str(**?characteristname**),"*Portability*") || regex(str(**?characteristname**),"*Reliability*") || regex(str(**?characteristname**),"*Security*") || regex(str(**?characteristname**),"*Usability*")) && (regex(str(**?subcharacname**),"*Accessibility*") || regex(str(**?subcharacname**),"*Accountability*") || regex(str(**?subcharacname**),"*Adaptability*") || regex(str(**?subcharacname**),"*Analysability*") || regex(str(**?subcharacname**),"*Appropriateness*") || regex(str(**?subcharacname**),"*Authenticity*") || regex(str(**?subcharacname**),"*Availability*") || regex(str(**?subcharacname**),"*Capacity*") || regex(str(**?subcharacname**),"*Coexistence*") ||
	regex(str(**?subcharacname**),"*Confidentiality*") || regex(str(**?subcharacname**),"*FaultTolerance*") || regex(str(**?subcharacname**),"*FunctionalAppropriateness*") || regex(str(**?subcharacname**),"*FunctionalCompleteness*") || regex(str(**?subcharacname**),"*FunctionalCorrectness*") || regex(str(**?subcharacname**),"*Installability*") ||

	regex(str(**?subcharacname**),"*Integrity*") || regex(str(**?subcharacname**),"*Interoperability*") || regex(str(**?subcharacname**),"*Learnability*") || regex(str(**?subcharacname**),"*Maturity*") || regex(str(**?subcharacname**),"*Modifiability*") || regex(str(**?subcharacname**),"*Modularity*") || regex(str(**?subcharacname**),"*NonRepudiation*") || regex(str(**?subcharacname**),"*Operability*") || regex(str(**?subcharacname**),"*Recoverability*")|| regex(str(**?subcharacname**),"*Replaceability*") || regex(str(**?subcharacname**),"*ResourceUtilisation*") || regex(str(**?subcharacname**),"*Reusability*") || regex(str(**?subcharacname**),"*Testability*") || regex(str(**?subcharacname**),"*TimeBehaviour*") || regex(str(**?subcharacname**),"*UserErrorProtection*")|| regex(str(**?subcharacname**),"*UserInterfaceAesthetics*"))) }
VARIABLE DE SALIDA	"?artifact", "?characteristname", "?subcharacname" y "?metric".

Tabla A.13. Documentación de la pregunta de competencia Q13.

ID	Q13
PREGUNTA	¿A qué artefacto se asocia la métrica 'Z'?
RESPUESTA	El nombre de las instancias del concepto "Artefacto" que se asocian al elemento.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX owl: <http://www.w3.org/2002/07/owl#>
	PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
	PREFIX s: <http://wwwfinalontology.org#>
	PREFIX q: <http://www.qualitymodel.org#>
	PREFIX m: <http://www.metric.org#>
	PREFIX sw: <http://www.softwaremodel.org#>
	SELECT DISTINCT **?artifactname**
	WHERE {
	?metricindividual rdf:type m:Metric .
	?metricindividual m:metricName ?metric .
	?attribute rdf:type sw:Attribute .
	?attribute sw:isMeasuredBy ?metricindividual .
	?entity rdf:type sw:Entity .
	?entity sw:hasAsProperty ?attribute .
	?artifact rdf:type sw:Artifact .
	?artifact sw:isDividedIn ?entity .
	?artifact sw:artifactName **?artifactname**
	FILTER (?metric = "Z")
	}

VARIABLES DE SALIDA	"?artifactname".

Tabla A.14. Documentación de la pregunta de competencia Q14.

ID	Q14
PREGUNTA	¿A qué programa de computadora se asocia la métrica 'Z'?

RESPUESTA	El nombre de las instancias del concepto “Programa de Computadora” que se asocian al elemento.

CONSULTA SPARQL	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX owl: <http://www.w3.org/2002/07/owl#>
	PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
	PREFIX s: <http://wwwfinalontology.org#>
	PREFIX q: <http://www.qualitymodel.org#>
	PREFIX m: <http://www.metric.org#>
	PREFIX sw: <http://www.softwaremodel.org#>
	SELECT DISTINCT **?softwaresystemname**
	WHERE
	{
	?metricindividual rdf:type m:Metric .
	?metricindividual m:metricName ?metric .
	?attribute rdf:type sw:Attribute .
	?attribute sw:isMeasuredBy ?metricindividual .
	?entity rdf:type sw:Entity .
	?entity sw:hasAsProperty ?attribute .
	?artifact rdf:type sw:Artifact .
	?artifact sw:isDividedIn ?entity .
	?softwaresystem rdf:type sw:SoftwareSystem .
	?softwaresystem sw:isDefinedBy ?artifact .
	?softwaresystem sw:softwareSystemName **?softwaresystemname**
	FILTER (?metric = "Z")
	}

VARIABLES DE SALIDA	“?softwaresystemname”.

Tabla A.15. Documentación de la pregunta de competencia Q15.

ID	Q15

PREGUNTA	¿Cuántos artefactos se asocian a la métrica 'Z'?
RESPUESTA	Un valor entero que indica la cantidad de veces que se han asociado los elementos.

CONSULTA SPARQL	`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
	`PREFIX owl: <http://www.w3.org/2002/07/owl#>`
	`PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>`
	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>`
	`PREFIX s: <http://wwwfinalontology.org#>`
	`PREFIX q: <http://www.qualitymodel.org#>`
	`PREFIX m: <http://www.metric.org#>`
	`PREFIX sw: <http://www.softwaremodel.org#>`
	`SELECT (COUNT(?artifact) AS ?quantity)`
	`{`
	`SELECT DISTINCT ?artifact`
	`WHERE {`
	`?metricindividual rdf:type m:Metric .`
	`?metricindividual m:metricName ?metric .`
	`?attribute rdf:type sw:Attribute .`
	`?attribute sw:isMeasuredBy ?metricindividual .`
	`?entity rdf:type sw:Entity .`
	`?entity sw:hasAsProperty ?attribute .`
	`?artifact rdf:type sw:Artifact .`
	`?artifact sw:isDividedIn ?entity .`
	`?artifact sw:artifactName ?artifactname`
	`FILTER (?metric = "Z")`
	`}`
	`}`

VARIABLES DE SALIDA	-

Tabla A.16. Documentación de la pregunta de competencia Q16.

ID	Q16
PREGUNTA	¿Cuántos programas de computadora se asocian a la métrica 'Z'?
RESPUESTA	Un valor entero que indica la cantidad de veces que se han asociado los elementos.
CONSULTA SPARQL	
VARIABLES DE SALIDA	-

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://wwwfinalontology.org#>
PREFIX q: <http://www.qualitymodel.org#>
PREFIX m: <http://www.metric.org#>
PREFIX sw: <http://www.softwaremodel.org#>
SELECT (COUNT(?softwaresystem) AS ?quantity) {
      SELECT DISTINCT ?softwaresystem
      WHERE {
        ?metricindividual rdf:type m:Metric .
        ?metricindividual m:metricName ?metric .
        ?attribute rdf:type sw:Attribute .
        ?attribute sw:isMeasuredBy ?metricindividual .
        ?entity rdf:type sw:Entity .
        ?entity sw:hasAsProperty ?attribute .
        ?artifact rdf:type sw:Artifact .
        ?artifact sw:isDividedIn ?entity .
        ?softwaresystem rdf:type sw:SoftwareSystem .
        ?softwaresystem sw:isDefinedBy ?artifact .
        FILTER (?metric = "Z")
      }
}
```

Apéndice B. Restricciones OCL del Metamodelo

En este apéndice se detallan las invariantes OCL diseñadas con el objetivo de restringir la conformación de los posibles diseños arquitectónicos instanciables a partir de la descripción UML presentada en el *Capítulo 8*.

Tabla B.1. Invariantes que restringen la descripción UML.

Nº	RESTRICCIÓN / DESCRIPCIÓN
1	**context** CloudApplication **inv:** CloudApplication.allInstances() → select(c:CloudApplication | c.name=self.name) → size()=1 Un *CloudApplication* debe tener un nombre único.
2	**context** CloudApplication **inv:** self.name→size()>0 Un *CloudApplication* debe tener un nombre.
3	**context** CloudApplicationDecomposedInLayers **inv:** self.decompositionApproach.oclIsTypeOf(LayerDecomposition) La descomposición utilizada por *CloudApplicationDecomposedInLayers* debe ser *LayerDecomposition*.
4	**context** CloudApplicationDecomposedInProcesses **inv:** self.decompositionApproach.oclIsTypeOf(ProcessDecomposition) La descomposición utilizada por *CloudApplicationDecomposedInProcesses* debe ser *ProcessDecomposition*.
5	**context** CloudApplicationDecomposedInPipesAndFilters **inv:**

```
self.decompositionApproach.oclIsTypeOf(PipesAndFiltersDecomposition)
```

La descomposición utilizada por *CloudApplicationDecomposedInPipesAnd* Filters debe ser *PipesAndFiltersDecomposition*.

```
context Tiers
inv: self.numberOfTiers>0
```

6

La cantidad de bandas de un enfoque basado en bandas (atributo *numberOfTiers* de la entidad *Tiers*) debe ser positiva.

```
context ExternalLink
inv: self.start.architecturalComponent <>
     self.end.architecturalComponent
```

7

El inicio y el fin de un *ExternalLink* no pueden referir al mismo *ArchitecturalComponent*.

```
context ExternalLink
inv: self.start.architecturalComponent.actualApplication =
     self.end.architec-turalComponent.actualApplication
```

8

Un *ExternalLink* debe vincular componentes arquitectónicos que estén incluidos en la misma aplicación.

```
context ArchitecturalComponent
inv:
   if(self.oclIsTypeOf(ElasticManager))
   then true
   else (self.input.externalLink →size() + self.output.externalLink→
         size()) > 0
   endif
```

9

Un *ArchitecturalComponent* no puede estar aislado, salvo que sea un *ElasticManager*.

```
context LoadBalancer
inv: self.numberOfSynchronousRequests >= 0
```

10

La cantidad de solicitudes (atributo *numberOfSynchronousRequests*) de un *LoadBalancer* debe ser mayor o igual a 0.

```
context MessageQueue
inv: self.numberOfQueuedMessages >= 0
```

11

La cantidad de mensajes en cola (atributo *numberOfQueuedMessages*) de una *MessageQueue* debe ser mayor o igual a 0.

```
context UndefinedApplicationComponent
inv: self.output.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(SpecialUnidirectionalExternalLink)) →
     forAll(e2:ExternalLink |
12
     e2.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::heartbeat)
```

Un *UndefinedApplicationComponent* que tiene en sus salidas un elemento del tipo *SpecialUnidirectionalExternalLink* debe ser del tipo *heartbeat*.

```
context Application
inv:
   if (self.distributionApproach→size()=2)
   then
       self.distributionApproach → select(d1:DistributionApproach|
       d1.oclIsTypeOf(Tiers)) → size() = 1 and
13
       self.distributionApproach → select(d2 : DistributionApproach |
       d2.oclIsTypeOf( ContentDistributionNetwork)) → size()=1
   else true
   endif
```

Si una aplicación tiene asociados dos enfoques de distribución, uno debe ser de *Tiers* y el otro *ContentDistributionNetwork*.

```
context UndefinedApplicationComponent
inv: self.input.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(Special UnidirectionalExternalLink)) →
     forAll(e2:ExternalLink |
14
     e2.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::scale)
```

Un *UndefinedApplicationComponent* que tiene en sus entradas un elemento del tipo *SpecialUnidirectionalExternalLink* requiere que su tipo sea *scale*.

```
context DefinedApplicationComponent
inv: self.input.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(Special UnidirectionalExternalLink)) →
     forAll(e2:ExternalLink |
15
     e2.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::testRequest)
```

Un *DefinedApplicationComponent* que tiene en sus entradas un elemento del tipo *SpecialUnidirectionalExternalLink* requiere que su tipo sea *testRequest*.

```
context UndefinedApplicationComponent
inv: self.output.externalLink → forAll(e |
16
     e.end.component.odlIsKindOf(Defined ApplicationComponent))
```

Los *ExternalLink* que salen de un *UndefinedApplicationComponent* sólo pueden ir a un *DefinedApplicationComponent*.

```
context DefinedApplicationComponent
inv: self.output.externalLink → forAll(e:ExternalLink |
     e.end.architectural Component.oclIsKindOf(
17   UndefinedApplicationComponent) or
     e.oclIsTypeOf(SpecialUnidirectionalExternalLink))
```

Un *DefinedApplicationComponent* debe tener en sus salidas *ExternalLink* cuyo destino sea *UndefinedApplicationComponent* o, en otro caso, un vínculo que corresponda al tipo *SpecialUnidirectionalExternalLink*.

```
context ElasticLoadBalancer
18 inv: self.input.externalLink→size()=1
```

Un *ElasticLoadBalancer* debe tener en su entrada un único *ExternalLink*.

```
context ElasticLoadBalancer
inv: self.input.externalLink → forAll(e |
     e.oclIsTypeOf(SpecialUnidirectional ExternalLink) and
19   e.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::numberOfSynchronousRequests)
```

La entrada de un *ElasticLoadBalancer* debe ser un *SpecialUnidirectionalExternalLink* del tipo *numberOfSynchronousRequests*.

```
context ElasticLoadBalancer
inv: self.output.externalLink→size()=1
```

20

Un *ElasticLoadBalancer* debe tener en su salida un único *ExternalLink*.

```
context ElasticLoadBalancer
inv: self.output.externalLink → forAll(e |
     e.oclIsTypeOf(SpecialUnidirectional ExternalLink) and
     e.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::scale)
```

21

La salida de un *ElasticLoadBalancer* debe ser *SpecialUnidirectionalExternalLink* tipo *scale*.

```
context LoadBalancer
inv: self.output.externalLink → select(e:ExternalLink |
     e.oclIsTypeOf(Special UnidirectionalExternalLink) and
     e.oclAsType(SpecialUnidirectionalExternal Link).type =
     TypeOfSpecialLink::numberOfSynchronousRequests) →size() > 0
```

22

Un *LoadBalancer* debe tener al menos una salida que sea *SpecialUnidirectionalExternalLink* de tipo *numberOfSynchronousRequests*.

```
context LoadBalancer
inv: self.output.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(SpecialUnidirectionalExternalLink)) →
     forAll(e2:ExternalLink|
     e2.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::numberOfSynchronous Requests)
```

23

Un *LoadBalancer* si tiene una salida *SpecialUnidirectionalExternalLink* esta debe ser de tipo *numberOfSynchronousRequests*.

```
context LoadBalancer
inv: self.output.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(UnidirectionalExternalLink) or
     e1.oclIsTypeOf(BidirectionalExternalLink)) →
     select(e2:ExternalLink |
     e2.end.architecturalComponent.oclIsKindOf(
```

24

```
     UndefinedApplicationComponent)) → size() > 0
```

Un *LoadBalancer* debe tener al menos una salida a *UndefinedApplicationComponent*.

```
context ElasticQueue
25 inv: self.input.externalLink→size() = 1
```

Una *ElasticQueue* debe tener en su entrada un único *ExternalLink*.

```
context ElasticQueue
inv: self.input.externalLink → forAll(e:ExternalLink |
     e.oclIsTypeOf(Special UnidirectionalExternalLink) and
26   e.oclAsType(SpecialUnidirectionalExternal Link).type =
     TypeOfSpecialLink::numberOfQueuedMessages)
```

La entrada de *ElasticQueue* debe ser *SpecialUnidirectionalExternalLink* tipo *numberOfQueuedMessages*.

```
context ElasticQueue
27 inv: self.output.externalLink→size()=1
```

Una *ElasticQueue* debe tener en su salida un único *ExternalLink*.

```
context ElasticQueue
inv: self.output.externalLink → forAll(e |
     e.oclIsTypeOf(SpecialUnidirectional ExternalLink) and
28   e.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::scale)
```

La salida de *ElasticQueue* debe ser *SpecialUnidirectionalExternalLink* cuyo tipo sea *scale*.

```
context MessageQueue
inv: self.output.externalLink → select(e:ExternalLink |
29   e.oclIsTypeOf(Special UnidirectionalExternalLink) and
     e.oclAsType(SpecialUnidirectionalExternal Link).type =
     TypeOfSpecialLink::numberOfQueuedMessages) → size() > 0
```

Una *MessageQueue* debe tener al menos una salida *SpecialUnidirectionalExternalLink* de tipo *numberOfQueuedMessages*.

```
context ElasticManager
inv: self.input.externalLink→isEmpty() and
30
     self.output.externalLink→isEmpty()
```

Un *ElasticManager* no debe vincularse con ningún *ExternalLink*.

```
context MessageQueue
inv: self.output.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(Special UnidirectionalExternalLink)) →
     forAll(e2:ExternalLink|
31
     e2.oclAsType(SpecialUnidirectionalExternalLink).type =
     TypeOfSpecialLink::numberOfQueuedMessages)
```

Un *MessageQueue* si tiene una salida *SpecialUnidirectionalExternalLink* esta debe ser de tipo *numberOfQueuedMessages*.

```
context MessageQueue
inv: self.output.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf( UnidirectionalExternalLink) or
     e1.oclIsTypeOf(BidirectionalExternalLink) →
32   select(e2:ExternalLink |
     e2.end.architecturalComponent.oclIsKindOf(
     UndefinedApplicationComponent)) → size() > 0
```

Una *MessageQueue* debe tener al menos una salida que esté vinculada a un *UndefinedApplicationComponent*.

```
context ProviderAdapter
33 inv: self.input.externalLink → size()>0
```

Un *ProviderAdapter* debe tener al menos un *ExternalLink* en su entrada.

```
context ProviderAdapter
inv: self.input.externalLink →
34   forAll(e|e.oclIsTypeOf(BidirectionalExternalLink) and
     e.start.architecturalComponent.oclIsKindOf(
     UndefinedApplication Component))
```

Un *ProviderAdapter* debe tener todas sus entradas del tipo *BidirectionalExternalLink* cuyo origen sea *UndefinedApplicationComponent*.

```
context ProviderAdapter
inv: self.output.externalLink->isEmpty()
```

Un *ProviderAdapter* no puede tener *ExternalLink* en sus salidas.

```
context Watchdog
inv: self.input.externalLink → forAll(e:ExternalLink |
     e.oclIsTypeOf(Special UnidirectionalExternalLink) or
     (e.oclIsTypeOf(BidirectionalExternalLink)))
```

Un *Watchdog* no puede tener en su entrada un vínculo *UnidirectionalExternalLink*.

```
context Watchdog
inv: self.input.externalLink →
     select(e1:ExternalLink |
     e1.oclIsTypeOf( BidirectionalExternalLink)) →
     forAll(e2:ExternalLink |
     e2.start.architecturalComponent.oclIsTypeOf(
     ConfigurationManager))
```

Un *Watchdog* si tiene un vínculo *BidirectionalLink* en su entrada debe ser uno que provenga de un *ConfigurationManager*.

```
context ConfigurationManager
inv: self.input.externalLink→isEmpty()
```

Un *ConfigurationManager* no tiene entradas.

```
context Watchdog
inv: self.input.externalLink → select(e:ExternalLink |
     e.oclIsTypeOf(Bidirectional ExternalLink) and
     e.start.architecturalComponent.oclIsTypeOf(
     Configuration Manager)) → size() = 1
```

Un *Watchdog* debe tener un único *BidirectionalLink* desde *ConfigurationManager*.

```
context Watchdog
inv: (self.input.externalLink →
     select(e1:ExternalLink |
40
     e1.oclIsTypeOf(SpecialUnidirectionalExternalLink))→size()+1)=
     self.input.externalLink→size()
```

Un *Watchdog* debe tener todas sus entradas *SpecialUnidirectionalExternalLink* (salvo una).

```
context Watchdog
inv: self.input.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(Special UnidirectionalExternalLink)) →
     forAll(e2:ExternalLink |
41
     e2.oclAsType(Special UnidirectionalExternalLink).type =
     TypeOfSpecialLink::heartbeat)
```

Todos los vínculos *SpecialUnidirectionalExternalLink* conectados a la entrada de un *Watchdog* deben ser del tipo *heartbeat*.

```
context Watchdog
inv: self.input.externalLink → select(e1:ExternalLink |
     e1.oclIsTypeOf(Special UnidirectionalExternalLink)) →
     forAll(e2:ExternalLink |
42
     e2.start.architecturalComponent.oclIsTypeOf(
     StatelessComponent))
```

Todos los vínculos *SpecialUnidirectionalExternalLink* conectados a la entrada de un *Watchdog* deben provenir de un *StatelessComponent*.

```
context Watchdog
inv: self.output.externalLink → forAll(e:ExternalLink |
     e.oclIsTypeOf(Special UnidirectionalExternalLink) and
     e.oclAsType(SpecialUnidirectionalExternal Link).type =
43
     TypeOfSpecialLink::testRequest and
     (e.end.architecturalComponent.oclIsTypeOf(LoadBalancer) or
     e.end.architecturalComponent.oclIsTypeOf(MessageQueue)))
```

Un *Watchdog* si tiene en sus salidas algún *ExternalLink* debe ser *SpecialUnidirectionalLink* del tipo *testRequest* y que se dirija hacia un *LoadBalancer* o hacia un *MessageQueue*.

```
context ConfigurationManager
inv: self.output.externalLink→
     forAll(e:ExternalLink |
     e.oclIsTypeOf(SpecialUnidirectionalExternalLink) and
44
     e.oclAsType(SpecialUnidirectionalExternal Link).type =
     TypeOfSpecialLink::configuration)
```

Las salidas de un *ConfigurationManager* deben ser todas *SpecialUnidirectionalExternalLink* del tipo *configuration*.

```
context UndefinedApplicationComponent
inv: self.part → select(p:FunctionalComponent |
45
     p.input.internalLink→size()=0) → size() = 1
```

Un *UndefinedApplicationComponent* debe contener un único *FunctionalComponent* cuyo *ControlInput* no tenga *InternalLink* asociados.

```
context UndefinedApplicationComponent
inv: self.part → select(p:FunctionalComponent |
46
     p.output.internalLink→size()=0) →size()=1
```

Un *UndefinedApplicationComponent* debe contener un único *FunctionalComponent* cuyo *ControlOutput* no tenga *InternalLink* asociados.

```
context FunctionalComponent
inv: if (self.application.part → size() = 1)
     then true
     else (self.input.internalLink→size() + self.output.internalLink
47
     → size()) > 0
     endif
```

Un *FunctionalComponent* no puede estar aislado, salvo que sea el único incluido en una aplicación.

```
context InternalLink
inv: self.start.component.application =
48
     self.end.component.application
```

Un *InternalLink* debe vincular dos componentes funcionales que se encuentren en el mismo componente de aplicación.

```
context InternalLink
49 inv: self.start.component <> self.end.component
```

El inicio y el fin de un *InternalLink* no pueden referir al mismo *FunctionalComponent*.

```
context StatelessComponent
inv: (self.handler → select(h | h.state.oclIsTypeOf(SessionState)) →
     size() = 1) and (self.handler → select(h |
50
     h.state.oclIsTypeOf(ApplicationState)) → size() = 1)
```

Los *StateHandler* de un *StatelessComponent* deben asignarse a un *SessionState* y a un *ApplicationState* respectivamente.

```
context StatefulComponent
inv: self.synchronizer → select(h |
     h.state.oclIsTypeOf(SessionState) → size() = 1 and
51
     self.synchronizer → select(h |
     h.state.oclIsTypeOf(AplicationState) → size() = 1
```

Los *StateSynchronizer* de un *StatefulComponent* deben asignarse a un *SessionState* y a un *ApplicationState* respectivamente.

```
context UserInterfaceComponent
52 inv: self.application.oclIsTypeOf(StatelessComponent)
```

Un *UserInterfaceComponent* debe estar incluido en un *UndefinedApplicationComponent* que sea *StatelessComponent*.

```
context UndefinedApplicationComponent
inv: UndefinedApplicationComponent.allInstances() →
53   select(c:Undefined ApplicationComponent|c.name=self.name) →
     size() = 1
```

Cada *UndefinedApplicationComponent* debe tener un nombre único.

```
context UndefinedApplicationComponent
54 inv: self.name→size()>0
```

Un *UndefinedApplicationComponent* debe tener un nombre.

```
context Application
55 inv:(self.managementComponent→size() + self.applicationComponent →
     size())>0
```

Una instancia de *Application* debe estar compuesta por al menos un componente del tipo *ArchitecturalComponent*.

```
context SpecialUnidirectionalExternalLink
56 inv: self.type <> TypeOfSpecialLink::unset
```

Un *SpecialUnidirectionalExternalLink* no puede tener como tipo *unset*.

```
context StateManager
57 inv: self.source <> TypeOfSource::unset
```

Un *StateManager* no puede tener como *source* el valor *unset*.

```
context Information
58 inv: self.consistent <> TypeOfConsistent::unset
```

Un *Information* no puede tener como *consistent* el valor *unset*.

```
context UndefinedApplicationComponent
59 inv: self.sourceOfApplicationInformation <> TypeOfSource::unset and
     self.sourceOfSessionInformation<>TypeOfSource::unset
```

Un *UndefinedApplicationComponent* no puede tener los *source* en *unset*.

```
context UndefinedApplicationComponent
60 inv: self.typeOfConsistentOfApplicationInformation <>
     TypeOfConsistent::unset and
     self.typeOfConsistentOfSessionInformation <>
     TypeOfConsistent::unset
```

Un *UndefinedApplicationComponent* no puede tener los atributos *typeOfConsistent* en *unset*.

Apéndice C. Modelos RDEVS y DEVS en Java

En este apéndice se presentan las clases Java diseñadas para los modelos RDEVS especificados formalmente en el *Capítulo 10*.

C.1 Modelo Esencial “Example”

```
public class Example extends rd.models.EssentialModel {

   private String componentName;

   private Distribution processingTimeDistributionUIC;
   private double faultProbabilityUIC, failureProbabilityUIC;
   private Distribution processingTimeDistributionPC1;
   private double faultProbabilityPC1, failureProbabilityPC1;
   private Distribution processingTimeDistributionDAC;
   private double faultProbabilityDAC, failureProbabilityDAC;
   private Distribution processingTimeDistributionPC2;
   private double faultProbabilityPC2, failureProbabilityPC2;

   private NextFunctionState nextFunctionState;

   public Example(Distribution processingTimeDistributionUIC, double
   faultProbabilityUIC, double failureProbabilityUIC, Distribution
   processingTimeDistributionPC1, double faultProbabilityPC1, double
   failureProbabilityPC1, Distribution processingTimeDistributionDAC, double
   faultProbabilityDAC, double failureProbabilityDAC, Distribution
   processingTimeDistributionPC2, double faultProbabilityPC2, double
   failureProbabilityPC2, NextFunctionState nextFunctionState) {
      super("Example", new String[] {"input"}, new String[] {"output"});
      this.componentName = "Example";
      this.processingTimeDistributionUIC = processingTimeDistributionUIC;
      this.faultProbabilityUIC = faultProbabilityUIC;
      this.failureProbabilityUIC = failureProbabilityUIC;
      this.processingTimeDistributionPC1 = processingTimeDistributionPC1;
      this.faultProbabilityPC1 = faultProbabilityPC1;
      this.failureProbabilityPC1 = failureProbabilityPC1;
      this.processingTimeDistributionDAC = processingTimeDistributionDAC;
      this.faultProbabilityDAC = faultProbabilityDAC;
      this.failureProbabilityDAC = failureProbabilityDAC;
      this.processingTimeDistributionPC2 = processingTimeDistributionPC2;
```

```
      this.faultProbabilityPC2 = faultProbabilityPC2;
      this.failureProbabilityPC2 = failureProbabilityPC2;
      this.nextFunctionState = nextFunctionState;
   }

   public ExampleState getInitialState(){
      return new ExampleState(ExamplePhase.waiting, State.INFINITY, null,
      this.nextFunctionState.get(this.faultProbabilityUIC,
      this.failureProbabilityUIC));
   }

   protected State externalTransitionFunction(State s, double e, int inputNumber,
   EventContent messageContent) throws UndefinedPortException,EmptyPortException {

      ExampleState state = (ExampleState) s;

      if(messageContent.getClass().toString().contains("OutOfUse")) {
         return new
         ExampleState(ExamplePhase.sendingState,0,null,FunctionState.empty);
      }

      if(ExamplePhase.isWaiting(state.getPhase()))
      {

         if(FunctionState.isOk(state.getNextFunctionState())) {
            Request req = (Request) messageContent;
            req.removeFirstFromComponentsList();
            double processingTimeUIC = this.processingTimeDistributionUIC.get();
            req.refreshProcessingTimes(processingTimeUIC);
            return new ExampleState(ExamplePhase.processingUIC, processingTimeUIC,
            req, this.nextFunctionState.get( this.faultProbabilityPC1,
            this.failureProbabilityPC1));
         }

         if(FunctionState.isFault(state.getNextFunctionState())) {
            Request req = (Request) messageContent;
            req.removeFirstFromComponentsList();
            return new
            ExampleState(ExamplePhase.sendingFault,0,req,FunctionState.empty);
         }

         if(FunctionState.isFailure(state.getNextFunctionState())) {
            return new ExampleState(ExamplePhase.sendingFailure, 0, null,
            FunctionState.empty);
         }

      }

      state.setSigma(state.getSigma()-e);
      return state;

   }

   protected State getNewState() {
      return this.getInitialState();
```

```
}
protected State internalTransitionFunction(State s) {

   ExampleState state = (ExampleState) s;

   if(ExamplePhase.isSendingRequest(state.getPhase()))
      return new ExampleState(ExamplePhase.sendingInformation, 0,
      state.getActualRequest(), FunctionState.empty);

   if(ExamplePhase.isSendingState(state.getPhase()))
      return new ExampleState(ExamplePhase.inactive, State.INFINITY, null,
      FunctionState.empty);

   if(ExamplePhase.isSendingFault(state.getPhase()))
   {
      Request req = state.getActualRequest();
      double processingTimeUIC = this.processingTimeDistributionUIC.get();
      req.refreshProcessingTimes(processingTimeUIC);
      req.setIncorrectProcessing();
      return new ExampleState(ExamplePhase.processingFaultsUIC,
      processingTimeUIC, req, this.nextFunctionState.get(
      this.faultProbabilityPC1, this.failureProbabilityPC1));
   }

   if(ExamplePhase.isSendingFailure(state.getPhase()))
      return new ExampleState(ExamplePhase.failure, State.INFINITY, null,
      FunctionState.empty);

   if(ExamplePhase.isSendingInformation(state.getPhase()))
      return new ExampleState(ExamplePhase.waiting, State.INFINITY, null,
      this.nextFunctionState.get( this.faultProbabilityUIC,
      this.failureProbabilityUIC));

   if(FunctionState.isFailure(state.getNextFunctionState()))
      return new ExampleState(ExamplePhase.failure, State.INFINITY, null,
      FunctionState.empty);

   if(ExamplePhase.isProcessingPC2(state.getPhase()) ||
   ExamplePhase.isProcessingWithFaultsPC2(state.getPhase())) {
      return new ExampleState(ExamplePhase.sendingRequest, 0,
      state.getActualRequest(), FunctionState.empty);
   }

   if(FunctionState.isOk(state.getNextFunctionState())) {

    if(ExamplePhase.isProcessingUIC(state.getPhase()) ||
    ExamplePhase.isProcessingWithFaultsUIC(state.getPhase()))
    {
        Request req = state.getActualRequest();
        double processingTimePC1 = this.processingTimeDistributionPC1.get();
        req.refreshProcessingTimes(processingTimePC1);
        return new ExampleState(ExamplePhase.processingPC1, processingTimePC1,
        req, this.nextFunctionState.get( this.faultProbabilityDAC,
        this.failureProbabilityDAC));
     }
```

```
      if(ExamplePhase.isProcessingPC1(state.getPhase()) ||
      ExamplePhase.isProcessingWithFaultsPC1(state.getPhase()))
      {
         Request req = state.getActualRequest();
         double processingTimeDAC = this.processingTimeDistributionDAC.get();
         req.refreshProcessingTimes(processingTimeDAC);
         return new ExampleState(ExamplePhase.processingDAC, processingTimeDAC,
         req, this.nextFunctionState.get( this.faultProbabilityPC2,
         this.failureProbabilityPC2));
      }

      if(ExamplePhase.isProcessingDAC(state.getPhase()) ||
      ExamplePhase.isProcessingWithFaultsDAC(state.getPhase()))
      {
         Request req = state.getActualRequest();
         double processingTimePC2 = this.processingTimeDistributionPC2.get();
         req.refreshProcessingTimes(processingTimePC2);
         return new ExampleState(ExamplePhase.processingPC2, processingTimePC2,
         req, FunctionState.empty);
      }

   }

   if(FunctionState.isFault(state.getNextFunctionState())) {

      if(ExamplePhase.isProcessingUIC(state.getPhase()) ||
      ExamplePhase.isProcessingWithFaultsUIC(state.getPhase()))
      {
         Request req = state.getActualRequest();
         double processingTimePC1 = this.processingTimeDistributionPC1.get();
         req.refreshProcessingTimes(processingTimePC1);
         req.setIncorrectProcessing();
         return new ExampleState(ExamplePhase.processingFaultsPC1,
         processingTimePC1, req, this.nextFunctionState.get(
         this.faultProbabilityDAC, this.failureProbabilityDAC));
      }

      if(ExamplePhase.isProcessingPC1(state.getPhase()) ||
      ExamplePhase.isProcessingWithFaultsPC1(state.getPhase()))
      {
         Request req = state.getActualRequest();
         double processingTimeDAC = this.processingTimeDistributionDAC.get();
         req.refreshProcessingTimes(processingTimeDAC);
         req.setIncorrectProcessing();
         return new ExampleState(ExamplePhase.processingFaultsDAC,
         processingTimeDAC, req, this.nextFunctionState.get(
         this.faultProbabilityPC2, this.failureProbabilityPC2));
      }

      if(ExamplePhase.isProcessingDAC(state.getPhase()) ||
      ExamplePhase.isProcessingWithFaultsDAC(state.getPhase()))
      {
         Request req = state.getActualRequest();
         double processingTimePC2 = this.processingTimeDistributionPC2.get();
```

```
            req.refreshProcessingTimes(processingTimePC2);
            req.setIncorrectProcessing();
            return new ExampleState(ExamplePhase.processingFaultsPC2,
            processingTimePC2, req, FunctionState.empty);
         }
      }
      return null;
   }

   protected Output outputFunction(State s) throws UndefinedPortException,
   EmptyPortException {
      ExampleState state = (ExampleState) s;
      if(ExamplePhase.isSendingState(state.getPhase()))
         return new Output(this.getOutputPort(0), new ComponentState(
         this.componentName, InfrastructureState.expired));

      if(ExamplePhase.isSendingRequest(state.getPhase()))
         return new Output(this.getOutputPort(0),state.getActualRequest());

      if(ExamplePhase.isSendingInformation(state.getPhase()))
         return new Output(this.getOutputPort(0),state.getActualRequest());

      if(ExamplePhase.isSendingFault(state.getPhase()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName, InfrastructureState.running,
         ActivityState.active, ProcessingState.fault));

      if(ExamplePhase.isSendingFailure(state.getPhase()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName, InfrastructureState.running,
         ActivityState.failure));

      if(ExamplePhase.isProcessingPC2(state.getPhase()) ||
      ExamplePhase.isProcessingWithFaultsPC2(state.getPhase()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName,
         InfrastructureState.running,ActivityState.active,ProcessingState.ok));

      if(FunctionState.isOk(state.getNextFunctionState()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName,
         InfrastructureState.running,ActivityState.active,ProcessingState.ok));

      if(FunctionState.isFailure(state.getNextFunctionState()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName,
         InfrastructureState.running,ActivityState.failure));

      if(FunctionState.isFault(state.getNextFunctionState()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName, InfrastructureState.running,
         ActivityState.active, ProcessingState.fault));

      return null;
   }
```

```
}
```

C.2 Modelo Esencial "MessageQueue"

```
public class MessageQueue extends rd.models.EssentialModel {

   private String componentName;

   private String applicationComponent;
   private ArrayList replicas;

   public MessageQueue(String applicationComponent, ArrayList replicas) {
      super("MessageQueue", new String[] {"input"}, new String[] {"output"});
      this.componentName = "MessageQueue";
      this.applicationComponent = applicationComponent;
      this.replicas = replicas;
   }

   public MessageQueueState getInitialState(){
       return new MessageQueueState(MessageQueuePhase.waiting, State.INFINITY,
       this.replicas);
   }

   protected State externalTransitionFunction(State s, double e, int inputNumber,
   EventContent messageContent) throws UndefinedPortException,EmptyPortException {

      MessageQueueState state = (MessageQueueState) s;

      if(messageContent.getClass().toString().contains("OutOfUse")) {
         return new MessageQueueState(MessageQueuePhase.sendingState,0,new
         ArrayList());
      }

      if(MessageQueuePhase.isWaiting(state.getPhase())) {

         Request req = (Request) messageContent;

         if(state.queueIsEmpty() && state.isInRequestsInProcess(req.getID()))
         {
            String replica = state.removeRequestInProcess(req.getID());
            state.addFreeReplica(replica);
            return new MessageQueueState(MessageQueuePhase.waiting, State.INFINITY,
            state.getQueue(), state.getSimulationTime() + e,
            state.getFreeReplicas(), state.getRequestsInProcess());
         }

         if(!state.queueIsEmpty() && state.isInRequestsInProcess(req.getID()))
         {
            String replica = state.removeRequestInProcess(req.getID());
            state.addFreeReplica(replica);
```

```
            return new MessageQueueState(MessageQueuePhase.sendingRequest, 0,
            state.getQueue(), state.getSimulationTime() + e,
            state.getFreeReplicas(), state.getRequestsInProcess());
         }

         if(state.freeReplicasEmpty() && !req.componentsListEmpty() &&
         req.secondFromComponentsList(this.applicationComponent))
         {
            req.removeFirstFromComponentsList();
            state.addQueue(state.getSimulationTime()+e,req);
            return new MessageQueueState(MessageQueuePhase.waiting, State.INFINITY,
            state.getQueue(), state.getSimulationTime()+e,
            state.getFreeReplicas(),state.getRequestsInProcess());
         }

         if(!state.freeReplicasEmpty() && !req.componentsListEmpty() &&
         req.secondFromComponentsList(this.applicationComponent))
         {
            req.removeFirstFromComponentsList();
            state.addQueue(state.getSimulationTime()+e,req);
            return new MessageQueueState(MessageQueuePhase.sendingRequest, 0,
            state.getQueue(), state.getSimulationTime() + e,
            state.getFreeReplicas(),state.getRequestsInProcess());
         }

         if(!req.componentsListEmpty() &&
         !req.secondFromComponentsList(this.applicationComponent))
         {
            return new MessageQueueState(MessageQueuePhase.waiting,State.INFINITY,
            state.getQueue(),state.getSimulationTime()+e,
            state.getFreeReplicas(),state.getRequestsInProcess());
         }
      }

      state.setSigma(state.getSigma()-e);
      state.setSimulationTime(state.getSimulationTime()+e);
      return state;

   }

   protected State getNewState() {
      return this.getInitialState();
   }

   protected State internalTransitionFunction(State s) {

      MessageQueueState state = (MessageQueueState) s;

      if(MessageQueuePhase.isSendingState(state.getPhase())) {
         return new MessageQueueState(MessageQueuePhase.inactive, State.INFINITY,
         state.getQueue(), state.getSimulationTime(), state.getFreeReplicas(),
         state.getRequestsInProcess());
      }

      if(MessageQueuePhase.isSendingRequest(state.getPhase())) {
```

```
            TimedRequest request = state.removeNextRequest();
            int idRequest = request.getID();
            String replica = state.removeFreeReplica();
            state.addRequestInProcess(new RequestInProcess(idRequest,replica));
            return new MessageQueueState(MessageQueuePhase.waiting, State.INFINITY,
            state.getQueue(), state.getSimulationTime(), state.getFreeReplicas(),
            state.getRequestsInProcess());
         }
         return null;
      }

      protected Output outputFunction(State s) throws UndefinedPortException,
      EmptyPortException {

         MessageQueueState state = (MessageQueueState) s;

         if(MessageQueuePhase.isSendingState(state.getPhase()))
            return new Output(this.getOutputPort(0),new
            ComponentState(this.componentName, InfrastructureState.expired));

         if(MessageQueuePhase.isSendingRequest(state.getPhase())) {
            TimedRequest request = state.getNextRequest();
            double waitingTime = state.getSimulationTime() - request.getStartTime();
            Request toSend = request.getRequest();
            toSend.refreshTotalTime(waitingTime);
            return new Output(this.getOutputPort(0),toSend);
         }
         return null;
      }
   }
```

C.3 Modelo Esencial "LoadBalancer"

```
public class LoadBalancer extends rd.models.EssentialModel {

   private String componentName;

   private String applicationComponent;
   private ArrayList replicas;

   public LoadBalancer(String applicationComponent, ArrayList replicas) {
      super("LoadBalancer", new String[] {"input"}, new String[] {"output"});
      this.componentName = "LoadBalancer";
      this.applicationComponent = applicationComponent;
      this.replicas = replicas;
   }

   public LoadBalancerState getInitialState(){
      return new LoadBalancerState(LoadBalancerPhase.waiting, State.INFINITY,
      this.replicas);
   }
```

```
protected State externalTransitionFunction(State s, double e, int inputNumber,
EventContent messageContent) throws UndefinedPortException,EmptyPortException {

   LoadBalancerState state = (LoadBalancerState) s;

   if(messageContent.getClass().toString().contains("OutOfUse"))
      return new LoadBalancerState(LoadBalancerPhase.sendingState,0);

   if(LoadBalancerPhase.isWaiting(state.getPhase())) {

      Request req = (Request) messageContent;

      if(state.isIncludedInReplicasState(req.getID())) {
         state.setFreeReplicaState(req.getID());
         return new LoadBalancerState(LoadBalancerPhase.waiting,
         State.INFINITY,state.getLastReplicaUsed(),state.getReplicasState(),stat
         e.getActualRequest());
      }

      if(state.allReplicasBusy())
         return new LoadBalancerState(LoadBalancerPhase.waiting,
         State.INFINITY,state.getLastReplicaUsed(),state.getReplicasState(),stat
         e.getActualRequest());

      if(!req.componentsListEmpty() && state.existReplicaFree()) {
         req.removeFirstFromComponentsList();
         int i = state.getFreeReplica();
         state.setReplicaState(i,req.getID());
         state.setReplicaUsed(i);
         return new LoadBalancerState(LoadBalancerPhase.sendingRequest, 0,
         state.getLastReplicaUsed(), state.getReplicasState(),req);
      }

   }
   state.setSigma(state.getSigma()-e);
   return state;
}

protected State getNewState() {
   return this.getInitialState();
}

protected State internalTransitionFunction(State s) {

   LoadBalancerState state = (LoadBalancerState) s;

   if(LoadBalancerPhase.isSendingState(state.getPhase()))
      return new LoadBalancerState(LoadBalancerPhase.inactive, State.INFINITY,
      state.getLastReplicaUsed(), state.getReplicasState(),
      state.getActualRequest());

   if(LoadBalancerPhase.isSendingRequest(state.getPhase()))
      return new LoadBalancerState(LoadBalancerPhase.waiting, State.INFINITY,
      state.getLastReplicaUsed(), state.getReplicasState(),null);
```

```
      return null;

   }
   protected Output outputFunction(State s) throws UndefinedPortException,
   EmptyPortException {

      LoadBalancerState state = (LoadBalancerState) s;

      if(LoadBalancerPhase.isSendingState(state.getPhase()))
         return new Output(this.getOutputPort(0),new
         ComponentState(this.componentName, InfrastructureState.expired));

      if(LoadBalancerPhase.isSendingRequest(state.getPhase()))
         return new Output(this.getOutputPort(0),state.getActualRequest());

      return null;
   }

   public String getApplicationComponent() {
      return applicationComponent;
   }

}
```

C.4 Modelo de Ruteo "ExampleInstance1", "ExampleInstance2" y "ExampleInstance3"

```
public class ExampleInstance1 extends rd.models.RoutingModel {

   public ExampleInstance1(EssentialModel componentModel) throws
   UndefinedRoutingFunctionException {
      super("ExampleInstance1",
         new Omega(30, new int[] {1},
         new RoutingFunction(
         ExampleInstanceRoutingFunction.getRoutingFunctionSpecification())),
         componentModel);
   }

   protected void initializeState() {
      Example e = (Example) this.getE();
      this.state = e.getInitialState();
   }

}

public class ExampleInstance2 extends rd.models.RoutingModel {
   public ExampleInstance2(EssentialModel componentModel) throws
   UndefinedRoutingFunctionException {
      super("ExampleInstance2",
```

```
        new Omega(31, new int[] {1},
        new RoutingFunction(
        ExampleInstanceRoutingFunction.getRoutingFunctionSpecification())),
        componentModel);
   }
   protected void initializeState() {
      Example e = (Example) this.getE();
      this.state = e.getInitialState();
   }

}

public class ExampleInstance3 extends rd.models.RoutingModel {

   public ExampleInstance3(EssentialModel componentModel) throws
   UndefinedRoutingFunctionException {
      super("ExampleInstance3",
      new Omega(32, new int[] {1},
      new RoutingFunction(
      ExampleInstanceRoutingFunction.getRoutingFunctionSpecification())),
      componentModel);
   }

   protected void initializeState() {
      Example e = (Example) this.getE();
      this.state = e.getInitialState();
   }

}
```

C.5 Modelo de Ruteo “LoadBalancerInstance1”

```
public class LoadBalancerInstance1 extends RoutingModel {

   public LoadBalancerInstance1(EssentialModel componentModel, ArrayList
   replicasId) throws UndefinedRoutingFunctionException {
      super("LoadBalancerInstance1",
      new Omega(1, new int[] {30,31,32},
      new RoutingFunction(
      LoadBalancerInstanceRoutingFunction.getRoutingFunctionSpecification(
      replicasId))), componentModel);
   }

   protected void initializeState() {
      LoadBalancer lb = (LoadBalancer) this.getE();
      this.state = lb.getInitialState();
   }

}
```

C.6 Modelo de Ruteo "MessageQueueInstance1"

```
public class MessageQueueInstance1 extends rd.models.RoutingModel {
   public MessageQueueInstance1(EssentialModel componentModel, ArrayList
   replicasId) throws UndefinedRoutingFunctionException {
      super("MessageQueueInstance1",
      new Omega(2, new int[] {30,31,32,40,41},
      new RoutingFunction(
      MessageQueueInstanceRoutingFunction.getRoutingFunctionSpecification(((
      MessageQueue) componentModel).getReplicas(),replicasId))),
      componentModel);
   }

   protected void initializeState() {
      MessageQueue mq = (MessageQueue) this.getE();
      this.state = mq.getInitialState();
   }

}
```

C.7 Modelo Atómico "MRM"

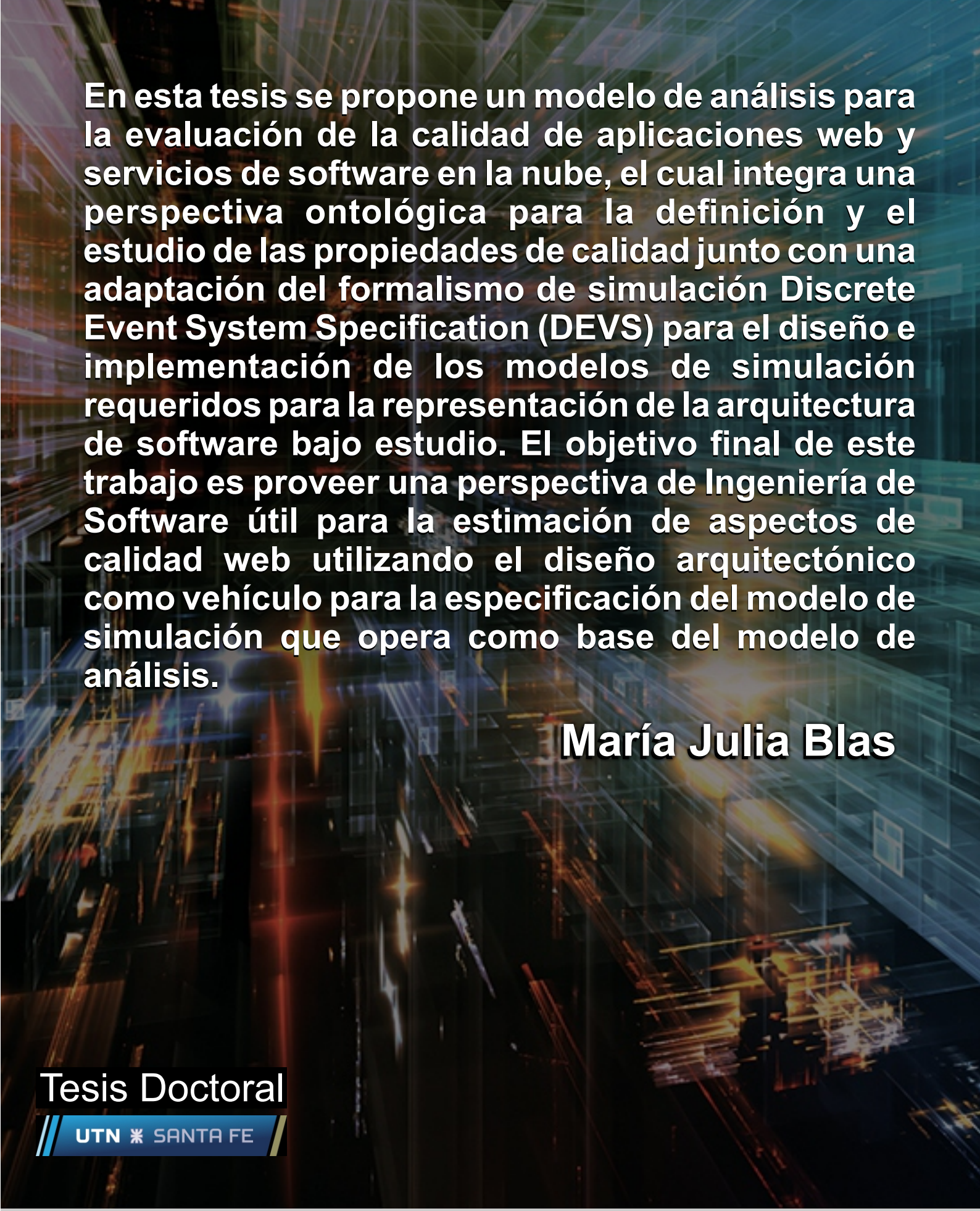
```
public class MRM extends ViewableAtomic {

   //MRM State
   private double simulationTime;
   protected int TR, IR, FNF, TF, RF;
   protected double FT, TT, ET, TSIT;
   protected int type;

   //MRM Parameters
   protected FileWriter timeMetrics = null, requestMetrics = null;
   protected LinkedList modelStructure;

   public MRM(LinkedList modelStructure,String outputFileNameTimeMetrics, String
   outputFileNameRequestMetrics){
      super("MRM");
      this.addInport("input");
      this.addOutport("output");
      this.modelStructure = modelStructure;
      try
      {
         timeMetrics = new FileWriter(outputFileNameTimeMetrics + "-log.csv" );
         timeMetrics.write( "Clock [time unit]; CR [request]; IR [request]; TR
         [request]; RA; AT [time unit]; FT [time unit]; TT [time unit]; SR; FNF
         [faults]; TF [faults]; CFT; RF [failures]; Tf [failures]; CFR\n");
         requestMetrics = new FileWriter(outputFileNameRequestMetrics+"-log.csv");
```

```
            requestMetrics.write("type [request type]; ET [time unit]; TSIS [time
            unit]; TBU;\n");
        }
        catch (IOException e) {
            System.out.println("MRM: Fail output file creation!");
        }
    }
    public void initialize() {
        super.initialize();
        this.phase = MRMPhase.waiting;
        this.sigma = INFINITY;
        this.simulationTime = 0;
        this.TR = 0; this.IR = 0; this.FNF = 0; this.TF = 0;
        this.RF = 0; this.FT = 0; this.TT = 0; this.ET = 0; this.TSIT = 0;
        this.type = -1;
    }

    public final void deltext(double e,message x){
        this.Continue(e);
        this.simulationTime += e;
        if(MRMPhase.isWaiting(this.phase)) {
            this.TT += e;
            boolean refreshFT = false, stop = false;
            for(int i=0;i<x.size();i++)
            {
                if(messageOnPort(x,"input",i))
                {
                    entity mensaje = x.getValOnPort("input",i);
                    if(mensaje.getClass().toString().contains("ComponentState"))
                    {
                        ComponentState state = (ComponentState) mensaje;
                        if(ComponentState.isRunning(state.getInfrastructureState()))
                        {
                            if(ComponentState.isFailure(state.getActivityState())) {
                                if(this.serviceInactive(state.getComponentName()))
                                    refreshFT = true;
                                this.TF++;
                            }
                            else
                            {
                                if(ComponentState.isFault(state.getProcessingState())) {
                                    this.FNF++;
                                    this.TF++;
                                }
                            }
                        }
                        else stop = true;
                    }
                    else
                    {
                            Request response = (Request) mensaje;
                            this.type = response.getType();
                            this.ET = response.getExecutionTime();
                            this.TSIT = response.getTotalTime();
                            if(response.isIncorrect()) this.IR++;
```

```
                this.TR++;
             }
          }
       }
       if(!stop) {
          if(refreshFT) this.FT += e;
          this.phase = MRMPhase.savingMetrics;
          this.sigma = 0;
       }
       else {
          this.phase = MRMPhase.stop;
          this.sigma = INFINITY;
       }
    }
    else {
       this.sigma = INFINITY;
    }
    holdIn(this.getPhase(),this.getSigma());
 }

 public final void deltint() {
    if(this.savingTimeMetrics() && this.savingRequestMetrics(this.type))
       this.phase = MRMPhase.waiting;
    else this.phase = MRMPhase.fileEror;
    this.type = -1;
    holdIn(this.getPhase(),INFINITY);
 }

 public final message out() {
    message m = new message();
    m.add(makeContent("output",new entity()));
    return m;
 }

 private boolean savingTimeMetrics() {
    if (this.timeMetrics == null ) return false;
    try
    {
       int CR = this.TR-this.IR, Tf= this.TF-this.FNF;
       double RA,AT = this.TT-this.FT,SR, CFT, CFR;
       if(this.TR!=0) RA = CR/(double) this.TR;
       else RA = -1;
       if(this.TT!=0) SR = AT /this.TT;
       else SR = -1;
       if(this.TF!=0) CFT = this.FNF /(double) this.TF;
       else CFT = -1;
       if(Tf!=0) CFR = this.RF / Tf;
       else CFR = -1;
       timeMetrics.write(Double.toString(this.simulationTime).replace('.',','));
       timeMetrics.write( ";" );
       timeMetrics.write(Integer.toString(CR));
       timeMetrics.write( ";" );
       timeMetrics.write(Integer.toString(this.IR));
       timeMetrics.write( ";" );
       timeMetrics.write(Integer.toString(this.TR));
```

```
      timeMetrics.write( ";" );
      if(RA!=-1) timeMetrics.write(Double.toString(RA).replace('.',','));
      else timeMetrics.write("");
      timeMetrics.write( ";" );
      timeMetrics.write(Double.toString(AT).replace('.',','));
      timeMetrics.write( ";" );
      timeMetrics.write(Double.toString(this.FT).replace('.',','));
      timeMetrics.write( ";" );
      timeMetrics.write(Double.toString(this.TT).replace('.',','));
      timeMetrics.write( ";" );
      if(SR!=-1) timeMetrics.write(Double.toString(SR).replace('.',','));
      else timeMetrics.write("");
      timeMetrics.write( ";" );
      timeMetrics.write(Integer.toString(this.FNF));
      timeMetrics.write( ";" );
      timeMetrics.write(Integer.toString(this.TF));
      timeMetrics.write( ";" );
      if(CFT!=-1) timeMetrics.write(Double.toString(CFT));
      else timeMetrics.write("");
      timeMetrics.write( ";" );
      timeMetrics.write(Integer.toString(this.RF));
      timeMetrics.write( ";" );
      timeMetrics.write(Integer.toString(Tf));
      timeMetrics.write( ";" );
      if(CFR!=-1) timeMetrics.write(Double.toString(CFR));
      else timeMetrics.write("");
      timeMetrics.write( ";\n" );
      timeMetrics.flush();
   }
   catch (IOException e) {
      System.out.println("Output file: Missing data.");
   }
   return true;
}

private boolean savingRequestMetrics(int request) {
   if(request!=-1){
      double TBU;
      if(this.TSIT!=0) TBU = this.ET / this.TSIT;
      else TBU = -1;
      if (this.requestMetrics == null ) return false;
      try
      {
         requestMetrics.write(Integer.toString(this.type));
         requestMetrics.write( ";" );
         requestMetrics.write(Double.toString(this.ET).replace('.',','));
         requestMetrics.write( ";" );
         requestMetrics.write(Double.toString(this.TSIT).replace('.',','));
         requestMetrics.write( ";" );
         if(TBU!=-1)
            requestMetrics.write(Double.toString(TBU).replace('.',','));
         else timeMetrics.write("");
         requestMetrics.write( ";\n" );
         requestMetrics.flush();
      }
```

```
            catch (IOException e) {
               System.out.println("Output file: Missing data.");
            }
            return true;
         }
         return true;
      }
      private boolean serviceInactive(String componentName) {
         int aux = this.modelStructure.size();
         for(int i=0;i<aux;i++) {
            ComponentReplicas cr = (ComponentReplicas) this.modelStructure.get(i);
            if(cr.isApplicationComponent(componentName)) {
               this.modelStructure.remove(i);
               cr.decreaseReplicasNumber();
               if(cr.isInactive()) return true;
            }
         }
         return false;
      }

}
```



En esta tesis se propone un modelo de análisis para la evaluación de la calidad de aplicaciones web y servicios de software en la nube, el cual integra una perspectiva ontológica para la definición y el estudio de las propiedades de calidad junto con una adaptación del formalismo de simulación Discrete Event System Specification (DEVS) para el diseño e implementación de los modelos de simulación requeridos para la representación de la arquitectura de software bajo estudio. El objetivo final de este trabajo es proveer una perspectiva de Ingeniería de Software útil para la estimación de aspectos de calidad web utilizando el diseño arquitectónico como vehículo para la especificación del modelo de simulación que opera como base del modelo de análisis.

María Julia Blas

Tesis Doctoral

UTN ✱ SANTA FE

edUTecNe

L U A Libro Universitario Argentino

CiN REUN Red de Editoriales de Universidades Nacionales de la Argentina