Proyecto Fin de Carrera de Ingeniería en Informática

# Implementación de un generador de estrategias de autoconfiguración para la mejora de prestaciones de software autoadaptativo


María Estíbaliz Fraca Santamaría

Director: Diego Carmelo Pérez Palacín

Codirector: José Javier Merseguer Hernáiz

Departamento de Informática e Ingeniería de Sistemas
Centro Politécnico Superior
Universidad de Zaragoza


Marzo 2010

# Implementación de un generador de estrategias de autoconfiguración para la mejora de prestaciones de software autoadaptativo

# RESUMEN

Dentro del campo de la Ingeniería del Software, este trabajo se enmarca en el paradigma *open-world software*. El objetivo de este trabajo es estudiar cómo satisfacer propiedades extra-funcionales en este tipo de sistemas, en concreto la propiedad de rendimiento o prestaciones. Es este un tema de investigación nuevo, y todavía se encuentra lejos de estar resuelto. Con este trabajo intentamos avanzar en lo que se refiere al desarrollo de herramientas que faciliten y automaticen dichas tareas de evaluación.

El paradigma de desarrollo *open-world software* propone la creación de sistemas *software* heterogéneos y distribuidos donde el entorno de ejecución cambia de forma continua e impredecible (por ejemplo, en términos de disponibilidad de servicios y degradación o aumento de las prestaciones de los mismos). Además, el paradigma entiende que los sistemas deben (a) ser conscientes de esos cambios en el entorno y (b) ser capaces de reaccionar ante ellos modificando su comportamiento, es decir, deben ser autoreconfigurables o autoadaptativos. Estos sistemas pueden seguir *estrategias de reconfiguración* para decidir cuándo y cómo hacer su adaptación . Un ejempo de *open-world software* son las arquitecturas del estilo SOA (Arquitectura Orientada a Servicios), donde el *software* a construir requiere servicios que son proporcionados por otras aplicaciones.

Este trabajo consiste en la implementación de una parte de la arquitectura propuesta en [1] por Pérez-Palacín, Merseguer y Bernardi para la creación de estrategias de reconfiguración para software autoadaptativo. El modelado de la arquitectura se ha realizado siguiendo el estándar UML aumentado con información sobre prestaciones utilizando el estándar "*Modeling and Analysis of Real-Time and Embedded systems*" (MARTE). La evaluación de los sistemas software se ha realizado utilizando el método formal de las redes de Petri estocásticas, que han sido generadas a partir de de los modelos UML anotados con MARTE. El resultado de esa evaluación sirve para crear la estrategia de reconfiguración, la cual indica al sistema que se reconfigure o permanezca el estado actual.

El resultado obtenido ha sido la creación de un paquete ejecutable preparado para ser integrado en sistemas autoadaptativos que siguen la arquitectura de tres capas utilizada. Este *software* desarrollado corresponde a la capa más desafiante y todavía desconocida de la arquitectura, la que conoceremos como "generador de estrategias". Mediante la realización de este paquete ejecutable se ha dado un paso adelante hacia la implementación real de los sistemas autoadaptativos en base a sus prestaciones.

El resultado generado por este paquete es una estrategia de reconfiguración que dirigirá las reconfiguraciones del *software* autoadaptativo reaccionando ante los cambios percibidos en el entorno del sistema.

# *Agradecimientos*

*Quiero agradecer en primer lugar su apoyo, su ayuda y su interés en mi proyecto a Diego Pérez y a José Merseguer. A Diego por las videoconferencias de una hora, por resolver mis dudas justo al poco rato de aterrizar de un avión, por emocionarse con las redes de Petri contagiando su ilusión. A José por su constancia, por resolver con sencillez y eficacia las cuestiones que para mi eran un mundo, por sus palabras de ánimo, por su rapidez en responder emails y solucionar problemas.*

*También a quienes me han acompañado en mi estancia diaria en el L1.03a: Ricardo, que aun con mil cosas en la cabeza simpre saca tiempo para un café; Misho, y las conversaciones profundas durante las comidas; Hanife, con su inocencia y sinceridad; y Simona, que apareció justo cuando necesité ayuda con las redes de Petri.*

*No quiero olvidarme de mis compañeros/as de carrera durante estos cinco años y pico... Alberto, Adrián, Pablo, Jorge, mis doce diferentes compañeros/as de prácticas, y otros muchos que no nombro por temor a dejarme a alguien.*

*Finalmente, quiero agradecer su apoyo y confianza en mí a Elisardo, Blanca y David; a Óscar; a Mónica, Silvia, Fernando y Chema; a Erika, a María, a Anna, a Nacho... Gracias.*

# Índice general


**I Memoria** XI

**1. Introducción** 1
1.1. Contexto . . . 1
1.2. Objetivo y alcance del proyecto . . . 1
1.3. Fases de desarrollo . . . 2
1.4. Ámbito y motivación . . . 3
1.5. Métodos y herramientas empleados . . . 3
1.5.1. Lenguajes y estándares . . . 3
1.5.2. Herramientas . . . 4
1.6. Organización del documento . . . 4

**2. Conceptos previos** 5
2.1. UML . . . 5
2.1.1. Diagrama de Actividad . . . 5
2.1.2. Diagrama de Componentes . . . 5
2.2. MARTE . . . 6
2.3. XML . . . 7
2.4. XMI . . . 7
2.5. Redes de Petri . . . 7
2.6. Open-world software . . . 8
2.7. Sistemas auto-adaptativos . . . 9
2.8. Arquitectura de tres capas para sistemas auto-adaptativos . . . 9

**3. Planteamiento del problema** 11
3.1. Capa de Control de Componentes . . . 11
3.2. Capa de Gestión del Cambio . . . 12
3.3. Capa de Gestión de Objetivos . . . 12

**4. Implementación del Generador de Estrategias** 15
4.1. Obtención de las precondiciones del algoritmo . . . 15
4.1.1. Elección de una herramienta de modelado UML . . . 16
4.1.2. Lectura de los ficheros XMI . . . 16
4.1.3. Obtención de la información temporal de los componentes . . . 16
4.2. Estrategia de reconfiguración . . . 17
4.3. Descripción de la generación de estrategias . . . 18
4.4. Evaluación de los *workflow* . . . 21

4.4.1. Traducción de los flujos de ejecución a redes de Petri . . . . . . . . . . . . . 21
4.4.2. Evaluación de las redes de Petri . . . . . . . . . . . . . . . . . . . . . . . . . 22

**5. Planificación temporal. 25**

**6. Conclusiones y trabajo futuro 27**
6.1. Conclusiones y resultados obtenidos. . . . . . . . . . . . . . . . . . . . . . . . . . 27
6.2. Trabajo futuro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**Bibliografía 30**

**II Apéndices 31**

**A. Caso práctico 33**
A.1. Descripción del problema a resolver . . . . . . . . . . . . . . . . . . . . . . . . . 33
A.2. Obtención de los datos de entrada . . . . . . . . . . . . . . . . . . . . . . . . . . 34
A.2.1. Obtención del diagrama de Actividades anotado con MARTE . . . . . . . . 35
A.2.2. Obtención del diagrama de Componentes . . . . . . . . . . . . . . . . . . . 36
A.2.3. Almacenamiento de los datos durante la implementación . . . . . . . . . . . 38
A.3. Descripción de la generación de estrategias . . . . . . . . . . . . . . . . . . . . . 39
A.4. Estrategia de reconfiguración obtenida . . . . . . . . . . . . . . . . . . . . . . . . 41

**B. El paquete ejecutable desarrollado 43**
B.1. El paquete ejecutable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
B.2. Diagrama de clases del componente desarrollado . . . . . . . . . . . . . . . . . . 44

**C. Fases de desarrollo. Hitos y problemas encontrados 47**

**D. MARTE 49**

**E. Traducción de los diagramas de Actividad de redes de Petri 51**

**F. Empleo de la herramienta GreatSPN 57**
F.1. Formato de los ficheros de GreatSPN . . . . . . . . . . . . . . . . . . . . . . . . . 57
F.2. Invocación del simulador . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**G. Glosario y abreviaturas empleadas 61**

# Índice de figuras


2.1. Ejemplo de Diagrama de Actividades . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2. Ejemplo de Diagrama de Componentes . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3. Ejemplo de fichero XML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4. Ejemplo de red de Petri . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5. Esquema de adaptación externa para sistemas autoadaptativos . . . . . . . . . . . . 9
2.6. Esquema de la arquitectura de tres capas propuesta por Kramer y Merge . . . . . . . 10

3.1. Arquitectura de tres capas aplicada al paradigma *open-world* . . . . . . . . . . . . . 11

4.1. Ejemplo de estrategia de reconfiguración . . . . . . . . . . . . . . . . . . . . . . . . 18
4.2. Ejemplo de traducción de elementos de un diagrama de Actividades a LGSPN . . . . 22
4.3. Ejemplo de composición de LGSPN. Se remarcan los elementos compuestos en uno solo 23
4.4. Red de Petri a evaluar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

5.1. Diagrama de Gantt del desarrollo del proyecto . . . . . . . . . . . . . . . . . . . . . 25
5.2. Distribución del esfuerzo en las fases de desarrollo . . . . . . . . . . . . . . . . . . 25

A.1. Diagrama de Actividad con anotaciones MARTE . . . . . . . . . . . . . . . . . . . . 33
A.2. Diagrama de Componentes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
A.3. Diagrama de Actividad modelado en la herramienta Papyrus con anotaciones MARTE 35
A.4. Componentes e interfaces del diagrama de Componentes . . . . . . . . . . . . . . . 36
A.5. Diagrama de Componentes modelado en la herramienta Papyrus . . . . . . . . . . . 37
A.6. Diagrama de Clases de la implementación del diagrama de Actividad . . . . . . . . . 38
A.7. Diagrama de Clases de la implementación del diagrama de Componentes . . . . . . 38
A.8. GSPN paramétrica obtenida a partir del *workflow* del sistema . . . . . . . . . . . . . 39
A.9. Estrategia de reconfiguración obtenida . . . . . . . . . . . . . . . . . . . . . . . . . 41

B.1. Componente software desarrollado . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
B.2. Diagrama de Clases del componente desarrollado . . . . . . . . . . . . . . . . . . . 45

D.1. Anotación GaWorkloadEvent de MARTE . . . . . . . . . . . . . . . . . . . . . . . . 49
D.2. Anotación Resource de MARTE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
D.3. Anotación GaAcqStep de MARTE . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
D.4. Anotación GaRelStep de MARTE . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
D.5. Anotación GaRelStep de MARTE . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

E.1. Diagrama de Actividades a traducir . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
E.2. Red de Petri obtenida de la traducción del diagrama de Actividad . . . . . . . . . . . 52
E.3. Traducción a RdP de los elementos de Acción . . . . . . . . . . . . . . . . . . . . . 53

E.4. Traducción a RdP de los elementos de Transición . . . . . . . . . . . . . . . . . . . . 54
E.5. Traducción a RdP del elemento Recurso . . . . . . . . . . . . . . . . . . . . . . . . . 54

F.1. Red de Petri correspondiente a simpleGSPN.def y simpleGSPN.net . . . . . . . . . 57

# Parte I

# Memoria

Capítulo 1

# Introducción

El objetivo de este Proyecto Fin de Carrera (PFC) ha sido la implementación de un generador de estrategias de autoconfiguración[1] para la mejora de prestaciones de *software* autoadaptativo dentro del paradigma de *software open-world* [2].

La piedra angular es el algoritmo propuesto en [1], que implementa la capa de *gestión de objetivos* dentro de una arquitectura de referencia. La arquitectura propone tres capas para describir las funcionalidades del *software* autoadaptativo. El algoritmo requiere información sobre el sistema representada en diagramas UML anotados con el perfil MARTE [3], y los traduce a Redes de Petri [4] con el objetivo de estimar sus prestaciones.

## 1.1. Contexto

El paradigma de desarrollo *open-world software* [2] propone la creación de sistemas *software* heterogéneos y distribuidos donde el entorno de ejecución cambia de forma continua e impredecible (por ejemplo, en términos de disponibilidad de servicios y degradación o aumento de las prestaciones de los mismos). Un ejempo de ello son las arquitecturas del estilo SOA (Arquitectura Orientada a Servicios), donde el *software* a construir requiere servicios que son proporcionados por otras aplicaciones.

Debido a esta dependencia de servicios externos, el rendimiento de la aplicación vendrá fuertemente condicionado por el rendimiento de terceras aplicaciones, siendo difícil pero interesante predecir su modo de funcionamiento, precisamente en términos de la calidad del servicio (QoS) que podrá proveer. Por ello, los sistemas deben ser conscientes de esos cambios en el entorno y ser capaces de reaccionar ante ellos modificando su comportamiento, es decir, deben ser auto-adaptables o auto-reconfigurables.

En [5] se propone una arquitectura software de tres capas: capa de control de componentes (capa inferior), capa de gestión del cambio (capa intermedia) y capa de gestión de objetivos (capa superior); para crear *software* autoconfigurable (ver fig. 2.6). Finalmente, en [1], la arquitectura se adapta para posibilitar que el software desarrollado sea capaz de autoconfigurarse en respuesta a las variaciones de las prestaciones de las terceras partes de las que obtiene los servicios requeridos (ver capítulo 3 y fig. 3.1).

## 1.2. Objetivo y alcance del proyecto

El objetivo principal consiste en la implementación del algoritmo de generación de estrategias de autoconfiguración propuesto en [1], dentro de la capa de gestión de objetivos de la arquitectura

[1] Los conceptos “estrategia de reconfiguración” y “estrategia de autoconfiguración” se emplean indistintamente a lo largo de la memoria. Estrictamente, se trata de estrategias para la reconfiguración de componentes *software*.

referida. Se trata de la capa más interesante y desafiante de la arquitectura. Este algoritmo será capaz de reaccionar frente a cambios en las prestaciones que puedan sufrir las diferentes entidades del entorno *open-world*.

El objetivo será crear, a partir de estos cambios, la mejor estrategia de autoconfiguración. Empleando dicha estrategia el *software* será capaz de ordenar a las capas inferiores (control de componentes y gestión del cambio) que las peticiones de los servicios requeridos se dirijan a los componentes que ofrezcan mejores prestaciones globales al sistema.

Para ello, habrá que utilizar una herramienta de modelado UML que permita crear diagramas (de Actividad y de Componentes) anotados con respecto al perfil de MARTE [3] y los exporte en formato XMI[6]. Estos diagramas proporcionarán la información necesaria a procesar por el algoritmo. Los diagramas de Actividad representarán el *workflow* del sistema, los cuales se convertirán en redes de Petri[4], que podrán ser evaluadas gracias a la herramienta GreatSPN [7]. La información de evaluación servirá para generar la estrategia de reconfiguración que aporte un mejor rendimiento a la aplicación *software open-world*.

## 1.3. Fases de desarrollo

El trabajo se ha desarrollado en diferentes etapas, que se presentan a continuación:

1. **Definición del problema y estudio de la documentación**
   Durante la primera fase se definió y acotó el problema y se procedió al estudio de dos artículos de investigación: *Performance evaluation of self-reconfigurable service-oriented software with stochastic Petri nets* [8] y *Performance Aware Open-world Software in a 3-Layer Architecture* [1], así como de otra documentación de referencia sobre el paradigma *open-world* [2] y los sistemas autoadaptativos [5, 9].

2. **Obtención de los diagramas UML**
   Esta fase consistió en obtener la información necesaria que serviría de entrada al algoritmo que se implementaría en la etapa 5. Para ello, en primer lugar se eligió una herramienta CASE que permitiera dibujar y exportar diagramas UML con anotaciones de prestaciones MARTE. A continuación, se manejó dicha herramienta para modelar los diagramas necesarios (diagramas de Actividad con anotaciones de prestaciones y diagramas de Componentes). Estos fueron exportados en formato XMI y finalmente procesados como entrada del algoritmo empleando árboles DOM, un método de procesado de ficheros XML. Los datos sobre prestaciones asociados al Diagrama de Componentes fueron obtenidos a partir de un fichero XML.

3. **Evaluación de *workflows* mediante su traducción a Redes de Petri**
   Esta fase engloba:

   - Traducción de los *workflows*, representados en forma de diagramas de Actividades y obtenidos según se explica en la sección 4.1, a Redes de Petri.
   - Estudio y manejo de la herramienta GreatSPN [7].
   - Evaluación de las Redes de Petri mediante su simulación con GreatSPN.

4. **Definición y documentación de la estrategia de autoconfiguración**
   Antes de comenzar con la implementación del algoritmo se definió el formato de salida del mismo (ver sección 4.2), esto es, la estrategia de autoconfiguración. Esta estrategia podrá ser usada por las capas inferiores de la arquitectura para llevar a cabo la reconfiguración del sistema.

5. **Implementación del algoritmo de generación de estrategias y pruebas**
   Finalmente, tras implementar la obtención de los datos de entrada (fase 2) y tras definir los datos de salida (fase 4), se realizó la implementación del algoritmo principal.

## 1.4. Ámbito y motivación

Este Proyecto Fin de Carrera se enmarca dentro del trabajo de investigación del estudiante de doctorado Diego Pérez-Palacín y su director el Dr. José Merseguer. En concreto, este Proyecto consiste en la implementación del Generador de Estrategias para la arquitectura propuesta por Pérez-Palacín, Merseguer y Bernardi en [1] para software autadaptativo en base a prestaciones. Todo ello se ubica dentro del Grupo de Ingeniería de Sistemas de Eventos Discretos (GISED), en el Departamento de Informática e Ingeniería de Sistemas (DIIS) de la Universidad de Zaragoza.

La principal motivación para la elección de este PFC fue la oportunidad de profundizar en el aprendizaje y estudio de las Redes de Petri, más allá de lo estudiado en las asignaturas de la carrera. Este Proyecto me ha ofrecido la oportunidad de explorar una interesante aplicación: la evaluación del rendimiento del software.

Otra razón que me llevó a elegir este Proyecto fue que versara sobre algo novedoso y en boga como es el *software open-world*. Me parece interesante la búsqueda de nuevas aplicaciones y métodos para este paradigma.

Por último, decidí elegir un Proyecto Fin de Carrera que me acercase al mundo de la investigación en informática, un ámbito que me interesaba conocer de cara a elegir hacia dónde enfocar mi carrera profesional.

## 1.5. Métodos y herramientas empleados

Durante el desarrollo de este trabajo se han empleado diversos lenguajes y herramientas de los cuales se destacan las más relevantes.

### 1.5.1. Lenguajes y estándares

- **UML** Lenguaje Unificado de Modelado (**U**nified **M**odeling **L**anguage) [10, 11]. Define un lenguaje gráfico (en forma de diagramas), empleado para construir, documentar, visualizar y especificar sistemas *software*. Los diagramas utilizados han sido el de Actividades, para representar el *workflow* del sistema y el de Componentes, para representar los servicios requeridos por el sistema y sus proveedores. Se ha empleado la versión 2 del estándar.

- **MARTE** [3]. Perfil de UML para el Modelado y Análisis de Sistemas Embebidos y de Tiempo Real (**M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded systems). Ha sido empleado para añadir anotaciones de prestaciones a los diagramas UML.

- **XMI** (**X**ML **M**etadata **I**nterchange) [6]. Se trata de un lenguaje de especificación para el intercambio datos entre herramientas de modelado UML. Está basado en el estándar XML[12], un estándar ISO.

- **Redes de Petri** [4]. Son una herramienta matemática que sirve para modelar sistemas cuyos comportamientos dinámicos se caracterizan por la concurrencia, la sincronización, la exclusión mutua y los conflictos.

- **JAVA** [13]. Lenguaje multiplataforma y orientado a objetos empleado (en su versión 1.5.0_02.).

### 1.5.2. Herramientas

- **Papyrus** [14]. Herramienta de modelado UML. Se alinea con el estándar UML 2 y soporta anotaciones con diferentes perfiles (UML *profiles*). Desde nuestro conocimiento es la única que soporta anotaciones del estándar MARTE. Se ha empleado la versión 1.11.
- **GreatSPN** [7]. Herramienta de modelado, validación y evaluación de Redes de Petri estocásticas generalizadas (GSPN) y su extensión coloreada. Se ha empleado la versión 2.0.

## 1.6. Organización del documento

El presente documento está dividido en dos partes: la memoria, donde se explica el desarrollo del Proyecto; y los apéndices, donde se amplía la información de ciertos puntos relevantes.

El capítulo 1 expone los objetivos del proyecto e introduce el documento. El capítulo 2 define los algunos conceptos previos que servirán de ayuda para comprender el resto del documento, como UML, red de Petri, o sistema autoadaptativo. A continuación, el capítulo 3 explica la arquitectura de tres capas en la que se enmarca el trabajo. El capítulo 4 explica la implementación del algoritmo Generador de Estrategias, así como la obtención de los datos de entrada y el significado de los datos de salida. En el capítulo 5 se hace balance del esfuerzo temporal empleado en la realización del PFC. Finalmente, el capítulo 6 presenta las conclusiones de carácter personal y técnico obtenidas durante la elaboración de este Proyecto Fin de Carrera; así como una serie de trabajos futuros que complementarían y ampliarían el trabajo desarrollado.

Respecto a los apéndices, el apéndice A es un caso práctico que muestra algunos detalles técnicos (modelado de diagramas con Papyrus, formato de los fichros XML, etc); y detalla la generación de estrategias través de un ejemplo. El apéndice B expone el paquete software obtenido como resultado del PFC y explica su empleo y parametrización. Además, muestra su diagrama de Clases. A continuación, el apéndice D enumera y explica las anotaciones MARTE empleadas en este trabajo. Los apéndices E y F detallan la traducción de los diagramas de Actividad a redes de Petri y explican el empleo de la herramienta GreatSPN para la evaluación de las redes. Por último, el apéndice G define algunos conceptos y abreviaturas empleados en la memoria.

Capítulo 2

# Conceptos previos

En el presente capítulo se presentan los conceptos más relevantes dentro de cada una de las áreas sobre las que trata este Proyecto Fin de Carrera. Lo explicado en este capítulo quiere servir de base para comprender el trabajo realizado que se desarrollará en los siguientes capítulos.

## 2.1. UML

El estándar UML es una especificación semi-formal que define un lenguaje gráfico que sirve para visualizar, especificar, construir y documentar los elementos de un sistema [10]. Fue adoptado por *Object Management Group* (OMG) en 1998 y es un estándar ISO. Actualmente se encuentra en su versión 2.

Proporciona soporte para la planificación y control del ciclo completo de vida del *software*, independientemente de la plataforma para la que se desarrolla. En él también se definen reglas semánticas y de corrección de modelos a través del lenguaje OCL (Object Constraint Language).

UML define doce tipos distintos de diagramas gráficos que sirven para describir las vistas que un modelo puede necesitar para ser caracterizado, enfocados desde el paradigma Orientado a Objetos (OO). Los empleados en este proyecto han sido diagrama de Actividad y diagrama de Componentes.

### 2.1.1. Diagrama de Actividad

Los diagramas de Actividad se utilizan para modelar los aspectos dinámicos de un sistema (los pasos secuenciales y concurrentes de un proceso computacional).

Son un caso especial de las máquinas de estados en los que únicamente hay actividades y no se producen cambios de estado en el objeto que realiza dichas actividades. En la figura 2.1 se muestra ejemplo un diagrama de actividades.

Los elementos principales de este tipo de diagramas son las actividades o acciones, que representan cada uno de los pasos de un proceso (en la figura se representa con rectángulos redondeados). Cabe destacar que podemos encontrarnos con actividades secuenciales o concurrentes. En el ejemplo anterior (ver figura 2.1) algunas actividades son ejecutadas de manera secuencial y otras que lo hacen manera concurrente, a través de los elementos de *fork* y *join* (bifurcación y unión) representados mediante cajas verticales. La selección entre dos rutas alternativas y el fin de una selección se representa con un cuadrado rotado 45º.

### 2.1.2. Diagrama de Componentes

Los diagramas de Componentes se utilizan para representar cómo un sistema *software* es dividido en componentes. Normalmente contienen interfaces, componentes y relaciones entre ellos; siendo un componente una parte sustituible de un sistema que realiza una interface y que puede ser reemplazable

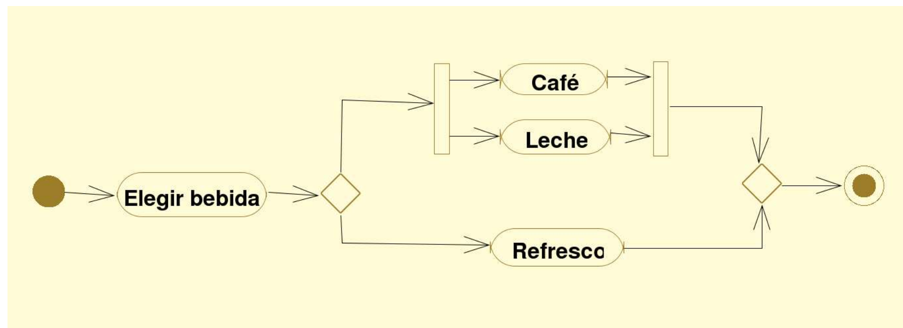


Figura 2.1: Ejemplo de Diagrama de Actividades

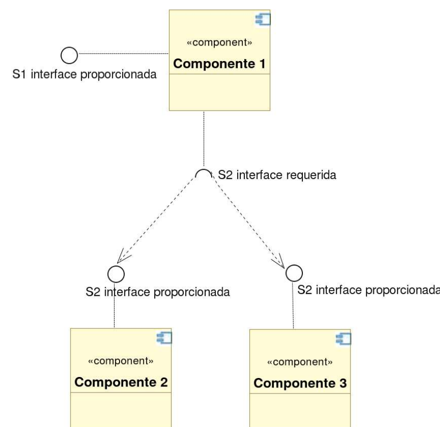


Figura 2.2: Ejemplo de Diagrama de Componentes

por otro que realice la misma interface. Y siendo una interface una especificación para las operaciones externas visibles de un componente, sin especificar su estructura interna.

En la figura 2.2 se muestra un ejemplo de un Diagrama de Componentes en el que se pueden ver los componentes, las interfaces requeridas y proporcionadas por ellos, y las relaciones entre ellos. Las interfaces ofrecidas por los componentes se representan como circunferencias, mientras que las interfaces requeridas por los componentes se representan con semicircunferencias.

## 2.2. MARTE

MARTE (***M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded systems*) [3]. Es un perfil de UML 2.0 para el modelado y análisis de sistemas embebidos y de tiempo real, incluidos los aspectos *software* y *hardware*. Proporciona soporte para las etapas de especificación, diseño y verificación.

El estándar MARTE ha sido desarrollado por OMG para extender las capacidades de UML para el desarrollo de sistemas de tiempo real y sistemas embebidos. Tiene el objetivo de proporcionar un estándar común de modelado para mejorar tanto la comunicación entre desarrolladores como la interoperabilidad entre herramientas.

En este proyecto el perfil MARTE se emplea para añadir anotaciones de prestaciones (anotaciones temporales) a los diagramas UML. En el anexo D se amplía información acerca de las anotaciones MARTE empleadas en este PFC.

## 2.3. XML

El estándar XML (**Ex**tensible **M**etadata **L**anguage) [12] es un metalenguaje basado en etiquetas que permite la definición de lenguajes para diferentes necesidades. Algunos lenguajes y especificaciones basados en XML son XHTML, SVG o XMI.

```
<fecha>
     <dia> 8 </dia>
     <mes> Mayo </mes>
     <año> 2010 </año>
</fecha>
```

Figura 2.3: Ejemplo de fichero XML

Algunas de las ventajas que proporciona el empleo de XML son su generalización, su carácter estándar como lenguaje de intercambio de datos y su simplicidad de uso tanto por humanos (es fácil de comprender al estar basado en texto) como por máquinas (mediante su procesado mediante SAX o árboles DOM, su transformación mediante XSLT, u otros). La figura 2.3 muestra un ejemplo de un fichero XML.

Existen dos aproximaciones para el procesado de XML: SAX y DOM, caracterizadas por diferentes ventajas y desventajas. En este proyecto se ha empleado DOM debido a que los ficheros a tratar no son especialmente grandes y a que se realizan muchas consultas sobre el mismo fichero XML.

## 2.4. XMI

XMI (**X**ML **M**etadata **I**nterchange) [6] es una especificación basada en XML para el intercambio de modelos entre diferentes herramientas de modelado UML, desarrollada por OMG.

Los diagramas UML empleados en este proyecto fueron obtenidos a través de la herramienta de modelado Papyrus y exportados en el formato XMI. Una vez obtenido el fichero XMI, éste fue procesado.

## 2.5. Redes de Petri

Las redes de Petri (RdP) [4] son una herramienta matemática que sirve para modelar sistemas cuyos comportamientos dinámicos se caracterizan por la concurrencia, la sincronización, la exclusión mutua y los conflictos.

Poseen una representación gráfica en forma de grafo dirigido en el cual los nodos pueden ser o lugares, representados como círculos, o transiciones, representadas como barras o cajas. Los arcos entre nodos pueden estar dirigidos desde un lugar hacia una transición o desde una transición hacia un lugar. Además, un lugar puede tener una o más marcas. Los lugares suelen describir estados del sistema, mientras que las transiciones se pueden interpretar como los eventos que modifican un estado determinado del sistema.

El comportamiento dinámico de las RdP está dirigido por la regla de disparo. Una transición puede dispararse si todos sus lugares de entrada contienen al menos una marca o *token* (o tantas marcas como las indicadas en el arco que lo une con la transición), entonces se dice que la transición está sensibilizada. Tras dispararse la transición, se eliminan tantas marcas como las indicadas en cada arco y se generan nuevas marcas en los lugares de salida.

En este trabajo se emplean redes de Petri estocásticas generalizadas (GSPN); caracterizadas por:

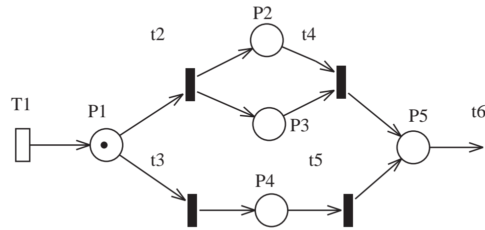


Figura 2.4: Ejemplo de red de Petri

1. Las transiciones pueden ser de dos tipos: temporizadas (representadas en color blanco) e inmediatas (representadas en color negro).

2. Dos transiciones en conflicto pueden ser equiprobables o bien tener diferentes pesos asociados.

3. Las transiciones pueden tener diferente prioridad. Al añadir este concepto, la condición de sensibilización de transiciones para un marcado *m* se restringe, y solo quedan sensibilizadas en *m* las transiciones que tienen mayor prioridad. Toda transición inmediata tiene más prioridad que cualquier transición temporizada.

## 2.6. Open-world software

Tradicionalmente, los paradigmas de desarrollo del *software* han asumido un entorno cerrado, donde el *software* contaba con requisitos y circunstancias conocidos e invariables. Éstos permanecían estables desde la definición del problema y durante la ejecución del mismo. Sin embargo, estas suposiciones dejan de ser válidas para un número cada vez más amplio de casos. Por ejemplo, en dispositivos portátiles o integrados en otros aparatos de uso común (computación obicua o *ubiquitous computing*), donde el entorno es esencialmente abierto.

Teniendo en cuenta estas características, aparece el paradigma *open-world* [2]. El método tradicional de desarrollo de *software* (con las etapas de definición de requisitos, especificación, implementación y pruebas sobre esos requisitos) no es lo suficientemente flexible para este tipo de problemas. El *open-world software* propone la creación de sistemas *software* heterogéneos y distribuidos, donde el entorno de ejecución, que es la red, cambia de forma continua e impredecible (por ejemplo, en términos de disponibilidad de servicios y degradación o aumento de las prestaciones de los mismos).

En el *open-world*, el mundo es intrínsecamente abierto; por lo tanto, las circunstancias (o el entorno) del *software* pueden cambiar en cualquier momento de manera imprevista.Pueden aparecer nuevos componentes, su comportamiento puede verse modificado y los componentes accesibles pueden dejar de estarlo; todo ello durante el tiempo de ejecución. Por ello, el *software* debería reconocer esos cambios y reaccionar ante ellos de manera autónoma y en tiempo de ejecución.

Desde una perspectiva histórica, el desarrollo del *software* ha ido evolucionando hacia ser cada vez más cada vez más modular y distribuído. Partiendo de la programación estructurada, pasando por los lenguajes OO, y llegando hasta la programación basada en servicios *open-world*.

## 2.7. Sistemas auto-adaptativos

Los sistemas *software* auto-adaptativos son aquellos capaces de modificar su comportamiento dinámicamente, y por ello se necesitan técnicas y mecanismos para que puedan adaptarse a cambios en su entorno.

Algunos mecanismos que soportan una cierta auto-adaptación son ya conocidos, ampliamente empleados, y proporcionados por ciertos lenguajes de programación o librerías *run-time*: manejo de excepciones, comprobación de guardas (*runtime assertion checking*), etc. Estos métodos son adecuados para reconocer y evitar errores, pero no reconocen anomalías más sutiles como la degradación gradual de las prestaciones o la pérdida de fiabilidad.

Con el objetivo de lograr mecanismos más sensibles y sutiles, en [9] se propone "adaptar" el sistema de manera externa, según se muestra en la figura 2.5. El sistema es monitorizado por un componente externo que determina si la ejecución cumple unos parámetros aceptables, y en caso de que no lo haga, es el responsable de reparar el sistema.

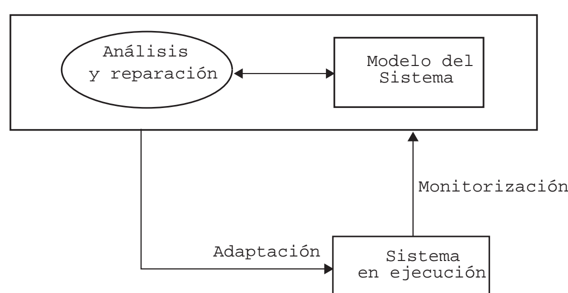


Figura 2.5: Esquema de adaptación externa para sistemas autoadaptativos

Algunas ventajas de esta aproximación son: su reusabilidad, debido a su independencia del sistema; que diferentes modelos pueden ser empleados dependiendo qué se quiere comprobar; y que pueden ser mantenidos con independencia del sistema.

## 2.8. Arquitectura de tres capas para sistemas auto-adaptativos

Kramer y Magee proponen en [5, 15] una arquitectura de referencia basada en tres capas para sistemas auto-adapatativos, capaz de detectar cambios dinámicos que se produzcan durante la ejecución del sistema y autoconfigurarse para continuar satisfaciendo su especificación.

Esta arquitectura está inspirada en otras arquitecturas desarrolladas en el campo de la robótica, en particular la descrita por Gat [16]. Kramer y Magee reconocieron que tanto los sistemas de robótica como los sistemas *software* auto-adaptativos eran cierto tipo de sistemas autónomos. Bajo esa idea, adaptaron una arquitectura de tres capas empleada tradicionalmente en robótica al ámbito de los sitemas auto-adaptativos.

En su propuesta no definen una implementación concreta, sino que proponen una arquitectura de referencia (esquematizada en la figura 2.6) que intenta satisfacer las necesidades de los sistemas auto-adapatativos y que puede ser adaptada a diferentes problemas. A continuación se describen las tres capas de la arquitectura.

- **Capa de Control de Componentes**
  Esta capa realiza un control directo sobre los componentes del sistema. Monitoriza su comportamiento, e informa de la situación actual a la capa inmediatamente superior (Capa de Gestión del Cambio).

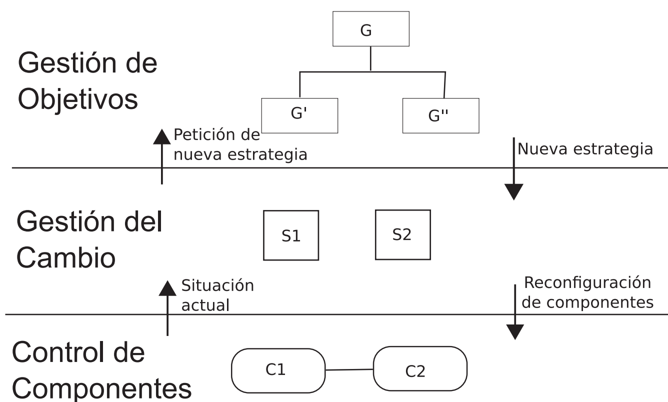


Figura 2.6: Esquema de la arquitectura de tres capas propuesta por Kramer y Merge

- **Capa de Gestión del Cambio**
  La capa intermedia tiene una conjunto de planes o estrategias para conseguir el objetivo de la aplicación. En base a la información sobre la situación actual que obtiene de la capa inferior (Capa de Gestión de Componentes), esta capa comprueba sus estrategias, y si es necesario realiza modificaciones para obtener una nueva configuración. Esto puede implicar introducir nuevos componentes o cambiar las interconexiones entre ellos. Cuando la situación actual no es soportada por ninguna de las estrategias que tiene esta capa, solicita una nueva estrategia a la capa superior (Capa de Control de Objetivos).

- **Capa de Control de Objetivos**
  Esta capa controla la consecución de los objetivos del sistema. Su tarea es crear nuevas estrategias para conseguir los objetivos del sistema teniendo en cuenta la situación actual del mismo. Esta capa recibe de la capa inferior (Capa de Gestión del Cambio) tanto la petición para realizar una estrategia como la información que necesita acerca de la situación actual.

## Capítulo 3

# Planteamiento del problema

En este este capítulo se presenta la arquitectura propuesta e implementada en este PFC, basada en [1], para la desarrollo de un sistema autoadaptativo en base a medidas de prestaciones en un entorno *open-world*.

En esta propuesta, se adapta la arquitectura de tres capas de Kramer y Magee (ver 2.8) al contexto del *open-world software*, en concreto a la evaluación de prestaciones. La arquitectura tendrá las mismas tres capas que la propuesta referida: Control de Componentes, Gestión del Cambio y Gestión de Objetivos. A continuación se definen las tareas y responsabilidades para cada etapa, adaptada a este supuesto de *open-world software* en base a prestaciones. El esquema obtenido se presenta en la figura 3.1.

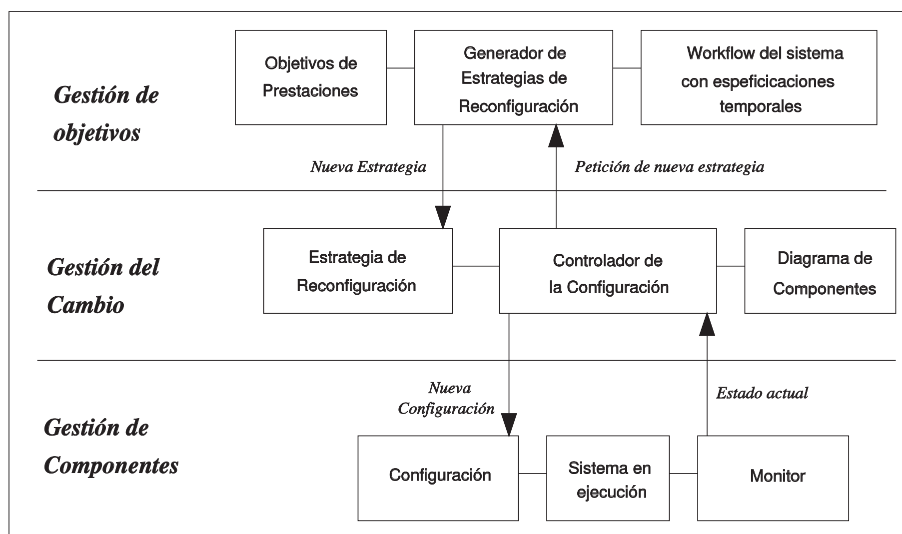


Figura 3.1: Arquitectura de tres capas aplicada al paradigma *open-world*

## 3.1. Capa de Control de Componentes

La capa de Control de Componentes es la de más bajo nivel, en contacto con el sistema y el entorno de ejecución. Por lo tanto, debe reaccionar con rapidez a los cambios que se puedan producir para adaptar el sistema de manera ágil. En nuestro contexto, esta capa es la responsable de gestionar los *bindings* y *unbindings* entre los diferentes componentes que propocionan ciertos servicios dentro del entorno de ejecución. Se identifican diferentes tareas que se deberán llevar a cabo en esta capa:

- Medir en cada momento las prestaciones de cada componente de la configuración actual.

- Descubrir nuevos componentes que proporcionen los servicios requeridos por el sistema.
- Comprobar si los componentes involucrados en la configuración actual siguen disponibles.

Se propone situar en esta capa un monitor (ver figura 3.1) que deberá llevar a cabo estas tres tareas. Para ello, necesitará conocer el workflow a ejecutar por el sistema, así como la configuración actual. Una configuración indica a qué proveedor requerir cada uno de los servicios solicitados en el *workflow* del sistema. Entendiendo por *workflow* la descripción de los pasos de ejecución a realizar por el sistema y será representado como un diagrama de Actividades conforme con el estándar UML [10].

Para realizar la primera de las tareas (medida de prestaciones de la configuración actual), el monitor medirá el tiempo transcurrido entre las llamadas a cada componente; mientras que para realizar las otras dos las tareas (descubrir nuevos componentes o detectar su falta de disponibilidad), puede emplear métodos ya conocidos y empleados en aplicaciones *open-world* (por ejemplo en el caso de los servicios web, la utilización del registro universal de descripción, descubrimiento e integración UDDI [17]).

Esta capa informará a la capa superior del estado actual del sistema cada vez que sea realizada una llamada a un componente, así como informará cuando un componente deje de estar disponible o cuando nuevos componentes sean encontrados.

## 3.2. Capa de Gestión del Cambio

La función de la capa de Gestión del Cambio es reaccionar frente a cambios que se detecten en la capa de Control de Componentes. Una característica clave de esta capa es que la reacción frente a los cambios debe realizarse en un tiempo despreciable. Esta capa deberá reaccionar de acuerdo con un conjunto de estrategias de reconfiguración en las que se indica cómo reaccionar a cada cambio. Estas estrategias no son creadas, sino consultadas, por lo que el tiempo de respuesta es mucho menor al de la creación de la estrategia.

Cada estrategia de autoconfiguración puede estar referida a un diferente criterio de interés: coste del servicio, prestaciones, etc. Además, esta capa debe detectar, si se da el caso, que el conjunto de estrategias de las que dispone dejen de ser adecuadas (porque se han producido cambios que las estrategias actuales no soportan, por ejemplo ya no se cumple el objetivo de prestaciones o los componentes seleccionados dejan de estar disponibles), en cuyo caso solicitará la creación de nuevas estrategias a la capa superior.

Aunque sería deseable disponer de varias estrategias de reconfiguración en base a diferentes criterios, simplificamos el problema refiriéndonos sólo a la medida de prestaciones. Proponemos un Controlador de la Configuración (ver figura 3.1) que realice las tareas nombradas para esta capa. Dicho Controlador de la Configuración necesitará la estrategia de reconfiguración a emplear obtenida de la capa superior, así como el estado actual obtenido de la capa inferior, y un diagrama de Componentes en el que esté descrito el comportamiento de cada componente.

## 3.3. Capa de Gestión de Objetivos

Por último, la misión de la capa de Gestión de Objetivos es que el sistema cumpla ciertas restricciones globales de prestaciones. Este objetivo será logrado a través de la creación de estrategias de reconfiguración.

Para ello, se propone un Generador de Estrategias, (ver fig. 3.1) que cree una nueva estrategia de reconfiguración cada vez que la capa de Gestión del Cambio la solicite. Para crear dicha estrategia, el Generador necesitará conocer el objetivo de prestaciones del sistema; el *workflow* de ejecución junto

a la especificación de ciertas propiedades de prestaciones; y la configuración actual de los componentes, proporcionada por la capa de Gestión del Cambio. La estrategia de reconfiguración obtenida para conseguir el objetivo de prestaciones, será comunicada a la capa inferior para reemplazar a la estrategia anterior.

Se considera que la implementación de la capa de Gestión de Objetivos es la más desafiante ya que es para la que menos mecanismos existen actualmente y sobre la que menos se ha investigado hasta ahora. Por ello, enfocamos nuestra atención en su definición e implementación. El capítulo 4 detalla la implementación del Generador de Estrategias, objeto de este PFC.

**Capítulo 4**

# Implementación del Generador de Estrategias

En este capítulo se explica el algoritmo Generador de Estrategias (GE) propuesto en 3.3, así como algunos aspectos destacables acerca de la implementación llevada a cabo en este PFC.

En primer lugar, se discute la obtención de las precondiciones del algoritmo. En segundo lugar, se define el formato del resultado a obtener por el GE: la estrategia de autoconfiguración. A continuación, se detalla el comportamiento del algoritmo GE. La sección 4.4.2 muestra cómo se ha llevado a cabo el proceso de evaluación de los flujos de ejecución mediante su conversión a redes de Petri. Por último, se presenta el paquete ejecutable obtenido.

## 4.1. Obtención de las precondiciones del algoritmo

El módulo Generador de Estrategias necesita cierta información del sistema:

- El *workflow* del sistema con anotaciones de prestaciones y el objetivo de prestaciones.
- Información acerca de los componentes que proporcionan los servicios requeridos por el sistema y sus especificaciones temporales, proporcionado por la capa de Gestión del Cambio.

En primer lugar, el ***workflow*** del sistema contiene información acerca de qué servicios son solicitados, cuántas veces y en qué orden; así como información acerca de la carga de trabajo del sistema. Éste es obtenido a partir de un diagrama de Actividad (AD) de UML (ver sección 2.1.1).

Sin embargo, los diagramas de Actividad no proporcionan mecanismos suficientes para representar algunos aspectos del dominio del problema, como son: la tasa de llegada de nuevas cargas de trabajo, o la información acerca de a qué componentes deben ser solicitados ciertos servicios. Para representar estos valores, se ha empleado el perfil de modelado de prestaciones MARTE, un estándar ISO desarrollado para ser utilizado junto a UML. El anexo D explica el subconjunto de anotaciones MARTE empleadas en este trabajo.

Se asume que el *workflow* necesita hacer llamadas a $K$ servicios diferentes, $s_k$, $k \in [1..K]$. Cada servicio podrá ser solicitado una o más veces en cada ejecución del *workflow*.

En segundo lugar, el algoritmo (Generador de Estrategias), necesitará conocer qué componentes ofrecen cada uno de los servicios solicitados por el *workflow*. Cada servicio puede ser proporcionado por uno o varios componentes. Esta información es obtenida a partir de un **diagrama de Componentes** (CD) de UML (ver sección 2.1.2). Estos diagramas representan los componentes del entorno de ejecución, las interfaces que ofrecen y requieren, y las relaciones entre ellos.

Dado un servicio $s_k$, éste puede ser ofrecido por varios componentes proveedores. Cada proveedor es denotado $C_{kl}$, siendo $k$ el número de servicio y $l$ el número de componente.

Además, el Generador de Estrategias necesitará una **tabla de tiempos** (TT) que contenga información sobre el comportamiento temporal de dichos componentes.

Para cada componente, podemos identificar distintos modos de funcionamiento, que denominamos fases de trabajo, con sus respectivos tiempos de respuesta medios. Por ejemplo, cuando los servidores están con una baja carga de trabajo, en una situación normal o ejecutando picos de trabajo.

Una fase de trabajo se caracteriza por dos cifras: el tiempo de servicio medio en la fase y el tiempo medio de duración de dicha fase. Esta información podrá provenir de la monitorización del sistema en las capas inferiores o bien de los propios proveedores. Se ha decidido representar esta información temporal en una tabla adjunta al CD para facilitar su gestión y estudio (ver, a modo de ejemplo, la tabla 4.1).

Finalmente, el algoritmo necesitará conocer el **objetivo de prestaciones** del sistema para intentar cumplirlo o informar acerca de su no cumplimiento.

A continuación, se explican las tareas llevadas a cabo para la obtención de los datos de entrada del algoritmo.

### 4.1.1. Elección de una herramienta de modelado UML

La segunda de las fases de desarrollo de este PFC consistió en la obtención de los diagramas UML. En primer lugar, hubo que elegir una herramienta de modelado UML apropiada. Los requisitos que debía cumplir la herramienta fueron:

- Modelado de diagramas bajo el estándar UML 2.0 .
- Posibilidad de añadir anotaciones MARTE a los diagramas UML.
- Posibilidad de exportar los diagramas en el formato estándar XMI[6], para facilitar su posterior importación como entrada del algoritmo. Muchas de las herramientas exportan los ficheros en XMI, pero tuvimos que comprobar que se trataba de ficheros XMI estándar.

La herramienta elegida para el modelado de los diagramas fue Papyrus [14]. El anexo A incluye un ejemplo de empleo de la herramienta.

### 4.1.2. Lectura de los ficheros XMI

Los ficheros modelados con Papyrus son exportados por dicha herramienta en el formato estándar XMI (ver sección 2.4). Antes de procesar dichos ficheros, se diseñaron los tipos de datos que almacenarían los diagramas de Actividad y de Componentes en el GE.

Existen dos aproximaciones para el procesado de XML (recordemos que XMI está basado en XML[12]): SAX y DOM (ver 2.3). Se ha elegido DOM ya que interpreta el fichero XML como un árbol, creando una estructura sobre la que se pueden realizar consultas y modificaciones. Para ello, carga todo los datos en memoria de modo que los accesos a elementos del fichero se realizan de manera eficiente.

### 4.1.3. Obtención de la información temporal de los componentes

El GE requiere de cierta información temporal que no puede ser proporcionada por un diagrama UML ni por sus anotaciones MARTE. Se trata de las fases de trabajo de cada uno de los componentes.

Esta información puede ser tabulada (TT). La tabla 4.1 presenta un ejemplo, en el que aparece información sobre cinco componentes proveedores de servicios. El componente $C_{11}$ (es decir, el proveedor número 1 del servicio $s_1$) tiene dos fases de trabajo, la primera de ellas tiene un tiempo medio de respuesta de 5 unidades de tiempo (u.t.) y permanece en ella 3000 u.t. en promedio y otra

fase con tiempo medio de respuesta 10 u.t. y permanece en ella 6000 u.t. Equivalentemente, $C_{21}$ tiene tres fases de trabajo, de 10, 70 y 250 u.t. de tiempo medio de respuesta y tiempos medios de permanencia de 6000, 2000 y 2000 u.t. respectivamente. Los componentes $C_{22}$ y $C_{31}$ siguen el comportamiento indicado en la tabla. Por último, el componente $C_{32}$ tiene una única fase, con un tiempo medio de respuesta de 30 u.t.

| **Fases de los componentes (u.t.)** | | | |
|---|---|---|---|
| | $fase_1$ | $fase_2$ | $fase_3$ |
| **C11** | (5,3000) | (20,6000) | |
| **C21** | (10,6000) | (70,2000) | (250,2000) |
| **C22** | (35,6000) | (140,4000) | |
| **C31** | (20,2000) | (70,2000) | |
| **C32** | (30,$\infty$) | | |

$fase_n$=(tiempo medio resp, tiempo medio en la fase)

Tabla 4.1: Comportamiento temporal de los proveedores *open-world* (TT)

La tabla se transcribe en un fichero XML que es procesado empleando DOM para servir de entrada al algoritmo. El formato de este fichero se detalla en el apéndice A. La información de la tabla está íntimamente relacionada con la información del CD, por lo que esta se añade al diagrama como información adicional relacionada con cada componente.

## 4.2. Estrategia de reconfiguración

El objetivo del Generador de Estrategias es obtener una “estrategia de reconfiguración” que pueda ser después interpretada y aplicada por la capa de Gestión del Cambio de la arquitectura (ver sección 3.2).

Para facilitar la comprensión de esta sección, se recuerda que un componente proporciona servicios en diferentes fases de trabajo (o modos de funcionamiento).

La estrategia de reconfiguración se representa como un grafo dirigido de nodos y arcos, como se explica a continuación. La figura 4.1 muestra un ejemplo.

**Nodos:** Un nodo representa una configuración del sistema; esto es, para cada servicio, el componente que está proporcionando el servicio y la fase de trabajo en la que se encuentra dicho componente. Una configuración también podría entenderse como el conjunto de componentes activos en un cierto instante y la fase de trabajo actual de cada uno de ellos.

**Arcos:** Un arco representa un cambio en la configuración del sistema. Gráficamente, un arco representa la unión entre dos nodos. Este cambio puede suponer el reemplazo de un componente por otro para proporcionar un cierto servicio o bien un cambio de fase de uno de los componentes. Los hay de dos clases:

- Arcos en línea continua: representan un cambio de configuración debido a que el sistema deja de comportarse como era esperado, es decir, se modifican sus prestaciones (seguramente como consecuencia de haberse degradado las prestaciones de alguno de los componentes proveedores). Estos arcos están caracterizados por dos valores: el servicio cuyo comportamiento se ha degradado y el nivel de confianza para realizar dicho cambio.

La capa de Gestión de Cambios, que interpreta la estrategia, decide si merece la pena cambiar del nodo origen al nodo destino a partir de los cambios observados en el comportamiento temporal de los componentes proveedores y el nivel de confianza indicado del arco. El cálculo del nivel de confianza necesario para tomar un arco se detalla en el algoritmo 3.

- Arcos en línea discontinua (arcos de retorno): representan un cambio de configuración debido al paso del tiempo. La utilidad de estos arcos es que, tras un cierto periodo de tiempo, el sistema pueda volver a una configuración anterior, la cual es posible que vuelva a funcionar con buenas prestaciones.

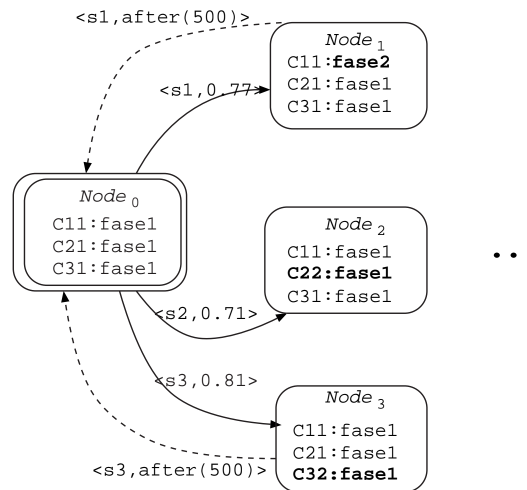


Figura 4.1: Ejemplo de estrategia de reconfiguración

La figura 4.1 muestra un ejemplo sencillo de una estrategia de reconfiguración: el nodo inicial, $Node_0$, representa la configuración del sistema en la cual los tres componentes activos son: $C_{11}$ en su fase 1, $C_{21}$ en su fase 1 y $C_{31}$ en su fase 1. El nodo $Node_1$ representa el cambio de fase del componente $C_{11}$: $C_{11}$ en su fase 2, $C_{21}$ en su fase 1 y $C_{31}$ en su fase 1. El arco de línea continua que une $Node_0$ y $Node_1$ representa el cambio de configuración debido a que el servicio $s_1$ (el cual provee $C_{11}$) ve degradadas sus prestaciones. Al interpretar la estrategia, este arco se tomará cuando se tenga un nivel de confianza menor que 0.77 de que el sistema se encuentra en la configuración descrita por $Node_0$. El nodo $Node_2$ representa un cambio de componente de $Node_0$ a $Node_2$. En este caso, el servicio afectado es $s_2$, y se produce un cambio de componente, de $C_{21}$ a $C_{22}$. Por último, los arcos en línea discontinua representan el retorno de $Node_1$ a $Node_0$ y de $Node_3$ a $Node_0$ tras 500 u.t. La estrategia de reconfiguración sería más grande, pero se muestran solo cuatro de sus nodos por simplicidad. El apéndice A muestra un ejemplo de una estrategia de reconfiguración completa.

Como resultado del algoritmo, la estrategia de reconfiguración es exportada en formato XML, empleando árboles DOM para la escritura del fichero.

## 4.3. Descripción de la generación de estrategias

Esta sección explica en alto nivel el comportamiento de los algoritmos que constituyen el Generador de Estrategias. Para una explicación más detallada y ejemplificada, ver apéndice A.

Se recuerda que el objetivo de este algoritmo es generar una estrategia para ser interpretada por la capa de Gestión del Cambio. Este algoritmo generará la estrategia cada vez que le sea solicitada

por dicha capa (ver sección 3.2). El algoritmo principal, algoritmo 1, recibe como entrada los datos descritos en la sección 4.1 y su objetivo es obtener la estrategia de reconfiguración explicada en la sección 4.2.

**Algorithm 1** Algoritmo principal

**Require:** Workflow del sistema ($AD$), objetivo de prestaciones ($objPrest$)
Componentes con su especificación temporal ($CD$,$TT$)
**Ensure:** Estrategia de reconfiguración (y un *warning* si no se consigue el objetivo de prestaciones)
1: inicializarEstrategia($E$)
2: $nodosPorProcesar \leftarrow \emptyset$
3: $N_i \leftarrow$ CrearNodoInicial($AD$, $CD$, $TT$)
4: $nodosPorProcesar \leftarrow N_i$
5: **while** $nodosPorProcesar \neq \emptyset$ **do**
6: $N_o \leftarrow$ ExtraerUnElemento($nodosPorProcesar$)
7: añadirALaEstrategia($N_o$)
8: **for all** $k \in [1..K]$ **do**
9: {Para cada servicio k}
10: $N_d \leftarrow$ CrearNodo($N_o$, $k$, $AD$, $CD$, $TT$)
11: **if** creado ($N_d$) **then**
12: $Arc \leftarrow$ CrearArco($N_o$, $N_d$, $k$)
13: añadirALaEstrategia($E$, $Arc$)
14: nodosPorProcesar $\leftarrow$ nodosPorProcesar $\cup$ $N_d$
15: **end if**
16: **end for**
17: **end while**
18: CrearNodosRetorno($E$)
19: **return** $\langle E$, comprobar($E$,$objPrest$)$\rangle$

La ejecución del mismo comienza con la inicialización de la estrategia de reconfiguración, $E$, que se define como un grafo de nodos y arcos y que comienza siendo un conjunto vacío. También se define un conjunto vacío de nodos por procesar. A continuación, el algoritmo crea el nodo inicial $N_i$ (como se explica más adelante), el cual es añadido al conjunto de nodos por procesar.

Mientras el conjunto de nodos por procesar no sea vacío, se extrae uno de los nodos ($N_o$), se añade $N_o$ a la estrategia y se realizan las siguientes operaciones para cada posible servicio $k$ del *workflow* de ejecución:
- Si el componente que está proporcionando el servicio $k$ en $N_o$ no está en su última fase (en cuyo caso su comportamiento no puede empeorar), entonces se crea $N_d$: el nodo adyacente a $N_o$ para el servicio $k$.
- Si $N_d$ ha sido creado, se calcula el arco desde $N_o$ hasta $N_d$, incluyendo dicho arco en la estrategia y añadiendo $N_d$ al conjunto de nodos por procesar.

Tras la ejecución de este bucle, todos los posibles nodos han sido procesados y creados, así como los arcos que los unen entre sí. Por último, habrá que crear los arcos de retorno de la estrategia (ver sección 4.2) a partir de unos criterios temporales.

La creación de un nuevo nodo se explica de manera abstracta en el algoritmo 2, siendo la creación del nodo inicial un caso particular de este algoritmo (el nodo inicial carece de nodo predecesor y de servicio a modificar).

La creación del nodo inicial consiste en la evaluación del *workflow* (AD) del sistema con todas las configuraciones posibles que éste pueda tener, seleccionando para cada uno de los componentes

**Algorithm 2** Creación de un nodo

**Require:** nodo predecesor ($N_0$), servicio que es modificado ($k$) {y la información de AD,CD,TT}
**Ensure:** Nodo ($mejorConf$)
1: $posiblesConf \leftarrow$ obtenerPosiblesConfiguraciones($N_0$,$k$)
2: $tiempoRespMin \leftarrow \infty$
3: **for all** $conf \in posiblesConf$ **do**
4: $rdp \leftarrow$ crearRdP(AD, $conf$)
5: $tiempoResp \leftarrow$ evaluarRdP($rdp$)
6: **if** $tiempoResp < tiempoRespMin$ **then**
7: $tiempoRespMin \leftarrow tiempoResp$
8: $mejorConf \leftarrow conf$
9: **end if**
10: **end for**
11: **return** $mejorConf$

la mejor de sus fases de trabajo. Tras la evaluación, se selecciona como configuración del nodo inicial aquella que proporciona un mejor tiempo de respuesta para el sistema.

Cuando la creación de un nodo $N_t$ se realiza a partir de otro nodo $N_0$ y un cierto servicio $k$, se realiza la evaluación de todas las posibles configuraciones, partiendo de la configuración del nodo $N_0$, y variando o bien el componente que proporciona el servicio $k$, o bien la fase del componente que proporcionaba el servicio $k$ en $N_0$. Como resultado, se selecciona como configuración del nuevo nodo aquella que mejor tiempo de respuesta proporcione.

El proceso de evaluación de los *workflows* se realiza a través de su traducción a redes de Petri y la evaluación de estas a través de la herramienta GreatSPN. La evaluación de los workflows se explica con mayor detalle en la sección 4.4.

El algoritmo de creación de un arco, algoritmo 3, recibe como parámetros de entrada los nodos origen ($N_o$) y destino ($N_d$) y el servicio ($k$) que es modificado entre uno y otro. Con esta información, el algoritmo calcula el nivel de confianza.

Nótese que debido a la característica *open-world* del sistema, las decisiones de paso de un nodo a otro (la capa de Gestión del Cambio decide los cambios de nodo al interpretar la estrategia) son tomadas en base a predicciones sobre el comportamiento de los componentes que proveen los servicios. Estas predicciones pueden ser acertadas o erróneas. Cuando se produce una predicción errónea se denomina "falso positivo".

El cálculo del nivel de confianza para pasar de un nodo a otro se calcula relacionando la ganancia de prestaciones ante una predicción acertada y la pérdida de prestaciones ante un falso positivo. Este nivel de confianza se calcula a partir de dos valores: la mejora de prestaciones al realizar una reconfiguración debido a una predicción correcta (líneas 6-11 del algoritmo 3), la mejora entre la configuración de $N_o$ con las prestaciones degradadas y $N_d$; y la pérdida de prestaciones al realizar una reconfiguración por una falso positivo, la pérdida por emplear la configuración de $N_t$ en lugar de la de $N_o$, que sigue con su comportamiento sin degradar (línea 11 del algoritmo 3):

$$nivelConf = \frac{mejoraPrest}{mejoraPrest + perdidaPrest}$$

La obtención del nodo empeorado $N_{emp}$ (línea 6 del algoritmo) de $N_0$ respecto a $k$ se realiza tomando los mismos componentes proveedores que $N_0$, de los cuales todos mantienen sus fases de trabajo excepto aquél que provee el servicio $k$, que pasa a su siguiente fase.

Cuando los nodos origen y destino de un arco tienen los mismos componentes activos (pero con uno de ellos en una fase diferente), el nivel de confianza se calcula relacionando los tiempos de

**Algorithm 3** Creación de un arco

**Require:** nodo origen ($N_o$), nodo destino ($N_d$), servicio que es modificado ($k$) {y la información de CD,TT}
**Ensure:** Arco de $N_o$ a $N_d$
{Calcular tiempo de respuesta $node_o$}
1: $rdp_o \leftarrow$ crearRdP($workflow$, $node_o$)
2: $tiempoResp_o \leftarrow$ evaluarRdP($rdp_o$)
{Calcular tiempo de respuesta $node_d$}
3: $rdp_d \leftarrow$ crearRdP($workflow$, $node_d$)
4: $tiempoResp_d \leftarrow$ evaluarRdP($rdp_d$)
5: **if** componentesActivos($N_o$) $\neq$ componentesActivos($N_d$) **then**
6: $node_{emp} \leftarrow$ obtenerNodoEmp($node_{emp}$,$k$)
{Calcular tiempo de respuesta $node_o$ con condiciones empeoradas ($node_{emp}$) para $k$}
7: $rdp_{emp} \leftarrow$ crearRdP($workflow$, $node_{emp}$)
8: $tiempoResp_{emp} \leftarrow$ evaluarRdP($rdp_{emp}$)
9: $mejoraPrest = tiempoResp_{emp} - tiempoResp_d$
10: $perdidaPrest = tiempoResp_d - tiempoResp_o$
11: $nivelConf = \frac{mejoraPrest}{mejoraPrest+perdidaPrest}$
12: **else**
13: $nivelConf = \frac{tiempoResp_o}{tiempoResp_d}$
14: **end if**
15: **return** $\langle k, nivelConf\rangle$

respuesta del nodo origen y el nodo destino (línea 13 del algoritmo 3):

$$nivelConf = \frac{tiempoResp_o}{tiempoResp_d}$$

## 4.4. Evaluación de los *workflow*

El Generador de Estrategias requiere la evaluación del *workflow* (representado por un diagrama de Actividad) para diferentes configuraciones del sistema. Esta evaluación se realiza a través de una red de Petri que es traducción del diagrama de Actividad que lo representa. El comportamiento temporal de esta red de Petri se evalua mediante la herramienta GreatSPN [7].

### 4.4.1. Traducción de los flujos de ejecución a redes de Petri

La traducción de diagramas de Actividad a redes de Petri fue propuesta de manera teórica en [18, 19] y ha sido implementada en algunos PFCs [20, 21, 22] desarrollados con anterioridad. Se trata de un potente mecanismo de traducción que convierte del metamodelo de diagramas de Actividad de UML al metamodelo de las redes de Petri. Por ello, cualquier diagrama conforme con el metamodelo podrá ser traducido a su correspondiente red de Petri.

El método empleado ha consistido en la traducción de cada elemento del diagrama de Actividad con anotaciones en MARTE en una red de Petri estocástica, generalizada y etiquetada (LGSPN) equivalente. Los elementos (lugares o transiciones) iniciales y finales de cada LGSPN parcial son etiquetados con el nombre del elemento que ha sido traducido. Una vez traducidos todos los elementos, y partiendo de su etiquetado, se realiza la composición de todas las LGSPN parciales en una sola. . Los elementos del diagrama de Actividad y las anotaciones del perfil MARTE traducidos a LGSPN se nombran a continuación. La traducción concreta de cada elemento se presenta en el Apéndice E.

1. **Acciones:** Engloban las actividades (o acciones) y los nodos inicial y final. En la traducción, los lugares y transiciones inicial y final son etiquetados con el nombre de la acción.

2. **Transiciones de los diagramas de Actividad:** Representan la unión entre dos o más acciones en forma de secuencia, selección, fin de selección, bifurcación y fin de bifurcación. Para posibilitar su correcta composición posterior, los lugares y transiciones son etiquetados con el nombre de las acciones predecesoras o sucesoras según corresponda.

3. **Adquisición y liberación de recursos:** La adquisición y liberación de recursos, así como el recurso en sí, son obtenidos a partir de las anotaciones MARTE del diagrama de Actividad y también son traducidas con elementos en la red de Petri resultante.

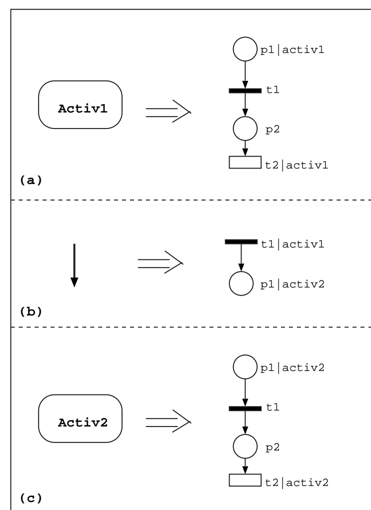


Figura 4.2: Ejemplo de traducción de elementos de un diagrama de Actividades a LGSPN

En la figura 4.2 se ejemplifica la traducción de tres elementos de un diagrama de actividad: dos actividades y la transición secuencial que las une.

La **composición** de redes de Petri se propone de manera teórica en [23]. La idea intuitiva bajo la composición de una red $rdp1$ con una red $rdp2$ se explica a continuación. Partiendo de la red de Petri $rdp1$, para cada lugar (o transición) de $rdp2$ se comprueba si existe un lugar (o transición) en $rdp1$ con la misma etiqueta. Si es así, ambos lugares (o transiciones) se identifican como uno solo. Si no lo es, entonces se añade dicho lugar (o transición) de $rdp2$ como un lugar (o transición) nuevo.

La figura 4.3 ejemplifica cómo se realiza la composición de tres LGSPN en una sola, a partir de la composicion de dos de las redes y el resultado de estas con la tercera.

### 4.4.2. Evaluación de las redes de Petri

Una vez obtenidas las redes de Petri a partir de los diagramas de Actividad, se evalúa su comportamiento temporal con ayuda de la herramienta GreatSPN. Nótese que el tipo de redes de Petri a evaluar en este trabajo han sido redes de carga abierta.

La evaluación de prestaciones de una red de Petri se puede realizar mediante dos técnicas: análisis y simulación.

El **análisis** consiste en resolver la red de manera teórica a través del análisis de su espacio de estados. Las herramientas conocidas no implementan el análisis de redes de Petri de carga abierta como las que aquí se abordan.

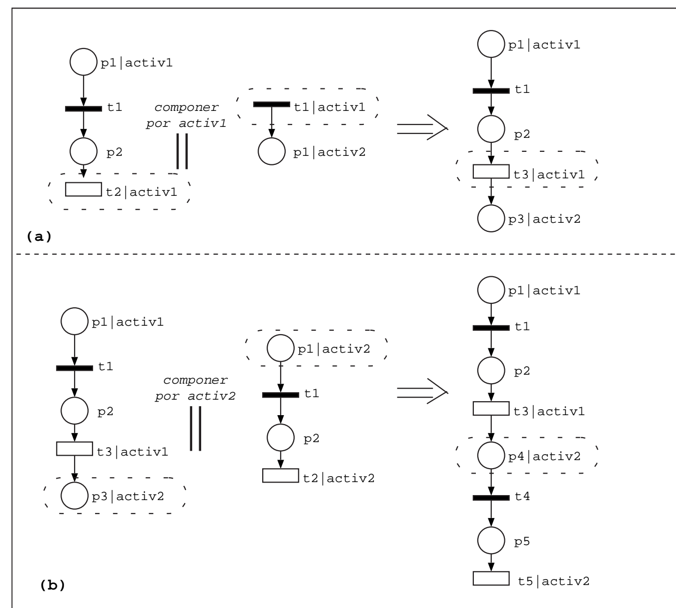


Figura 4.3: Ejemplo de composición de LGSPN. Se remarcan los elementos compuestos en uno solo

La **simulación**, por su parte, consiste en ejecutar el comportamiento de la red hasta que se satisfacen ciertos umbrales de precisión y obtener los valores medios resultantes. La técnica de simulación sí está implementada para el estudio y análisis de redes de carga abierta. En este PFC se emplea la simulación debido a que es la técnica implementada por GreatSPN para las redes de Petri de este trabajo.

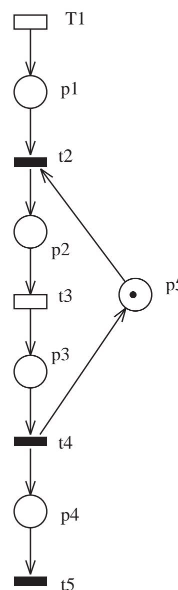


Figura 4.4: Red de Petri a evaluar

La simulación de las redes de Petri obtenidas a partir del diagrama de Actividades se realizó con

llamadas a las funciones de GreatSPN. Para ello, hubo que realizar las siguientes operaciones:

- Exportar las redes de Petri a un fichero acorde con el formato de la herramienta GreatSPN.
- Realizar la llamada a la función de simulación de GreatSPN (*WNSIM*) desde el programa implementado en JAVA, a través de una llamada a un comando de sistema.
- Leer el fichero resultante de la ejecución de la función de GreatSPN para obtener los resultados de la simulación.
- Calcular el tiempo de respuesta medio a partir de los datos de la simulación aplicando la Ley de Little, como se explica a continuación. Este tiempo de respuesta medio es el empleado como resultado de la "evaluación" de la red en el algoritmo Generador de Estrategias.

La ley de Little [24] relaciona los valores medios de tres variables de importancia en un sistema, y se enuncia como: El número medio de usuarios en el sistema ($N$) es igual a la tasa media de llegada ($\lambda$) multiplicado por el tiempo promedio de un cliente en el sistema o tiempo medio de respuesta del sistema ($T$).

$$N = \lambda * T$$

Empleando la ley de Little, se puede calcular el tiempo medio de servicio $T$. A partir del ejemplo de la figura 4.4, se define el número medio de usuarios en el sistema como el número medio de *tokens* en el lugar $p1$, ya que es el número de peticiones que han tenido que esperar en el lugar p1 para ser ejecutadas; sumado al número medio de peticiones en ejecución. Esto es 1 - (ocupación media de $p5$). Por último, la tasa media de llegada equivale al rendimiento (*throughput*) de la transición $T1$. En la fórmula enunciada a continuación, se denota $\#p_i$ el número medio de *tokens* en el lugar $P_i$ y $thrg(T_j)$ el *throughput* medio de la transición $T_j$.

$$T = \frac{\#p1 + (1 - \#p5)}{thrg(T1)}$$

## Capítulo 5

# Planificación temporal.

Este capítulo proporciona una revisión acerca de la planificación temporal del proyecto, así como el tiempo invertido en cada una de las fases de desarrollo en las que éste fue dividido. En el apéndice C se explican los hitos de cada fase de desarrollo, así como los problemas encontrados.

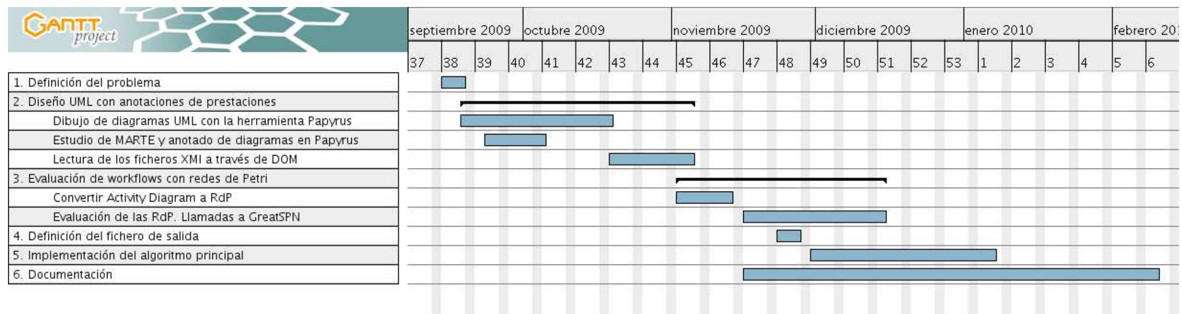


Figura 5.1: Diagrama de Gantt del desarrollo del proyecto

El diagrama de Gantt de la figura 5.1 muestra cómo ha evolucionado el proyecto a lo largo del tiempo. El diagrama de la figura 5.2 distribuye de las horas empleadas en las diferentes fases de desarrollo.

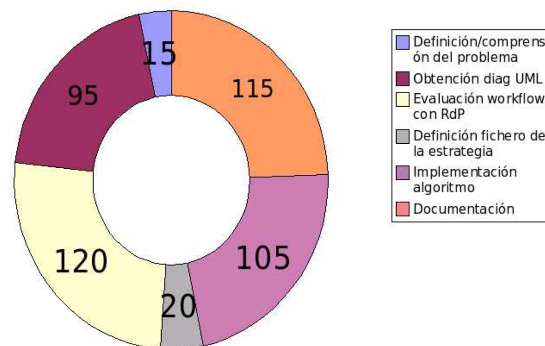


Figura 5.2: Distribución del esfuerzo en las fases de desarrollo

El proyecto ha sido realizado entre septiembre de 2009 y febrero de 2010. El trabajo ha sido constante durante este periodo, con dedicación completa. Esto hace un total de seis meses de trabajo y un total de 470 horas empleadas.

## Capítulo 6

# Conclusiones y trabajo futuro

Este capítulo presenta algunas conclusiones obtenidas de la elaboración de este PFC. Además, plantea el trabajo futuro.

## 6.1. Conclusiones y resultados obtenidos.

Una vez terminado todo el proceso que me ha llevado a redactar estas líneas, desde la tarea de elegir Proyecto Fin de Carrera hasta la redacción de esta misma memoria, puedo ahora obtener ciertas conclusiones, tanto técnicas como personales.

Respecto al trabajo desarrollado, se ha cumplido el objetivo propuesto. Se ha obtenido como resultado la implementación de un paquete ejecutable que se encarga de generar estrategias de reconfiguración. Este paquete ejecutable está listo para ser integrado en sistemas autoadaptativos que siguen la arquitectura de tres capas explicada. Con la integración de este paquete en el sistema, se conseguirá, dado el *workflow* de la aplicación *software*, que el sistema sea capaz de autoconfigurarse respondiendo a estímulos recibidos en un entorno *open-world*.

Este *software* desarrollado corresponde a la capa más desafiante y todavía desconocida de la arquitectura. Por lo tanto, mediante la realización de este paquete ejecutable se ha dado un paso adelante hacia la implementación real de estos tipos de sistemas autoadaptativos en base a sus prestaciones. Por ello, se considera que este trabajo tiene cierta relevancia en el ámbito de la investigación de nuevos métodos y técnicas para el *open-world software* y sistemas autoadaptativos en tiempo de ejecución.

En cuanto a lo personal, estoy satisfecha con el trabajo realizado. El tamaño del proyecto y el hecho de dedicarme a tiempo completo a él (sin horarios ni calendarios previamente fijados) han causado que haya aprendido a organizar mi tiempo, así como a realizar una planificación inicial y revisarla periódicamente.

Por otra parte, he ampliado mis conocimientos sobre redes de Petri (PN, GSPN, LGSPN...) y sobre evaluación de prestaciones; así como otras tecnologías como el procesado de ficheros XML o la aplicación de perfiles (como el estándar MARTE) a los diagramas UML.

Finalmente, este trabajo me ha acercado al ámbito de la investigación científica, debido a que se trata de implementar algo recientemente investigado y a que he tenido que leer y comprender varios artículos. Además, he podido entender algunos desafíos que plantea y planteará en los próximos años el paradigma del *open-world software*, el cuál todavía está en etapas tempranas de investigación.

## 6.2. Trabajo futuro

En esta sección se exponen algunos trabajos futuros que completarían o ampliarían el trabajo desarrollado en este PFC:

- El generador de estrategias considera los principales elementos de los diagramas de Actividad (selección, bifurcación, unión, etc), pero no todos. Podría ampliarse el sistema para que tratase otros elementos como el envío y recepción de señales, los cuales pueden provocar interrupciones en el flujo de ejecución.

- Implementar de manera más precisa la creación de los arcos de retorno. En la actualidad, se emplea como tiempo de retorno el tiempo medio de llegada de nuevas cargas de trabajo. Dedicándole más tiempo de investigación se podría realizar un cálculo más sofisticado que mejorase la precisión de la predicción para estos arcos.

- En el entorno open-world, los retrasos obtenidos al solicitar un servicio son la suma del retraso del servicio en sí y los retrasos causados por la red de comunicación (satélite, WLAN, LAN, etc). En este proyecto no se realiza distinción entre cuál es la causa del retraso, sin embargo sí podría realizarse. Para ello podría añadirse al sistema un diagrama de Distribución de UML y una tabla asociada en la que almacenar los retrasos causados por la red. Este es un tema que necesita todavía investigación. El objetivo sería conseguir cierto grado de confianza al predecir si un retraso percibido por el cliente se refiere a (1) una respuesta lenta del servidor o (2) a un retraso causado por la red de comunicación entre componentes.

- La estrategia de reconfiguración considerada en el sistema es obtenida en base a las medidas de prestaciones de los proveedores de servicios. Sin embargo, podrían calcularse y emplearse otras estrategias de reconfiguración basadas en otros criterios, como el coste del servicio o la fiabilidad del servicio obtenido o incluso llegando a compromisos entre ellas.

- PNML[25] es un estándar para el intercambio de redes de Petri basado en XML de creciente popularidad. En este PFC los diagrama de Actividad se traducen directamente a redes de Petri. El diagrama de Actividad sería convertido a una red de Petri representada con el estándar PNML, y dicha red de Petri en formato PNML sería convertida a una red de Petri en formato de entrada de GreatSPN. Esto podría mejorar la potencia de comunicación y colaboración con otros paquetes o módulos. Por ejemplo en caso de (1) utilizar las RdP creadas por otros paquetes o (2) facilitar la el uso de las redes creadas por otros paquetes que las necesiten.

- Implementación de las capas inferiores (capa de control de componentes y capa de gestión del cambio) de la arquitectura propuesta en el capítulo 3. Estas capas completarían el sistema, de manera que no solo se crearían las estrategias de reconfiguración sino que el sistema sería capaz de emplearlas y autoconfigurarse. Abordar esta tarea de implementación exige un trabajo previo de investigación sobre las particularidades de dichas capas.

# Bibliografía


[1] D. Pérez-Palacín, J. Merseguer, and S. Bernardi. Performance aware open-world software in a 3-layer architecture. In *Proceedings of the 1st International Conference on Performance Engineering (ICPE'10)*, pages 49–56.

[2] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

[3] Object Management Group, `http://www.promarte.org`. *A UML Profile for MARTE.*, 2009.

[4] M. Silva. *Las redes de Petri en la Automática y la Informática*. AC, Madrid (Spain), 1985.

[5] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[6] XML Metadata Interchange (XMI).
`http://www.omg.org/technology/documents/formal/xmi.htm`.

[7] La herramienta GreatSPN. `http://www.di.unito.it/~greatspn`.

[8] D. Pérez-Palacín and J. Merseguer. Performance evaluation of self-reconfigurable service-oriented software with stochastic petri nets. In *Proceedings of the Fourth International Workshop on Practical Applications of Stochastic Modelling (PASM'09), Electronic Notes in Theoretical Computer Science (2010).*

[9] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.

[10] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language*. Addison Wesley, 1999.

[11] Object Management Group. Unified Modeling Language: Superstructure, July 2005. Version 2.0, formal/05-07-04.

[12] The Extensible Markup Language (XML). `http://www.w3.org/XML`.

[13] Java technology. `http://www.sun.com/java/`.

[14] La herramienta de modelado Papyrus UML. `http://www.papyrusuml.org/`.

[15] J. Kramer and J. Magee. A rigorous architectural approach to adaptive software engineering. *Journal of Computer Science and Technology*, 24(2):183–188, 2009.

[16] R. Murphy E. Gat, R. P. Bonnasso and A. Press. On three-layer architectures. *Artificial Intelligence and Mobile Robots*, pages 195–200, 1997.

[17] Universal Description, Discovery and Integration (UDDI). `http://uddi.xml.org/`.

[18] J. Merseguer. *Software Performance Engineering based on UML and Petri nets*. PhD thesis, University of Zaragoza, Spain, March 2003.

[19] J.P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic Petri nets: Application to software performance engineering. In *Proceedings of the Fourth International Workshop on Software and Performance (WOSP'04)*, pages 25–36, Redwood City, California, USA, January 2004. ACM.

[20] J. P. López Grao. Evaluación del rendimiento de software (mediante UML + GSPN): desarrollo de una case. Proyecto Final de Carrera. Universidad de Zaragoza, Enero 2002.

[21] I. Trigo Conde. Análisis automático de prestaciones de sistemas a partir de modelos en UML, mediante traducción a Redes de Petri generalizadas. Proyecto Final de Carrera. Universidad de Zaragoza, Septiembre 2004.

[22] B. Fernandez Bacarizo. Evaluación del rendimiento del software: traducción de UML (máquinas de estados + diagramas de actividades) a GSPN. Proyecto Final de Carrera. Universidad de Zaragoza.

[23] S. Bernardi, S. Donatelli, and A. Horváth. Implementing compositionality for stochastic Petri nets. *STTT*, 3(4):417–430, 2001.

[24] J. D. C. Little. A proof of the queueing formula $l = \lambda w$. *Operations Research*, 9:383–387, 1961.

[25] J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri net markup language: Concepts, technology, and tools. In *ICATPN*, volume 2679 of *LNCS*, pages 483–505, 2003.

[26] L. Arracó Checa. Desarrollo en ArgoSPE de una API para redes de Petri. Traduccion a TimeNET. Proyecto Final de Carrera. Universidad de Zaragoza, Agosto 2007.

[27] Document Object Model (DOM). `http://www.w3.org/DOM/`.

[28] Object Management Group. `http://www.omg.org`.

[29] Simple API for XML (SAX). `http://www.saxproject.org/`.

[30] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`.

[31] The Extensible Stylesheet Language Family (XSL). `http://www.w3.org/Style/XSL`.

# Parte II

# Apéndices

## Apéndice A

# Caso práctico

Este anexo ejemplifica el proceso de creación de una estrategia de reconfiguración con un caso práctico. En primer lugar, se explica la obtención de los datos de entrada precisados por el Generador de Estrategias. A continuación, se explican algunos aspectos de la ejecución del algoritmo. Finalmente, se explica la estrategia obtenida como resultado.

## A.1. Descripción del problema a resolver

En primer lugar, se propone un sistema ejemplo sobre el cual ilustrar todo el proceso de obtención de la estrategia de reconfiguración.

Proponemos un sistema que ejecuta tres operaciones secuenciales, las cuales realizan llamadas a tres servicios $S1$, $S2$ y $S3$ de su entorno, considerado un entorno abierto (*open-world*). Suponemos la carga de trabajo del sistema abierta, la cual tiene una media de una petición cada 500 unidades de tiempo (u.t.) y sigue una distribución exponencial. Este comportamiento es modelado con el diagrama de Actividad anotado con el perfil MARTE de la figura A.1.

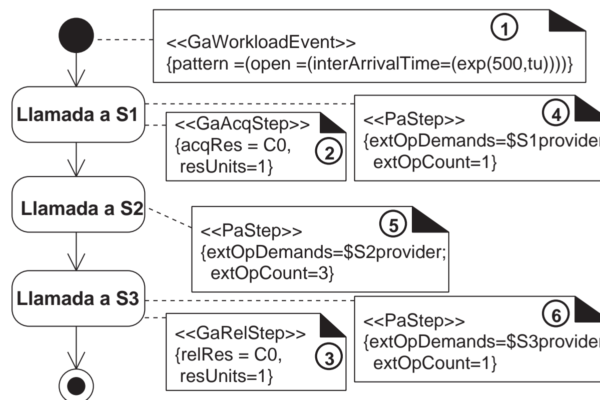


Figura A.1: Diagrama de Actividad con anotaciones MARTE

A continuación se explican cada una de las anotaciones MARTE empleadas:

① Representa la carga de trabajo del sistema, que sigue el patrón explicado en el párrafo anterior.

② Representa la adquisición de 1 unidad (atributo `resUnits`) del recurso C0 (atributo `acqRes`). En este ejemplo, consideramos que el recurso C0 es el sistema en sí, por lo que el sistema es "reservado" o "adquirido" al ejecutar la primera actividad (Llamada a S1).

③ Representa la liberación de 1 unidad (`resUnits`) del recurso C0 `relRes`). La liberación del sistema en sí (C0) se realiza con la última operación.

④,⑤,⑥ Anotación para representar el número de ejecuciones de cada acividad (atributo `extOpCount`) y el identificador del servicio externo demandado por cada actividad (`extOpDemands`).

El diagrama de Componentes de la figura A.2 representa los proveedores disponibles para cada uno de los tres servicios requeridos por el sistema. El servicio $S1$ es ofrecido por un único proveedor, $C11$; mientras que los servicios $S2$ y $S3$ son ofrecidos por dos proveedores cada uno: $C21$, $C22$, y $C31$, $C32$ respectivamente.

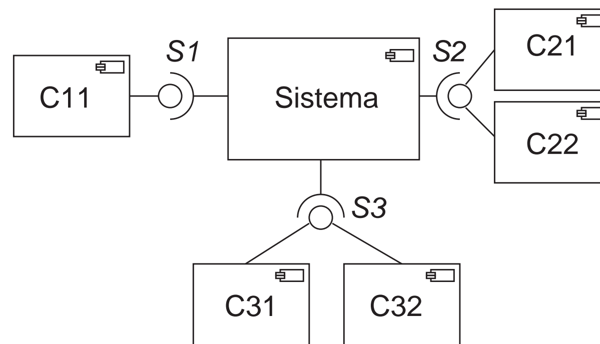


Figura A.2: Diagrama de Componentes

Finalmente, la tabla A.1 proporciona información acerca de las fases de cada uno de los componentes. Donde cada fase es una tupla de dos valores: el tiempo medio de servicio y el tiempo medio de permanencencia en dicha fase.

| **Fases de los componentes (u.t.)** | | | |
|---|---|---|---|
| | $fase_1$ | $fase_2$ | $fase_3$ |
| **C11** | (5,3000) | (20,6000) | |
| **C21** | (10,6000) | (70, 2000) | (250,2000) |
| **C22** | (35,6000) | (140,4000) | |
| **C31** | (20,2000) | (70,2000) | |
| **C32** | (30,$\infty$) | | |

Tabla A.1: Comportamiento temporal de los proveedores *open-world*

## A.2. Obtención de los datos de entrada

Esta sección explica el proceso de obtención de los datos de entrada del algoritmo: los diagramas de Actividad y de Componentes y la tabla sobre el comportamiento temporal de los componentes.

### A.2.1. Obtención del diagrama de Actividades anotado con MARTE

#### Obtención del diagrama con la herramienta Papyrus

La obtención del diagrama de Actividades se realiza introduciendo los datos del diagrama de Actividades en la herramienta Papyrus.

En la herramienta, el diagrama de Actividad es un diagrama que modela el comportamiento de un componente o clase. Por ello, en primer lugar se crea un componente que representa el Sistema, y a continuación se añade un diagrama de actividad a dicho componente. Los elementos del diagrama (acciones, nodo inicial, transiciones, etc.) son añadidos dentro de una Actividad.

Tras ello, hay que importar y aplicar el perfil MARTE para que los elementos puedan ser anotados con dicho perfil.

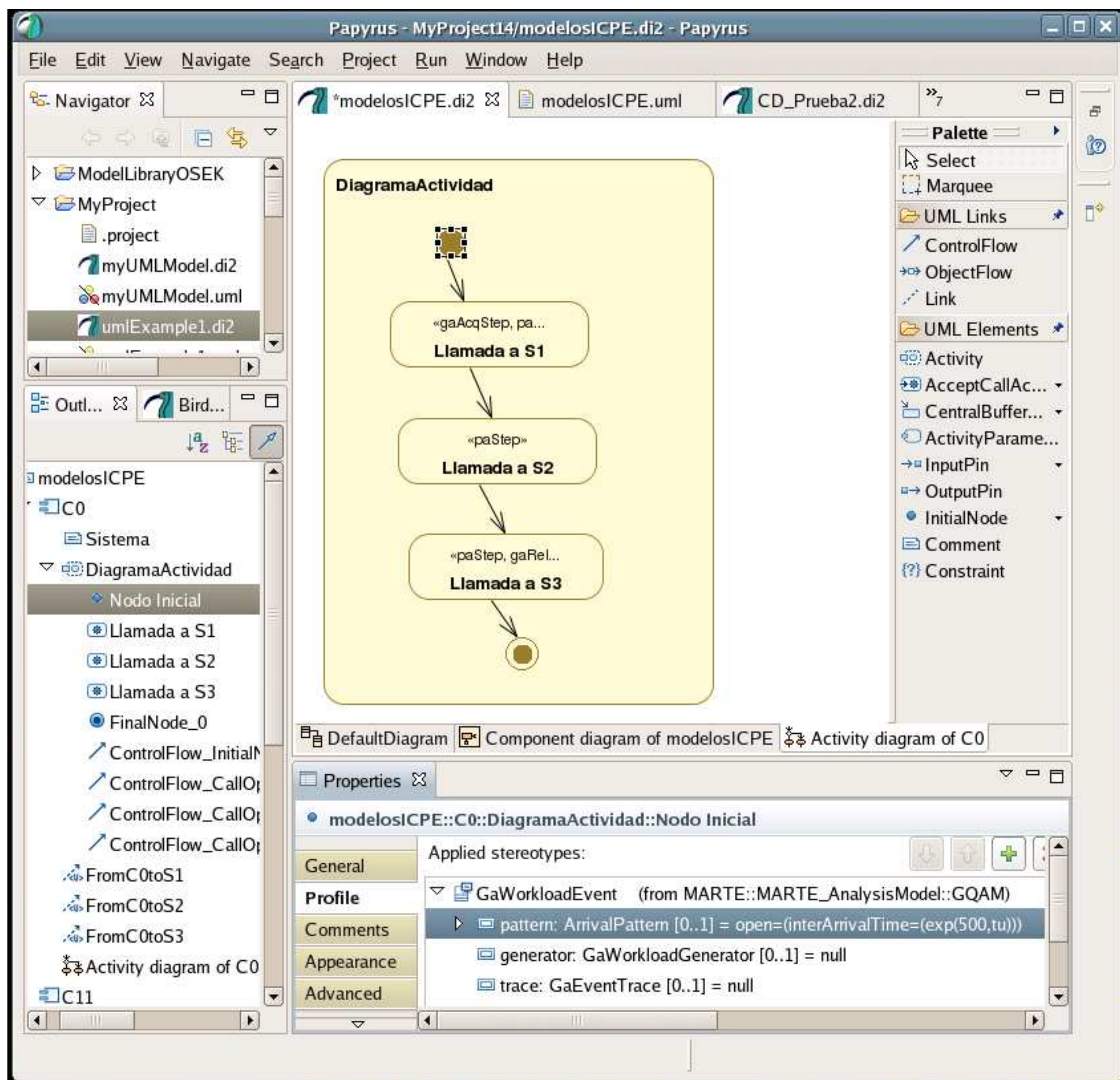


Figura A.3: Diagrama de Actividad modelado en la herramienta Papyrus con anotaciones MARTE

La figura A.3 muestra cómo se ha modelado el diagrama de Actividad con la herramienta Papyrus. Las anotaciones MARTE son incluidas como un “Profile” dentro de la pestaña de “Properties” para cada elemento del diagrama.

**Parseo del fichero XMI**

La herramienta exporta el diagrama de acuerdo con el estándar XMI. En el fichero XMI, el diagrama de Actividades está representado como un elemento dentro del elemento que representa al sistema (el componente Sistema del diagrama de componentes). Cada elemento del diagrama de Actividades es representado en el fichero XMI como un elemento dentro del elemento diagrama de Actividad. Por ejemplo, el nodo inicial de nuestro diagrama de actividades es representado como:

```
<node xmi:type="uml:InitialNode" xmi:id="_nOvYgOiqEd6GKesHY96ouQ"
name="Nodo Inicial" outgoing="_L9Z40OirEd6GKesHY96ouQ"/>
```

El atributo `xmi:type` representa el tipo de elemento UML del que se trata, `xmi:id` es un identificador único que el programa asigna a este elemento, `name` es el nombre visible y modificable por el usuario y `outgoing` es el identificador del siguiente elemento del diagrama, en este caso, el identificador de la transición secuencial que sucede al nodo inicial.

### A.2.2. Obtención del diagrama de Componentes

**Obtención del diagrama con la herramienta Papyrus**

La obtención de un diagrama de Componentes en Papyrus requiere en primer lugar realizar un diagrama de clases en el que se definan los componentes, sus interfaces, y las relaciones de "uso" o "realización" que existen entre ellos (ver figura A.4).

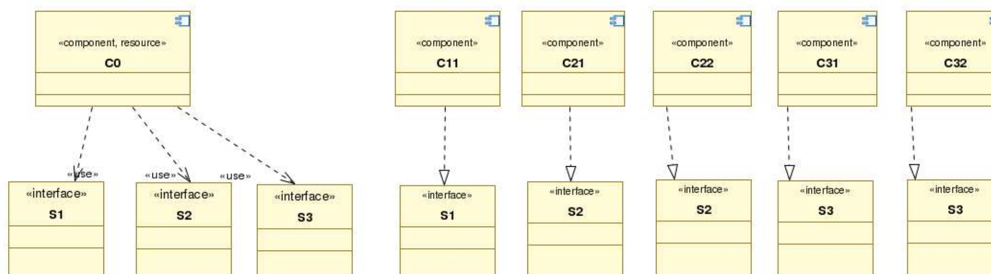


Figura A.4: Componentes e interfaces del diagrama de Componentes

En segundo lugar, se realiza el diagrama de Componentes. La herramienta muestra las interfaces ofrecidas ("realización") o requeridas ("uso") de cada componente. Las interfaces usadas y sus correspondientes realizaciones son unidas gráficamente en el diagrama de actividad a través de una unión de "dependencia". El diagrama de componentes del sistema propuesto como ejemplo es el mostrado en la figura A.5

**Procesado del fichero XMI**

Al igual que en el caso del diagrama de Actividades, el fichero XMI es obtenido directamente a partir de la herramienta Papyrus. Es posible la obtención de ambos diagramas a partir de un único fichero XMI.

El fichero relaciona los componentes, las interfaces, las realizaciones y los usos. La línea de ejemplo presentada a continuación representa el componente "Sistema":

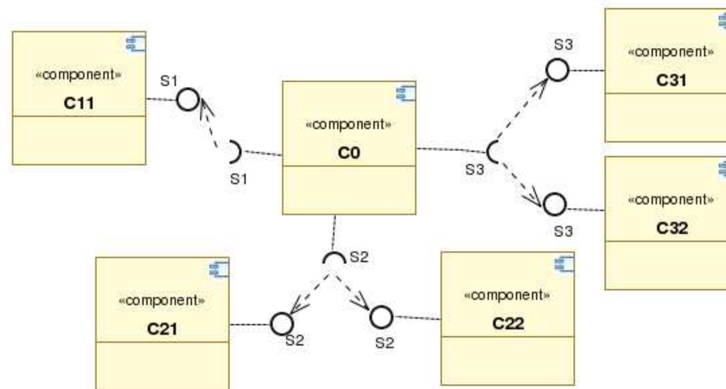


Figura A.5: Diagrama de Componentes modelado en la herramienta Papyrus

```
<packagedElement xmi:type="uml:Component"
xmi:id="_iUmesMInEd6Az8L0rwC1FA" name="C0"
clientDependency="_MC0CMMIoEd6Az8L0rwC1FA _MPtOsMIoEd6Az8L0rwC1FA
_MdDHIMIoEd6Az8L0rwC1FA"/>
```

Los atributos `xmi:type`, `xmi:id` y `name` tienen el mismo significado que el explicado para los diagramas de Actividades. El atributo `clientDependency` engloba los idenficadores de todas las relaciones de “realización” o “uso” en los que está implicado este compontente. En este caso, tres relaciones de “uso”, correspondientes a $S1$, $S2$ y $S3$.

A continuación se presenta una de las relaciones de “uso” de este componente, en la cual se puede comprobar que están relacionados a través de sus identificadores:

```
<packagedElement xmi:type="uml:Usage" xmi:id="_MC0CMMIoEd6Az8L0rwC1FA"
name="FromC0toS1" supplier="_4ZkKsMInEd6Az8L0rwC1FA"
client="_iUmesMInEd6Az8L0rwC1FA"/>
```

**Obtención de la tabla de tiempos asociada a los proveedores**

Con el objetivo de parsear la información temporal acerca de los componentes proveedores planteada en la tabla A.1, ésta es escrita en un fichero XML con la estructura siguiente:

```
<timeTable>
   <provider name='C11'>
      <phase serviceTime='5'  sojournTime='3000'/>
      <phase serviceTime='20' sojournTime='6000'/>
   </provider>
   (...)
   <provider name='C31'>
      <phase serviceTime='20' sojournTime='2000'/>
      <phase serviceTime='70' sojournTime='2000'/>
   </provider>
   <provider name='C32'>
      <phase serviceTime='30' sojournTime='1000'/>
   </provider>
```

```
</timeTable>
```

El elemento raíz del fichero XML, `<timeTable>` representa la tabla temporal en su conjunto. Cada proveedor (`provider`) está identificado por su nombre, y puede tener una o más fases de funcionamiento. `serviceTime` representa el tiempo medio de respuesta del proveedor, y `sojournTime` representa el tiempo medio en el que el proveedor está en dicha fase.

Como se ha explicado, este fichero es parseado mediante DOM, y la información proporcionada por cada de las fases es almacenada en el diagrama de Componentes junto al correspondiente componente proveedor de un servicio.

### A.2.3. Almacenamiento de los datos durante la implementación

Una vez obtenidos los datos, son almacenados en objetos de JAVA para servir de parámetro de entrada para el Generador de Estrategias implementado en JAVA. El diagrama de Clases de la figura A.6 muestra el metamodelo de los diagramas de Actividades empleados en este proyecto. El diagrama de Componentes está estructurado según el diagrama de Clases de la figura A.7; el cual engloba la información obtenida del diagrama de Componentes a través del fichero XMI y la información sobre las fases de cada componente obtenidas a partir de la tabla. El anexo B muestra el diagrama de clases completo del Generador de Estrategias desarrollado.

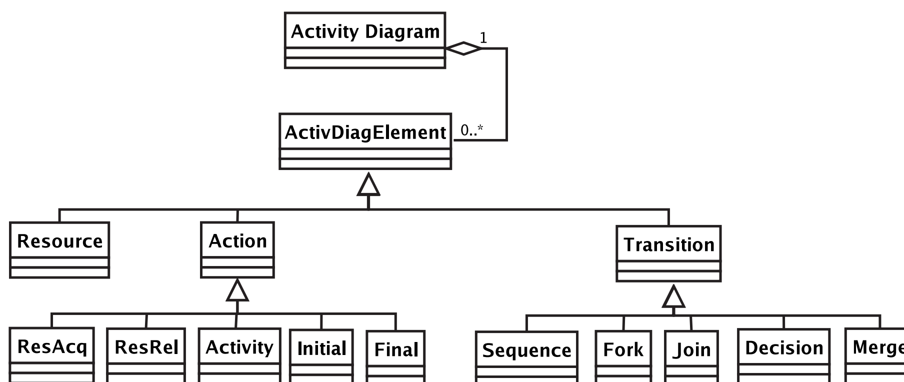


Figura A.6: Diagrama de Clases de la implementación del diagrama de Actividad

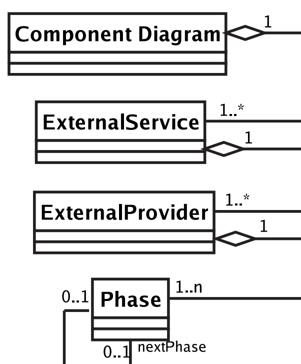


Figura A.7: Diagrama de Clases de la implementación del diagrama de Componentes

## A.3. Descripción de la generación de estrategias

El *workflow* de ejecución (ver figura A.1) es traducido a una red de Petri como la mostrada en la figura A.8.

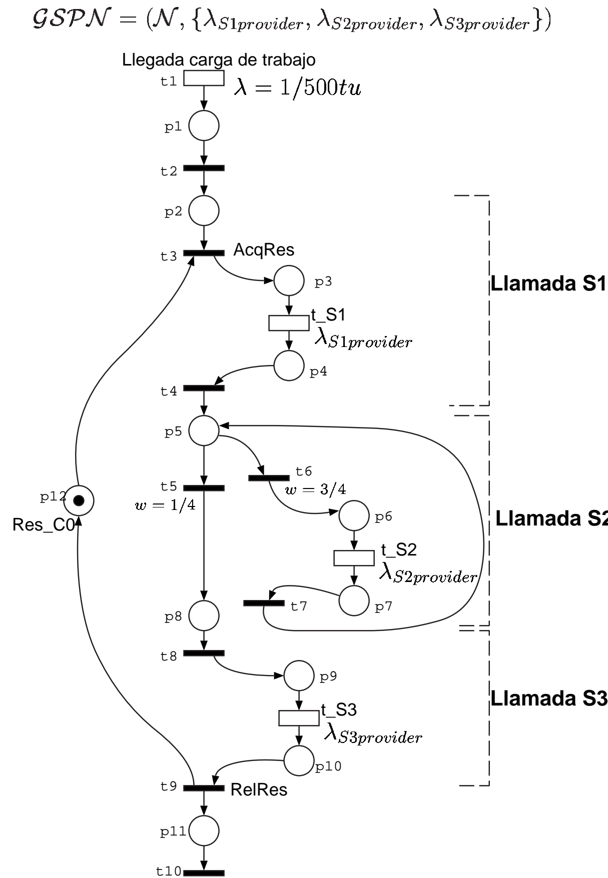


Figura A.8: GSPN paramétrica obtenida a partir del *workflow* del sistema

Debido a que la estructura de la red de Petri resultante para cada configuración del sistema es la misma, y lo único que cambia es el tiempo de algunas transiciones temporales, se emplea una red de Petri paramétrica. De esta manera, la traducción del *workflow* a red de Petri se realiza sólo una vez. Cada vez que es necesario simular el sistema, se modifican los parámetros de dicha red de Petri paramétrica y se realiza la simulación.

En el diagrama de Actividad se indica que las operaciones $S1$ y $S3$ se ejecutan una vez, y que la operación $S2$ es ejecutada tres veces en promedio (“extOpCount = 3”). La llamada a $S2$ se traduce con un elemento.

El recurso “C0” es traducido como un lugar con un token. El lugar se queda vacío cuando se dispara la transición “AcqRes”, justo antes de realizar la llamada a la operación $S1$; y vuelve a ser ocupado por el token cuando se dispara la transición de “RelRes”.

El primer paso del algoritmo es la creación del nodo inicial. Para ello, se toman las mejores fases de cada proveedor (ver tabla A.1): C11:fase1, C21:fase1, C22:fase1, C31:fase1, C32:fase1. Durante la ejecución del algoritmo de creación del nodo, se crean las cuatro configuraciones posibles. Se presentan a continuación, con la representación “(proveedor utilizado:fase)” para cada uno de los tres servicios, $S1$, $S2$ y $S3$.

**Conf1** (C11:fase1, C21:fase1, C31:fase1)
**Conf2** (C11:fase1, C21:fase1, C32:fase1)
**Conf3** (C11:fase1, C22:fase1, C31:fase1)
**Conf4** (C11:fase1, C22:fase1, C32:fase1)

La red de Petri paramétrica es instanciada para cada una de las cuatro posibles configuraciones del sistema. El resultado de la evaluación ha sido:

**Conf1** 60.5 u.t.
**Conf2** 177.7 u.t.
**Conf3** 72.5 u.t.
**Conf4** 193.8 u.t.

El algoritmo de creación de un nodo elige aquella con mejor tiempo de respuesta calculado:

(C11:fase1, C21:fase1, C31:fase1)

Esta configuración consituye el nodo inicial, $Nodo_0$ en la figura A.9. En este caso, la configuración obtenida como nodo inicial está compuesta por los proveedores que mejores tiempos de respuesta individuales tenían. Sin embargo, en otros sistemas es posible que el mejor tiempo de respuesta global no sea proporcionado por los mejores tiempos de respuesta locales, sino por otra configuración. Por ejemplo en sistemas donde varios proveedores pueden competir por recursos compartidos.

Una vez creado el nodo inicial, se considera que cada uno de los proveedores activos en la configuración del nodo inicial degrada su tiempo de respuesta. En ese caso, crea un nuevo nodo para cada posible proveedor degradado (C11, C21 y C31): los nodos $Nodo_1$, $Nodo_2$ y $Nodo_3$ respectivamente.

$Nodo_1$ es creado suponiendo que C11 degrada sus prestaciones. Para crear un nuevo nodo, se calculan las posibles configuraciones que mantienen la configuración de $Nodo_0$, modificando el proveedor y/o la fase de $S1$. En este caso existe una única configuración posible:

(C11:fase2, C21:fase1, C31:fase1)

Esta es la configuración elegida para $Nodo_1$, ya que es la única posible.

El arco entre $Nodo_0$ y $Nodo_1$ se etiqueta con el servicio S1, ya que es aquél cuyo comportamiento se ve modificado. Para calcular el nivel de confianza, se emplea tiene en cuenta que los componentes activos son los mismos en $Nodo_0$ y $Nodo_1$, y se calcula como el tiempo de respuesta de $Nodo_0$ dividido por el de $Nodo_1$:

$$nivelConfianza = \frac{tiempoResp_o}{tiempoResp_d}$$

Cabe observar que ningún arco sale de $Nodo_1$ con la etiqueta del servicio $S1$, debido a que todos los proveedores de $S1$ (C11) han pasado por todas las fases posibles (C11:fase1 y C11:fase2).

En la creación de $Nodo_2$, se calculan todas las configuraciones posibles:

- (C11:fase1, C21:fase2, C31:fase1)
- (C11:fase1, C21:fase2, C32:fase1)
- (C11:fase1, C22:fase1, C31:fase1)
- (C11:fase1, C22:fase1, C32:fase1)

Las dos primeras son las posibles configuraciones manteniendo el componente C21, el cual es considerado en su fase 2 (con prestaciones degradadas). En la tercera y la cuarta posibles configuraciones el componente proveedor del servicio S2 cambia de C21 a C22, el cual se considera en su fase1. Las cuatro posibles configuraciones son evaluadas a partir de la red de Petri. La tercera de ellas obtiene el mejor tiempo de respuesta, por lo que es elegida como $Nodo_2$.

El arco que une $Nodo_1$ y $Nodo_2$ está etiquetado con el servicio modificado entre ellos: S2. El nivel de confianza se calcula relacionando dos cantidades: la mejora potencial de prestaciones, calculada como lo que se gana al emplear C22 en su fase 1 en lugar de emplear el componente C21 con las prestaciones degradadas (fase 2); y la pérdida potencial de prestaciones debida a una mala predicción (falso positivo), esto es, se pasa a emplear C22 en su fase 1, sin embargo C21 se mantiene en su fase 1.

$$mejoraPotencial = tiempoResp_{emp} - tiempoResp_{d}$$

$$perdidaPotencial = tiempoResp_{d} - tiempoResp_{o}$$

$$nivelConfianza = \frac{mejoraPotencial}{mejoraPotencial + perdidaPotencial}$$

El resto de nodos y arcos del diagrama son creados de manera análoga.

Una vez creados todos los nodos y sus correspondientes arcos, el algoritmo genera los arcos de retorno. Estos arcos permiten volver a una configuración anterior en la que los componentes funcionaban en una fase mejor. Tomando un ejemplo, la configuración de $Nodo_1$ tiene un componente (C11) que no está en su mejor fase (que sería C11:fase1), sino en su fase 2. Este componente que está funcionando en una fase que no es su fase mejor, puede volver a su fase 1. Este cambio de fase de un componente que esté en cualquier fase (que sea distinta de su fase 1) hacia su fase 1 es representado con un arco de retorno. Los otros dos componentes de $Nodo_1$ (C21 y C31) están en su fase 1, por lo que no pueden pasar a una fase mejor y $Nodo_1$ no tiene más arcos de retorno.

## A.4. Estrategia de reconfiguración obtenida

La estrategia obtenida de ejecutar el Generador de Estrategias con el ejemplo propuesto en este anexo es la presentada en la figura A.9. Por claridad, solo se muestran dos de los arcos de retorno (los arcos en línea discontinua).

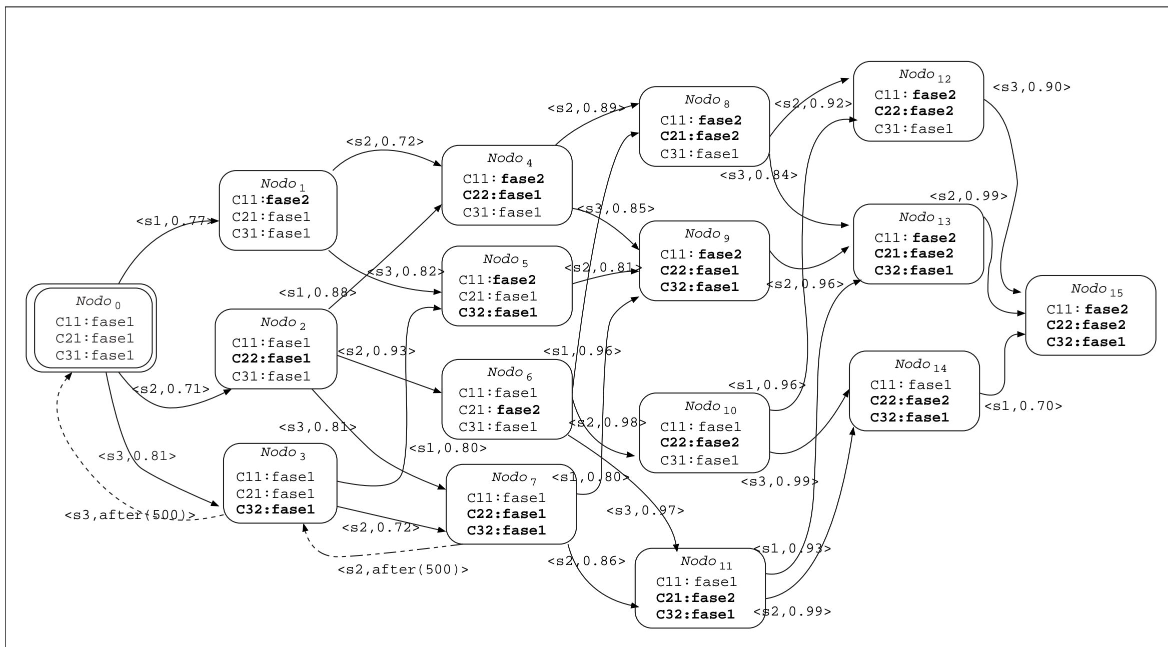


Figura A.9: Estrategia de reconfiguración obtenida

**Exportado de la estrategia al formato XML**

Esta estrategia de reconfiguración se exporta a un fichero XML. La estructura empleada para el fichero XML es la siguiente:

```
<ReconfigurationStrategy>
   <NodeList>
   (...)
   </NodeList>
   <EdgeList>
   (...)
   </EdgeList>
</ReconfigurationStrategy>
```

A continuación se presenta la traducción del nodo $Node_0$. Cada nodo es traducido de la misma manera y es incluido dentro del elemento `NodeList`.

```
<Node name="Node_0">
   <Phase providedService="S1" providerComponent="C11" phase="phase1">
      <ServiceTime>5</ServiceTime>
      <SojournTime>3000</SojournTime>
   </Phase>
   <Phase providedService="S2" providerComponent="C21" phase="phase1">
         <ServiceTime>10</ServiceTime>
         <SojournTime>6000</SojournTime>
   </Phase>
   <Phase providedService="S3" providerComponent="C31" phase="phase1">
      <ServiceTime>20</ServiceTime>
      <SojournTime>2000</SojournTime>
   </Phase>
</Node>
```

Tanto los nodos normales como los arcos de retorno son incluidos en el elemento `EdgeList`. Ejemplos de dos de ellos son mostrados a continuación.

```
<Edge source="Node_0" target="Node_1">
   <Service>S1</Service>
   <ConfidenceLevel>0.7576326433377804</ConfidenceLevel>
</Edge>

<WayBackEdge source="Node_1" target="Node_0">
   <Service>S1</Service>
   <TimeOut>after(500.0)</TimeOut>
</WayBackEdge>
```

Apéndice B

# El paquete ejecutable desarrollado

## B.1. El paquete ejecutable

El resultado del desarrollo del *software* es un paquete ejecutable que constituye el generador de estrategias. Se presenta en un fichero JAR (`StratGenerator.jar`) que realiza la generación de estrategias, representado en la figura B.1. Este paquete ejecutable puede integrarse en un sistema conectado a él a través de la interfaz `CreateNewStrategy`. Para dar respuesta a una petición de una nueva estrategia, el paquete necesita evaluar redes de Petri, requiere la interfaz `EvaluatePN`.

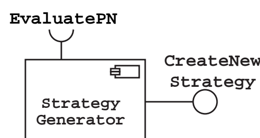


Figura B.1: Componente software desarrollado

Además, el generador de estrategias puede ejecutarse de manera independiente. Para ello, recibe como parámetros de entrada un fichero XMI, que debe contener el diagrama de Actividad y el diagrama de Componentes, y el fichero XML que representa la tabla de tiempos; y devolviendo como resultado una estrategia de reconfiguración.

Para parametrizar estos valores, el ejecutable consulta el fichero `settings.xml` (si existe). Dicho fichero puede contener los siguientes datos:

- **umlDiagFilename**: ruta del fichero XMI donde están contenidos los diagramas UML.
- **ttFilename**: ruta del fichero XML que almacena de tabla temporal asociada a cada componente.
- **directoryFilenamePN**: indica un directorio en el que almacenar las redes de Petri generadas como resultado intermedio.
- **strategyFilename**: ruta del fichero en el que se almacenará el resultado de salida (la estrategia generada) en XML.
- **PATH_WNSIM**: ruta del ejecutable para la simulación de redes de Petri de la herramienta GreatSPN. WNSIM acepta, entre otros, dos parámetros: accuracy y conflevel.
- **ACCURACY**: Indica el *accuracy level* (nivel de exactitud) con el que son realizadas las llamadas al simulador WNSIM.
- **CONFLEVEL**: Indica el *confidence level* (nivel de confianza) con el que son realizadas las llamadas al simulador WNSIM.

Un ejemplo de fichero `settings.xml` es el siguiente:

```
<settings>
   <umlDiagFilename>umlDiagrams/modelosICPE.uml</umlDiagFilename>
   <ttFilename>timeTable.xml</ttFilename>
   <directoryFilenamePN>./GSPNnets/</directoryFilenamePN>
   <strategyFilename>Strategy.xml</strategyFilename>
   <PATH_WNSIM>/usr/local/GreatSPN/WNSIM</PATH_WNSIM>
   <ACCURACY>5</ACCURACY>
   <CONFLEVEL>95</CONFLEVEL>
</settings>
```

## B.2. Diagrama de clases del componente desarrollado

En este anexo se muestra, en la figura B.2, el diagrama de clases del componente desarrollado.

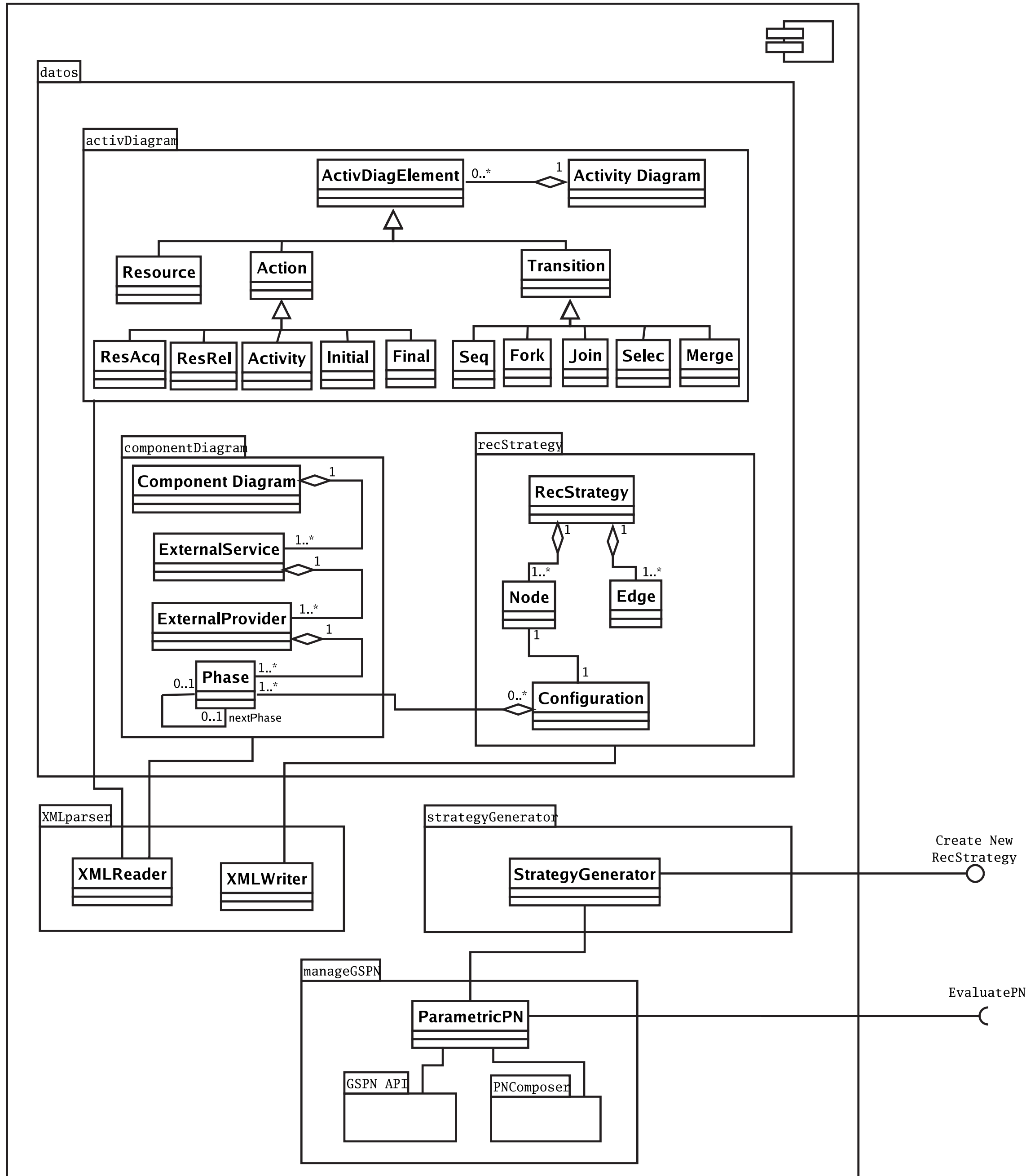


Figura B.2: Diagrama de Clases del componente desarrollado

Apéndice C

# Fases de desarrollo. Hitos y problemas encontrados

En este Apéndice se nombran los principales hitos y problemas encontrados en cada una de las fases de desarrollo en las que se dividió el proyecto.

1. **Definición del problema y estudio de la documentación**

   Hitos

   - Definición del plan de trabajo.

   - Lectura de los artículos de investigación en los que se ha basado el trabajo [1, 8] y de otra documentación de referencia sobre el paradigma *open-world* [2] y los sistemas autoadaptativos [5, 9].

2. **Obtención de los diagramas UML**

   Hitos

   - Seleccionar una herramienta CASE para el modelado de diagramas UML. Para ello hubo que evaluar su grado de madurez y cumplimiento de estándares.

   - Crear diagramas con Papyrus: diagrama de Actividad, diagrama de Componentes.

   - Realizar anotaciones MARTE en los diagramas.

   - Definir las estructuras de datos que almacenarían los diagramas de Actividad y diagramas de Componentes.

   - Comprender el tratamiento de XML con el paradigma DOM.

   - Realizar la lectura de los ficheros XMI empleando DOM.

   Problemas encontrados

   - La herramienta Papyrus es muy nueva (está aún en desarrollo). Pocos tutoriales y muy escuetos.

   - Modelado de las relaciones (de uso, de realización o de dependencia) entre las interfaces y los componentes en el diagrama de Componentes. Ya que la herramienta Papyrus modela de manera compleja estas relaciones.

3. **Evaluación de *workflows* mediante su traducción a Redes de Petri**

   Hitos

   - Crear una primera red de Petri.

- Traducir un elemento de un diagrama de Actividad a RdP.

- Traducir un diagrama de Actividad completo a RdP.

- Parametrizar las RdP.

- Escribir la red en un fichero del formato de GreatSPN.

- Elegir de manera automática qué transiciones y lugares emplear en la evaluación (fórmula de la Ley de Little).

- Ejecutar una llamada a GreatSPN.

- Leer el fichero de salida de GreatSPN.

Problemas encontrados

- Problemas con las redes de Petri paramétricas.

- Parametrizar qué transiciones y lugares consultar en la evaluación de cualquier RdP.

- La llamada a GreatSPN produce una gran cantidad de resultados intermedios por la salida estándar. Al realizar la llamada a GreatSPN desde el programa java, la salida estándar se almacena a un buffer o fichero de tamaño finito, que enseguida se llenaba. Tras probar diversas alternativas, la solución adoptada fue crear un script que ejecutase la función y despreciase la salida estándar.

4. **Definición y documentación de la estrategia de autoconfiguración**

   Hitos

   - Definición de las estructuras de datos en Java que almacenarían la estrategia.

   - Definición del formato del fichero XML

   - Escritura de una estrategia en XML.

5. **Implementación del algoritmo de generación de estrategias y pruebas**

   Hitos

   - Obtención de las posibles configuraciones en el algoritmo de crear nodo.

   - Creación del nodo inicial.

   - Cálculo del nivel de confianza.

   - Creación de los nodos de retorno.

6. **Elaboración de la documentación**

   Hitos

   - Definir la Propuesta de PFC.

   - Diseñar la estructura de la memoria.

   - Crear un primer documento en LaTeX.

   - Realizar los capítulos y anexos de la memoria.

Apéndice D

# MARTE

MARTE (***M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded systems)*[3] es un perfil de UML 2.0 para el modelado y análisis de sistemas embebidos y de tiempo real, incluidos los aspectos *software* y *hardware*. Proporciona soporte para las etapas de especificación, diseño y verificación del proceso de desarrollo *software*.

El estándar MARTE ha sido desarrollado por OMG para extender las capacidades de UML para el desarrollo de sistemas de tiempo real y sistemas embebidos. Tiene el objetivo de proporcionar un estándar común de modelado para mejorar tanto la comunicación entre desarrolladores como la interoperabilidad entre herramientas.

En este proyecto el perfil MARTE se emplea para añadir anotaciones de prestaciones (anotaciones temporales) a los diagramas UML. A continuación se presentan las anotaciones empleadas en este trabajo.

1. **GaWorkloadEvent**
   El estereotipo GaWortkloadEvent representa el conjunto de eventos que inician el comportamiento del sistema. Este puede ser generado de diferentes maneras, siguiendo una carga fija o abierta.

   El atributo "pattern" define el patrón de llegada de los eventos. Este "pattern" puede ser abierto o cerrado y puede seguir una función de probabilidad (Poisson, determinística u otras)

   El ejemplo de la figura D.1 representa una carga de trabajo de un sistema de tipo abierto, con una petición cada 200 unidades de tiempo (u.t.) en promedio y que sigue una distribución exponencial.

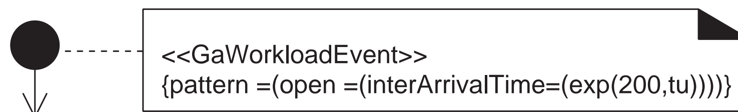


Figura D.1: Anotación GaWorkloadEvent de MARTE

2. **Resource**
   Este estereotipo marca un elemento de un diagrama UML como recurso. Al considerarse un recurso, éste puede ser adquirido (mediante la anotación `AcqRes`) o liberado (anotación `RelRes`) por otros elementos del diagrama UML.

   El ejemplo de la figura D.2 representa el estereotipado de un elemento UML, por ejemplo un componente, como `Resource`

3. **GaAcqStep**
   El estereotipo `GaAcqStep` representa la adquisición de un recurso. Cuando se realiza en una

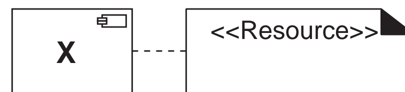


Figura D.2: Anotación Resource de MARTE

actividad, en primer lugar se adquiere el recurso y tras ello se realiza la actividad en sí. Los atributos empleados han sido `acqRes`, que indica qué recurso es adquirido; y `resUnits`, que indica el número de recursos adquiridos. La figura D.3 muestra un ejemplo de una anotación `GaAcqStep`. En ella, se adquieren dos unidades del recurso X.

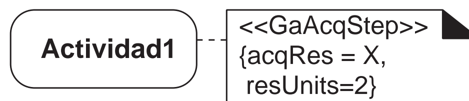


Figura D.3: Anotación GaAcqStep de MARTE

4. **GaRelStep**
   El estereotipo `GaRelStep` representa la liberación de un recurso. Los atributos empleados han sido `relRes`, que indica qué recurso es liberado; y `resUnits`, que indica el número. La figura D.5 muestra un ejemplo de una anotación `GaRelStep`. En ella, se liberan dos unidades del recurso X.

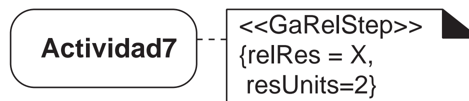


Figura D.4: Anotación GaRelStep de MARTE

5. **PaStep**
   El estereotipo `PaStep` representa la ejecución de una actividad. Los atributos empleados han sido `extOpCount`, que indica el número medio de ejecuciones de la operación; y `extOpDemands`, que indica el tiempo medio de respuesta de una ejecución. La figura D.5 muestra un ejemplo de una anotación `GaRelStep`. En ella, se ejecuta una operación una media de 3 veces y el conjunto de los identificadores de los servicios externos demandados por la actividad (`extOpDemands`).

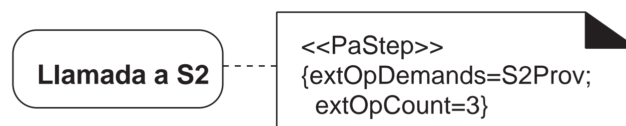


Figura D.5: Anotación GaRelStep de MARTE

## Apéndice E

# Traducción de los diagramas de Actividad de redes de Petri

El algoritmo 4 realiza la traducción de los diagramas de Actividad a redes de Petri. En primer lugar, se traduce cada uno de los elementos del diagrama de Actividad a una red de Petri parcial ($subRdP$), al mismo tiempo, se obtiene el conjunto de etiquetas que han sido generadas para posibilitar la composición de RdP. Tras ello, se realiza la composición de todas las redes en una sola.

**Algorithm 4** Traducción de diagrama Actividad a red de Petri

**Require:** diagrama de Actividades ($ad$)
**Ensure:** red de Petri ($rdp$)
  $listaSubRdp \leftarrow \emptyset$
  $etiquetasAComponer \leftarrow \emptyset$
  **for all** $elemento \in ad$.getElementos() **do**
    $\{$*Para cada acción o transición de ad*$\}$
    $\langle subRdp, etiq \rangle \leftarrow$ traduccionElemento($elemento$)
    $listaSubRdp \leftarrow listaSubRdp \cup subRdp$
    $etiquetasAComponer \leftarrow etiquetasAComponer \cup etiq$
  **end for**
  $rdp \leftarrow$ emptyRdP();
  **for all** $subRdp \in listaSubRdp$ **do**
    $rdp \leftarrow$ compose($rdp$, $subRdp$, $etiquetasAComponer$)
  **end for**
  embellecer($rdp$)
  **return** $rdp$

Una vez obtenida la red, ésta contiene todos los lugares, transiciones, *tokens* y arcos entre ellos. Sin embargo, si mostrásemos esta RdP con una herramienta gráfica (por ejemplo GreatSPN), los lugares y transiciones no tienen definidos unas posiciones en el espacio donde ser mostrados, por lo que todos los elementos serían representados en la misma posición. De esta manera, sería interpretable por una máquina, pero dificil de comprender por un ser humano. Se decidió crear un método para hacer las redes legibles. Esto sirvió para comprobar de manera sencilla la corrección de las redes creadas. La asignación de posiciones a los lugares y transiciones no puede realizarse durante la traducción, ya que durante ésta no se conoce si se está traduciendo un elemento del principio o del final de la red. Las figuras E.1 y E.2 representan un diagrama de Actividad y la red de Petri asociada generada automáticamente.

Los elementos del diagrama de Actividad y las anotaciones del perfil MARTE traducidos a

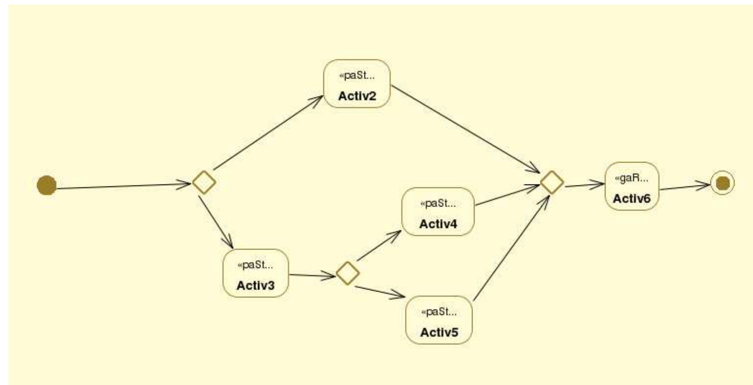


Figura E.1: Diagrama de Actividades a traducir

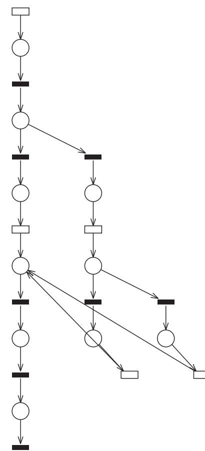

Figura E.2: Red de Petri obtenida de la traducción del diagrama de Actividad

LGSPN han sido los siguientes:

1. **Acciones:** engloban las acciones y los nodos inicial y final. En la traducción, los lugares y transiciones inicial y final son etiquetados con el nombre de la acción.

    - **Actividad:** puede ser una actividad sencilla de duración determinada o una actividad en la que se realiza una petición de un servicio a un cierto proveedor (anotado con MARTE), en ambos casos es traducida con transiciones temporizadas y lugares. Además, la actividad puede ser ejecutada en promedio una o más veces, modelado con transiciones con probabilidad.
    - **Nodo inicial:** los nodos iniciales son los encargados de gestionar la carga de trabajo que soporta el *workflow* (número de usuarios ejecutando o número de peticiones de ejecución por unidad de tiempo). Este proyecto se ha centrado en el número de peticiones por unidad de tiempo, lo que se entiende como carga abierta.

      El nodo inicial es traducido como una transición que "genera" *tokens* con una cierta función de probabilidad. Un ejemplo de ello es la transición $t1$ de la figura 4.4.
    - **Nodo final:** se traduce como una transición que no genera *tokens* al dispararse (no está seguida por ningún lugar).

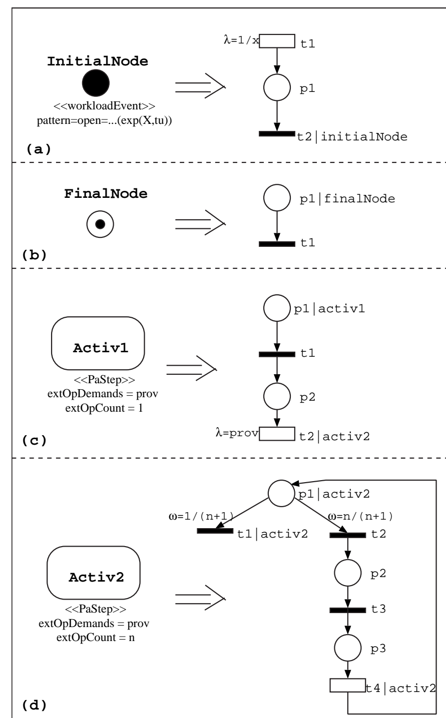


Figura E.3: Traducción a RdP de los elementos de Acción

2. **Transiciones de los diagramas de Actividad:** representan la unión entre dos o más acciones. Para posibilitar su correcta composición posterior, los lugares y transiciones son etiquetados con el nombre de las acciones predecesoras o sucesoras según corresponda.

   - **Secuencia:** representa la unión secuencial de dos acciones. Se traduce como una transición y un lugar que unen de manera directa dichas acciones.
   - **Selección:** representa la selección de uno de los flujos de ejecución sucesores a la selección. Se traduce con un lugar y varias transiciones con probabilidad, una por cada acción sucesora. De esta manera, cuando un *token* llegue a dicho lugar, sólo una de las transiciones se disparará y sólo uno de los flujos de ejecución serán ejecutados.
   - **Fin de selección:** es la transición en la que convergen varios flujos de ejecución diferentes generados por una o varias transiciones de selección. Se traduce con varias transiciones, una por cada flujo de ejecución predecesor, que convergen en un único lugar, de manera que independientemente de desde qué flujo de ejecución llegue el *token*, este único lugar obtendrá un único *token*.
   - **Bifurcación:** representa el inicio de la ejecución concurrente de varios flujos de ejecución. Se traduce con una transición y varios lugares, de manera que cuando la transición se dispara, se producen varios *tokens*, uno por cada flujo de ejecución.
   - **Fin de bifurcación:** es la transición en la que convergen varios flujos de ejecución concurrentes generados por una o varias bifurcaciones. Se traduce con varias transiciones que desembocan en varios lugares y una única transición, seguida por un único lugar, de manera que sólo cuando todos los lugares tengan algún *token* la transición será disparada. Los lugares contienen un *token* por cada flujo de ejecución que había antes del fin de bifurcación, y tras él la transición dispara un único *token*, permitiendo un único flujo de ejecución.

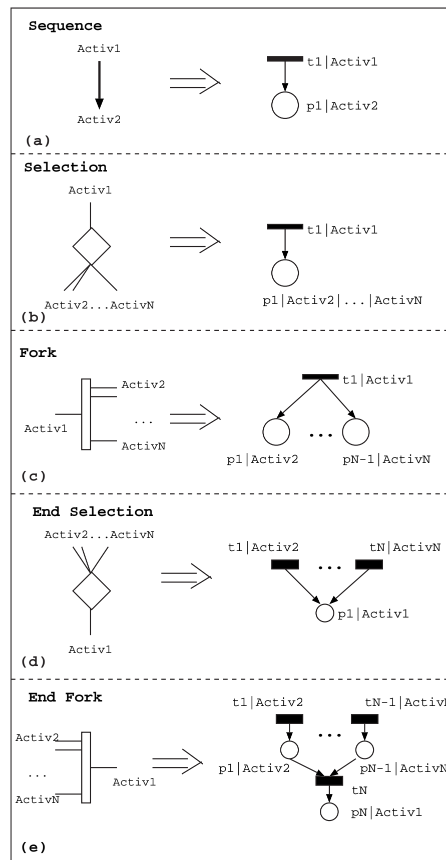


Figura E.4: Traducción a RdP de los elementos de Transición

3. **Adquisición y liberación de recursos:** la adquisición y liberación de recursos, así como el recurso en sí, son obtenidos a partir de las anotaciones MARTE del diagrama de Actividad y también son traducidas con elementos en la red de Petri resultante.

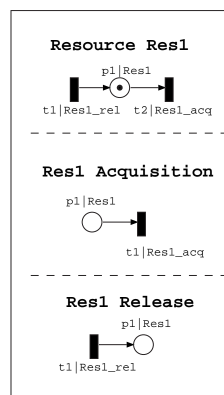


Figura E.5: Traducción a RdP del elemento Recurso

- **Recurso:** un recurso puede representar el sistema en sí o un recurso externo que el sistema deba emplear. En la red de Petri resultante de un diagrama de Actividad, un recurso se representa como un lugar con un token. Un ejemplo de representación de un recurso es el

lugar $p5$ de la figura 4.4.

- **Adquisición de recurso:** la adquisición de un recurso, denotada como una anotación MARTE anexa a una actividad, representa que una actividad adquiere un cierto recurso antes de comenzar a ejecutarse. Se traduce como un lugar y una transición unidos por un arco. Al componer las distintas redes parciales, el lugar será compuesto con el recurso en sí y la transición que será compuesta con el comienzo de la actividad asociada a la adquisición del recurso.
- **Liberación de recurso:** la liberación de un recurso, por su parte, representa la liberación de dicho recurso al finalizar una cierta actividad. Se traduce de manera equivalente a la adquisición de recurso. Se traduce como una transición y un lugar unidos por un arco desde la transición hasta el lugar. Al componer las distintas redes parciales, la transición será compuesta con el final de la actividad asociada a la adquisición del recurso y el lugar será compuesto con el recurso en sí.

Apéndice F

# Empleo de la herramienta GreatSPN

Como se ha explicado, la evaluación de las redes de Petri se realiza mediante la simulación de las redes de Petri. Para ello se ha empleado la herramienta GreatSPN mediante la invocación de una de sus funciones: “WNSIM”.

## F.1. Formato de los ficheros de GreatSPN

WNSIM necesita la red en el formato de GreatSPN, por lo que ésta es transformada a dicho formato. GreatSPN requiere dos ficheros para la definición de una red de Petri: `nombreRdP.net` y `nombreRdP.def`. La conversión de las redes de Petri a este formato es realizada mediante el empleo de una API desarrollada en el PFC de L.Arracó [26] para dicho objetivo.

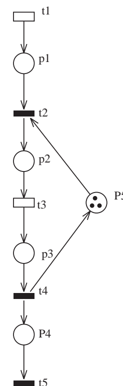


Figura F.1: Red de Petri correspondiente a simpleGSPN.def y simpleGSPN.net

Para ilustrar el formato de los ficheros, A continuación se presenta un ejemplo de los ficheros `simpleGSPN.def` y `simpleGSPN.net`, correspondiente a la red de Petri de la figura F.1, en los que se puede apreciar el particular formato de dichos ficheros.

- simpleGSPN.def

  ```
  |256
  %
  |
  ```

  Este fichero puede definir ciertas variables, como datos relacionados con la creación de redes de Petri coloreadas, las cuales no son relevantes en nuestro trabajo.

- simpleGSPN.net

```
|0|
|
f 0 5 0 5 1 0 0
p1 0 1.800000 0.916667 1.983333 0.900000 0
p2 0 1.800000 2.016667 1.966667 2.050000 0
p3 0 1.800000 3.050000 2.000000 3.100000 0
p5 3 2.616667 2.483333 2.816667 2.450000 0
p4 0 1.800000 3.966667 1.966667 4.000000 0
G1 1.816667 1.500000 1
t3 1.500000e+01 1 0 1 0 1.800000 2.483333 1.950000 2.558332 1.966666 2.566666 0
1 2 0 0
1
1 3 0 0
0
t1 2.000000e-03 0 0 0 0 1.800000 0.391667 2.000000 0.383333 1.966667 0.475001 0
1
1 1 0 0
0
t5 1.000000e+00 1 1 1 0 1.800000 4.516667 1.966667 4.550000 1.966667 4.600000 0
1 5 0 0
0
0
t4 1.000000e+00 1 1 1 0 1.800000 3.533333 1.966667 3.524999 1.966667 3.616667 0
1 3 0 0
2
1 4 0 0
1 5 0 0
0
t2 1.000000e+00 1 1 2 0 1.800000 1.483333 1.983333 1.516667 1.966666 1.566666 0
1 4 0 0
1 1 0 0
1
1 2 0 0
0
```

Las dos primeras líneas, en las que aparece `|0|` y `|`, son obligatorias para todos los ficheros `.net` de GreatSPN, pudiendo contener comentarios la segunda de las líneas.

La línea que comienza por `f` introduce la definición de los lugares en las cinco siguientes líneas. La definición de un lugar (por ejemplo `p5`) indica el número de tokens del lugar (3 en `p5`), las coordinadas del lugar ((2.616667,2.483333) en `p5`) y las coordenadas de su etiqueta ((2.816667,2.450000) en `p5`).

La línea que comienza por `G` introduce la definición de las transiciones. La definición de una transición almacena si esta es inmediata o temporizada, así como almacena si está parametrizada (y en ese caso almacenaría el parámetro asociado). De manera análoga a los lugares, almacena las coordenadas de su posición, de la posición de su etiqueta y de la posición de su throughput.

## F.2. Invocación del simulador

La invocación al simulador WNSIM se realiza con dos parámetros: la precisión o *accuracy* (con el parámetro `-a`) y el nivel de confianza o *confidence level* (parámetro -c):

```
WNSIM nombreRdP -a 5 -c 95
```

Con dicha invocación, el ejecutable simula la red con valores aleatorios hasta que las soluciones convergen con una precisión del 5 % en la estimación de cada valor, con un nivel de confianza del 95 %. El resultado de la simulación (`WNSIM simpleGSPN -a 5 -c 95`) se obtiene en un fichero con el siguiente formato (`simpleGSPN.simres`):

```
****** Simulation *******

MEAN NUMBER OF EVENTS : 1.907089
MAX NUMBER OF EVENTS : 4
MIN NUMBER OF EVENTS : 1

*********************************

Results :
Throughput of t3 (291.000000 ): 0.00199854914967 <= X <= 0.00200314835031
Value 0.00200084592788 Mean Value 0.00200084874999 Accuracy 0.114931241885
Throughput of t1 (291.000000 ): 0.00199847725768 <= X <= 0.0020030707145
Value 0.0020007711662 Mean Value 0.00200077398609 Accuracy 0.114791996831
Throughput of t5 (291.000000 ): 0.00199849923187 <= X <= 0.00200309921617
Value 0.00200079639826 Mean Value 0.00200079922402 Accuracy 0.114953670612
Throughput of t4 (291.000000 ): 0.00199855546029 <= X <= 0.00200315699845
Value 0.00200085340404 Mean Value 0.00200085622937 Accuracy 0.114989225369
Throughput of t2 (290.000000 ): 0.00199846879527 <= X <= 0.00200306609077
Value 0.00200076462455 Mean Value 0.00200076744302 Accuracy 0.114888302292
Mean n.of tokens in p1 : 0.000351026857691 <= mu <= 0.000387976220463
Value 0.000369499837469 Mean Value 0.000369501539077 Accuracy 4.99989294558
Mean n.of tokens in p2 : 0.1532063827 <= mu <= 0.153817455764
Value 0.153511650283 Mean Value 0.153511919232 Accuracy 0.199031146003
Mean n.of tokens in p3 : 0 <= mu <= 0
Value 0 Mean Value 0 Accuracy 0
Mean n.of tokens in P5 : 2.84618254424 <= mu <= 2.8467936173
Value 2.84648834972 Mean Value 2.84648808077 Accuracy 0.0107338067153
Mean n.of tokens in P4 : 0 <= mu <= 0
Value 0 Mean Value 0 Accuracy 0

Efficiency --->83759 transition firings per second
Time required for 17840841 events ------->213
Simulated time -------->1784124804.492880
Numero di campioni usati -------->7143
Grado di approssimazione -------->5
Livello di confidenza -------->4

*********************************
```

A partir de la lectura de este fichero `simpleGSPN.simres`, se puede obtener el *throughput* de cada transición (`t1`, `t2`, etc) o el número medio de tokens en cada lugar (`p1`, `p2`, etc). Partiendo de dicha información, se puede evaluar la red de Petri según lo explicado en la memoria principal de este trabajo (sección 4.4.2).

A modo de ejemplo, en el fichero mostrado, el troughput de la transición `t1` es 0.00200077398609 y el número medio de tokens en el lugar `p1` es 0.000369501539077.

## Apéndice G

# Glosario y abreviaturas empleadas

Este glosario define algunos de los conceptos empleados en la memoria principal, así como las abreviaturas empleadas.

***bind/unbind*** En el contexto de la Ingeniería del Software Basada en Componentes, el término *bind* denota la unión entre dos componentes y el término *unbind* denota su separación.

**CASE** *Computer Aided Software Engineering*. Las herramientas CASE son herramientas de apoyo al desarrollo del software. Algunas de sus funcionalidades pueden ser el modelado de sistemas con diagramas UML, el calculo de costes, implementación de parte del código automáticamente con el diseño dado, compilación automática o documentación entre otras.

**DOM** *Document Object Model* [27]. Es una interface independiente del lenguaje o de la plataforma que permite a los programas (o *scripts*) acceder y modificar el contenido de ficheros XML. Se emplea para generar documentos HTML dinámicos y para procesar ficheros XML desde un programa JAVA, entre otros. Representa la información del fichero XML como una estructura de árbol, que es cargado completamente en memoria. Es adecuado para realizar consultas aleatorias y búsquedas sobre un fichero XML.

**MARTE** *Modeling and Analysis of Real-Time and Embedded systems* [3]. Es un perfil de UML 2 para el modelado y análisis de sistemas embebidos y de tiempo real, incluidos los aspectos *software* y *hardware*. Proporciona soporte para las etapas de especificación, diseño y verificación del software.

**Metalenguaje** Lenguaje que describe un lenguaje.

**Metamodelo** Modelo que representa un modelo.

**OMG** *Object Management Group* [28]. Es un consorcio a nivel internacional que integra a los principales representantes de la tecnología de información Orientada a Objetos. El OMG tiene como objetivo central la promoción y el impulso de la industria Orientada a Objetos, proponiendo y adaptando especificaciones que se convierten en estándar ISO por defecto.

**OO** Orientación a Objetos. Paradigma para el desarrollo de sistemas de software que representa el dominio de aplicación basándose en los objetos que se implican en dicho dominio. Emplea diversos métodos para representar de forma abstracta los objetos, definiendo su estructura, comportamiento, agrupaciones, estados, etc.

**Open-world software** [2] Es un paradigma de desarrollo de aplicaciones *software* que propone la creación de sistemas *software* heterogéneos y distribuidos, donde el entorno de ejecución cambia de forma continua e impredecible (por ejemplo, en términos de disponibilidad de servicios y degradación o aumento de las prestaciones de los mismos).

**PNML** *Petri Net Markup Language* [25]. Es un estándar de intercambio de redes de Petri basado en XML. Pretende ser un formato estándar para el intercambio de ficheros entre diferentes aplicaciones que trabajen con redes de Petri.

**PFC** Proyecto Fin de Carrera.

**QoS** *Quality of Service* o Calidad del Servicio. La QoS es la capacidad para realizar un servicio con una calidad razonable. Por ejemplo, en la descarga de un fichero a través de internet el tiempo de inicio (retardo inicial) de la descarga puede ser largo porque la descarga sólo se inicia una vez, mientras que la velocidad de descarga (tasa de trasferencia) debe ser rápida para obtener una buena calidad de servicio. Otro ejemplo es una aplicación interactiva, donde el tiempo de inicio es más relevante que la velocidad.

**Red de Petri** Es un modelo matemático, una clase particular de grafo dirigido junto con un estado inicial [4]. El grafo subyacente $N$ de una red de Petri es bipartito, dirigido y con pesos, y consta de dos clases de nodos, llamados lugares y transiciones, de tal forma que los arcos van de un lugar a una transición o de una transición a un lugar.

**SAX** Simple API for XML [29]. Es una interfaz para el tratamiento de ficheros XML, originalmente desarrollada para JAVA. Trata el fichero XML de manera secuencial, por lo que es adecuado para el procesado secuencial de ficheros XML, pero no para realizar búsquedas o accesos aleatorios al mismo.

**SOA** *Service Oriented Architecture* o Arquitectura Orientada a Servicios. SOA puede definirse como una arquitectura de aplicación en la cual todas las funciones se definen como servicios independientes con interfaces invocables bien definidas. Estos servicios definidos por dichas interfaces pueden ser llamadas por otras aplicaciones para formar procesos de negocio.

**UDDI** *Universal Description, Discovery and Integration*. UDDI es un catálogo de negocios de Internet, en el contexto de los servicios Web. Es uno de los estándares básicos de los servicios web y su objetivo es ser servir de catálogo de servicios. Para cada servicio web se describen los requisitos del protocolo y los formatos del mensaje solicitado para interactuar con los servicios web del catálogo.

**UML** *Unified Modeling Language* [10]. El estándar UML es una especificación semi-formal que define un lenguaje gráfico que sirve para visualizar, especificar, construir y documentar los elementos de un sistema *software* durante todas las etapas de su ciclo de vida, siguiendo el paradigma orientado a objetos. En él también se definen reglas semánticas y de corrección de modelos a través del lenguaje OCL (Object Construction Language). Fue adoptado por Object Management Group (OMG) en 1998 y es un estándar ISO.

UML define doce tipos distintos de diagramas gráficos que sirven para describir todas las vistas que un modelo puede necesitar para ser caracterizado, enfocados desde el paradigma OO. Estos diagramas se agrupan a su vez en tres categorías:

- Diagramas estructurales: Diagrama de clases, diagrama de objetos, diagrama de componentes y diagrama de despliegue. Permiten modelar los aspectos estáticos de un sistema, como las clases que lo consituyen o dónde se encuentran distribuidas.
- Diagramas de comportamiento: Diagrama de casos de uso, diagrama de secuencia, diagrama de actividades, diagrama de colaboración y diagrama (o máquina) de estados. Permiten modelar los aspectos dinámicos del sistema, esto es, su comportamiento en ejecución.

- Diagramas de organización del modelo: Diagrama de paquetes, subsistemas y modelos. Estos diagramas no representan o describen aspectos del sistema en sí sino que se refieren a aspectos de organización interna del sistema para hacerlo más fácilmente entendible.

**XMI** *XML Metadata Interchange* [6]. Es una especificación de OMG basada en XML para el intercambio de modelos UML entre diferentes herramientas de modelado.

**XML** *Extensible Metadata Language* [12]. Es un metalenguaje basado en etiquetas que permite la definición de lenguajes para diferentes necesidades. Algunos lenguajes y especificaciones basados en XML son XHTML, SVG o XMI.

**XSLT** *XSL Transformations* [30]. Es un lenguaje para la transformación de ficheros XML en otros ficheros (generalmente en formato XML, pero también a otro formato). Se basa en XSL (Extensible Stylesheet Language) [31], una especificación para la transformación y presentación de XML.