Proyecto Fin de Carrera de Ingeniería en Informática

Evaluación del rendimiento del software: Diagramas de colaboración y composición con máquinas de estados.

Aitor Acedo Legarre

Director: José Merseguer Hernáiz
Codirectora: María Elena Gómez Martínez

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática e Ingeniería de Sistemas

Diciembre 2005

Agradecimientos

Me gustaría agradecer la dedicación y esfuerzos mostrados en todo momento por mis directores, José y Elena, ellos han aportado lo máximo para hacerme comprender todas las dudas que me han surgido, ¡mil gracias!.

Tengo que añadir en este apartado a una persona que sin estar directamente relacionada con mi proyecto, me ha escuchado pacientemente y ha sabido corregirme cuando ha hecho falta, gracias por todo Simona.

A todos los compañeros que han conseguido que mi paso por la universidad sea un excelente recuerdo, sois demasiados así que no escribiré nombres, pero os lo agradezco de todo corazón.

A mis amigos, Martín, Daniela, Iván, David, Fran, y a los que me dejo en el tintero, os debo mucho.

A Laura, que con su apoyo ha logrado que muchos momentos se hicieran menos cuesta arriba.

Por último, y no por ello menos importante, quiero dar las gracias a mis padres, Gregorio y Mª Teresa, y a mi hermana Paula por la paciencia y el cariño que me han brindado desde siempre, no lo habría podido hacer sin vosotros.

The ideal situation occurs when the things that we regard as beautiful are also regarded by other people as useful.

— Donald Knuth

Índice general

Índice de figuras

Parte I

Memoria

Capítulo 1

Introducción

En este capítulo se van a explicar los aspectos que hemos considerado más generales relacionados con nuestro proyecto fin carrera, dentro de estos aspectos nos encontramos con la motivación que me ha llevado a la realización de este trabajo, el ámbito en el que se enmarca, los conceptos que serán utilizados en el proceso de su elaboración, los objetivos que se definieron en la propuesta, las fases en las que hemos organizado nuestro trabajo y por último una breve explicación de la estructura del presente documento.

1.1. Motivación

Cuando se comenzó con este proyecto no dejaba de resultarme atractiva la idea de que estaba contribuyendo, de alguna manera, al crecimiento de herramientas de software libre (ArgoUML [3] y ArgoSPE [2]), lo cual fue una de mis principales motivaciones.

Unido a esto estaba también el hecho de que podría adquirir gran cantidad de conocimientos sobre temas tan interesantes como la ingeniería y la arquitectura del software. Sin olvidar, por su puesto, la puesta en práctica de muchos de los conceptos adquiridos durante la carrera.

1.2. Ámbito

Este proyecto nace como resultado del trabajo de investigación efectuado por el profesor del Grupo de Ingeniería de Sistemas de Eventos Discretos (**GISED**), Dr. José Javier Merseguer en colaboración con otros doctores, como Simona Bernardi, y doctorandos, en especial María Elena Gómez, en relación con la traducción de diagramas de **UML** a modelos formales de redes de Petri, motivada por el análisis de las prestaciones de un sistema software.

Este trabajo de investigación se materializó en una serie de publicaciones, que posteriormente citaremos, y en la tesis doctoral de dicho profesor.

Toda esta labor también ha traído consigo la elaboración de varios proyectos fin de carrera, entre los más destacados se encuentran, *Evaluación del rendimiento del software: Traducción de UML (máquinas de estado + diagramas de actividad a GSPN)*, por Borja Fernández Bacarizo, *Análisis automático de prestaciones de sistemas a partir de modelos en UML, mediante traducción a redes de Petri generalizadas*, por Isaac Trigo, y éste que nos ocupa, que es extensión de los dos anteriores.

1.3. Contexto del proyecto

El presente proyecto queda ubicado en diferentes áreas de la informática que serán detalladas ampliamente en el transcurso de las secciones posteriores. Dentro de estas áreas podemos destacar, la Ingeniería del Software, la Ingeniería de Prestaciones del Software (**SPE**), las redes de Petri estocásticas generalizadas (**GSPN**) y el estándar de modelado UML.

Aparte también lo ubicaremos en el contexto de ArgoUML, herramienta CASE de código libre, y uno de sus módulos, ArgoSPE, que apareció fruto de los proyectos previamente mencionados. Ambas herramientas, programadas en Java, residen como proyectos en la comunidad de desarrollo de software libre, relacionada con la Ingeniería, **Tigris** [29].

1.4. Objetivos

Este proyecto fin de carrera consiste fundamentalmente en desarrollar una plataforma software que permita **la definición del comportamiento de un sistema informático**, y su correspondiente **evaluación de prestaciones.** Dicho comportamiento se concreta en el modelado de escenarios que definan diferentes usos del sistema para su posterior evaluación cuantitativa y/o cualitativa. Los escenarios serán descritos por medio de diagramas de colaboración.

Para la consecución de este objetivo es preciso completar varios objetivos previos. Lo primero será la **traducción** de los diagramas de colaboración a modelos de redes de Petri estocásticas que sean capaces de ser analizadas con **GreatSPN** [14], herramienta que utilizaremos en etapas posteriores para la evaluación cuantitativa propuesta. El resultado de la composición será una red de Petri que representa el comportamiento de los escenarios propuestos en el contexto del sistema modelado.

El siguiente paso consistirá en **componer** las redes resultantes de la fase anterior con las redes de Petri que ya generaba ArgoSPE al traducir las máquinas de estados y los diagramas de actividades, esto se realizará en parte con la ayuda de un programa denominado *algebra* el cual acompaña a GreatSPN y su funcionamiento quedará detallado en capítulos posteriores.

Y por último tendremos que implementar alguna **consulta** que sea interesante desde el punto de vista del analista que modela el sistema informático. Dicha consulta se refiere a interrogar el modelo de red de Petri resultante de la composición anterior. La consulta tendrá que proporcionar información útil que ayude a detectar problemas en el modelo para su posterior corrección.

1.5. Herramientas utilizadas

Para realizar el desarrollo software simplemente se ha necesitado un editor de texto y el entorno de desarrollo implementado por Sun, que nos proporcione la máquina virtual de Java. Además también se han utilizado otras herramientas que han proporcionado una serie de considerables beneficios, ésta es la lista de todas ellas:

- **Java 2 SDK (J2SE)**: Como es lógico necesitaremos el entorno de desarrollo en Java, en nuestro caso la versión 1.5.0_02.

- **JBuilder**: Éste es el entorno de desarrollo para Java que se comenzó a utilizar, aunque tras un cierto tiempo usándose se decidió utilizar un editor menos *pesado*.

- **Vim**: Éste es el editor de textos que ha sido utilizado para modificar los ficheros fuente que hemos requerido, se ha utilizado a modo de entorno de desarrollo *ligero*, gracias a muchas opciones que posee, como por ejemplo grep interno y ctags entre otros.

- **Ant**: Es una herramienta, de código libre, basada en las tecnologías Java y XML; empleada en la compilación y creación de programas Java. Es muy similar a la utilidad *make*. Tanto ArgoUML como ArgoSPE la utilizan en su proceso de desarrollo y construcción. Su manejo es muy sencillo aunque se le encontraron algunos posibles problemas[1].

- **Issuezilla**: Es básicamente una base de datos para errores de programación empleada en la gestión de proyectos. La comunidad Tigris la emplea para sus proyectos y es accesible vía Web. Esta aplicación nos permite registrar, informar y discrepar sobre problemas o asuntos del código, además de asignar desarrolladores a cada asunto.

- **CVS**: Es el sistema de versiones concurrentes utilizado tanto por nuestro módulo como por la herramienta CASE. Este programa gestiona los repositorios donde se almacenan los fuentes de las anteriores aplicaciones. Permite a varios desarrolladores trabajar paralelamente sobre el mismo código. Esta muy arraigado dentro de la comunidad Tigris ya que facilita el control en proyectos grandes.

- **GreatSPN 2.0.2**: Es un paquete software que permite el modelado, la validación y la evaluación de prestaciones de sistemas distribuídos usando redes de Petri estocásticas generalizadas y su extensión coloreada. Esta herramienta ha sido desarrollada en la Universidad de Turín.

Junto con estas herramientas, cabe destacar dos de los estándares que ArgoUML soporta y que han resultado indispensables a la hora del desarrollo de este trabajo:

- **UML**[2]: Lenguaje Unificado de Modelado (**U**nified **M**odeling **L**anguage). Utilizado para especificar, visualizar y documentar modelos de sistemas de software.

- **XMI**: Siglas de **X**ML **M**etadata **I**nterchange. Se basa en XML y es utilizado para almacenar e intercambiar datos entre herramientas de modelado.

1.6. Fases del trabajo

Tras conocer los objetivos que engloban a este proyecto, lo más sensato fue organizar nuestro esfuerzo de una manera lógica, es decir, las primeras fases consistirían en la adquisición de los conceptos teóricos que se iban a utilizar en el transcurso del proyecto, para posteriormente dedicarnos a su implementación. Con todo esto las principales etapas de nuestra labor, fueron las siguientes:

[1] La limitación de visibilidad en métodos llamados por otras clases, son fuente de ejecuciones no deseadas.
[2] ArgoUML está basado en la versión 1.3 de este estándar.

1. **Estudio del estándar UML**: con el fin de conocer la semántica inherente a los diagramas de colaboración, así como los elementos que podían llegar a formar parte de ellos. Tenemos que destacar que los documentos utilizados se referían a la versión 1.4.2 de UML[3]. Además se estudiarán los diagramas de máquinas de estados y de actividades, pues será necesario componerlos con los de colaboración.

2. **Utilización de la herramienta ArgoUML**: lo cual nos proporcionará un conocimiento del nivel de implementación de los elementos en los que estamos interesados para nuestro trabajo, éstos serán tanto los diagramas de colaboración como los diagramas de máquinas de estados.

3. **Comprensión del proceso teórico de traducción y composición**: Básicamente consistía en el estudio de una parte de la tesis doctoral [17], así como diferentes artículos relacionados [5, 6, 16, 18]. Este punto es fundamental para evitar en la medida de lo posible, problemas a la hora de la implementación del módulo. Hay que destacar la dificultad de esta fase, por la complejidad de estas cuestiones.

4. **Estudio del módulo ArgoSPE**: cuyo objetivo será el familiarizarse con el manejo de esta aplicación y con su diseño, ya que todo este trabajo nos facilitará bastante la fase de implementación de nuestro trabajo. Este estudio consistió básicamente en entender el código desarrollado en los proyectos [4, 7].

5. **Estudio de la aplicación GreatSPN, de algebra y de las redes de Petri estocásticas**: en esta etapa nos familiarizamos con el formato de los ficheros manejados por GreatSPN, estudiamos la teoría de las GSPN's, y comprobamos el funcionamiento de *algebra* para su posterior uso.

6. **Fase de implementación** de la traducción, composición y posterior consulta de prestaciones de los diagramas de colaboración.

1.7. Organización del documento

Dejando a un lado este capítulo introductorio, el presente documento se encuentra estructurado en dos partes, la primera engloba la memoria de este trabajo y la segunda parte la constituyen los anexos, la parte inicial está estructurada como sigue:

- **Capítulo 2. Conceptos Previos**: a lo largo de este capítulo intentaremos ofrecer una explicación de las ideas y teorías que han sido utilizadas para la realización de este proyecto.

- **Capítulo 3. Planificación del trabajo**: aquí describimos cómo se ha organizado el trabajo a lo largo del tiempo y mis conclusiones al respecto.

- **Capítulo 4. Resultados y conclusiones**: este capítulo es uno de los más importantes de la memoria. En él se explican las dificultades encontradas, se resume el trabajo realizado y se presentan diversas conclusiones.

La parte de los anexos está constituída por:

[3]Los cambios entre esta versión y la 1.5 son mínimos, con lo cual podremos hablar indistintamente de las dos.

- **Anexo A. Análisis de UML**: en este apéndice realizamos una explicación detallada de UML con objeto de profundizar en los diagramas de Estados y de Colaboración y en los elementos que en ellos se definen así como su semántica, su notación y otros aspectos reseñables.

- **Anexo B. ArgoUML**: durante este anexo queremos presentar las carencias que se han detectado en la implementación de los diagramas de colaboración de la herramienta ArgoUML. La segunda parte de este apéndice explica la representación realizada por ArgoUML, en su versión 0.18.1, de los modelos UML en formato XMI (ver B.2).

- **Anexo C. Implementación de soluciones**: ofrece una explicación detallada de los aspectos que he considerado más relevantes en la implementación llevada a cabo en este trabajo, los puntos más destacados son los errores subsanados, el proceso de traducción y la implementación de la composición.

- **Anexo D. Caso práctico**: intenta ser una referencia para aquellas personas cuyo trabajo tenga relación con la herramienta ArgoSPE, para evitar en la medida de lo posible los problemas iniciales que puedan surgir con el manejo de esta aplicación.

- **Anexo E. The GNU General Public License**: como su nombre indica es la Licencia Pública General de GNU, bajo la que se ha desarrollado nuestra aplicación.

- **Anexo F. Glosario**: este último apéndice intenta reunir todos aquellos conceptos que han resultado especialmente útiles para la lectura y comprensión de la documentación bibliográfica recogida en este proyecto.

Capítulo 2

Conceptos Previos

En el capítulo que nos ocupa vamos a poner de manifiesto algunos de los aspectos más relevantes dentro de cada una de las áreas que han sido objetivo de mi proyecto fin de carrera.

El objetivo del presente capítulo es poner en contexto al lector en las diferentes teorías y herramientas utilizadas en nuestro trabajo con el fin de que se pueda entender el resto de los capítulos. Obviamente estas pequeñas introducciones a cada una de las teorías no ponen de manifiesto la profundidad que el proyectando ha adquirido en cada una de ellas.

2.1. Estado del Arte

Hoy en día el número de usuarios de herramientas informáticas ha crecido enormemente con respecto a épocas pasadas, este fenómeno es debido fundamentalmente al gran auge de Internet, que ha abierto las puertas de la informática a un cuantioso colectivo de personas.

Por otro lado las empresas que desarrollan software sacan al mercado aplicaciones cada vez más complejas, con un número mayor de funcionalidades que intentan mejorar las ya existentes, y las cuales se desarrollan a una velocidad vertiginosa.

La alta exigencia de calidad, robustez, eficiencia y funcionalidad por parte de todos esos usuarios que hemos comentado, hace que la industria del software tenga que recurrir a disciplinas que ayuden a conseguir todos estos propósitos.

Este es el caso de la **Ingeniería del software** [24] que se encarga de optimizar el proceso de desarrollo de un producto software, con el fin de hacer cumplir los requisitos impuestos evitando prácticas que puedan retrasar la finalización del producto.

El problema existente es que esta disciplina no está enfocada a asegurar las características del software relacionadas con su rendimiento, lo que obliga contínuamente a valorar las prestaciones de un sistema en sus fases finales de desarrollo, con una versión funcional.

Esta práctica conlleva riesgos inevitables, por ejemplo, podría darse el caso de que una vez conseguida un versión funcional, y tras realizar una serie de pruebas llegaramos a la conclusión de que no podemos alcanzar los niveles de prestaciones esperados, con lo que habríamos desaprovechado una gran cantidad de tiempo que no podríamos recuperar.

Una solución diferente sería el adelantar el análisis del rendimiento de este tipo de

productos a las etapas iniciales de sus ciclos de vida, con el consiguiente ahorro del tiempo invertido en desarrollar un producto, que no va a cumplir con las prestaciones requeridas. Este concepto es la piedra angular de un campo de investigación conocido como **Ingeniería de Prestaciones del Software** o **SPE** [28].

En este punto aparece el problema de cómo incluir este nuevo enfoque al proceso de desarrollo del software actual. UML es, hoy en día, el estándar de facto en la ingeniería del software y es utilizado por la gran mayoría de personas dedicadas al mundo del desarrollo, lo cual lo convierte en el candidato idóneo para incluirle las prestaciones de los sistemas.

Ésta es a grandes rasgos la idea que han seguido los trabajos de investigación realizados en el GISED comentados anteriormente, y que mencionaremos más adelante.

Todavía quedan en el aire preguntas importantes como por ejemplo, ¿**cómo** incluiremos las medidas de prestaciones en UML?, ¿**qué modelo** utilizaremos para analizar las **prestaciones** que nos van a interesar?, todas estas cuestiones quedarán respondidas en las secciones siguientes.

2.2. UML

El Object Management Group (**OMG**) [21] es un consorcio a nivel internacional que integra a los principales representantes de la industria de la tecnología de información Orientada a Objetos (**OO**). El OMG tiene como objetivo central la promoción, y el impulso de la industria OO. Éste propone y adopta por consenso especificaciones en torno a la tecnología OO, especificaciones que se convierten en estándar **ISO** por defecto.

Una de las especificaciones más importantes es la adopción en 1998 de UML como un estándar. Éste se integra dentro de Model Driven Architecture (**MDA**) de OMG, que es a la postre un conjunto de estándares que sirven para planificar y controlar el ciclo completo de vida del software, independientemente de la plataforma para la que se desarrolla ese software. Y es aquí donde queda clara su relación con lo que conocemos como Ingeniería del Software.

UML es pues una especificación semi-formal que define un lenguaje gráfico que nos sirve para visualizar, especificar, construir y documentar los artefactos o elementos de nuestro sistema. En él también se definen reglas semánticas y de corrección de modelos lo que lo convierten en algo más que un lenguaje gráfico.

UML define **doce** tipos distintos de diagramas gráficos, que sirven para describir todas las vistas que un modelo pueda necesitar para ser caracterizado, vistas que se ajusten al paradigma OO.

Estos doce diagramas se agrupan a su vez en tres categorías: la que incluye diagramas que definen estructuras o elementos estáticos, la compuesta por diagramas que definen diferentes aspectos del comportamiento dinámico y la que engloba diagramas que definen como organizar los módulos de una aplicación. Éstas son las tres categorías respectivas, junto con los nombres de los diagramas que forman parte de ellas:

- **Diagramas estructurales**: diagrama de clases, diagrama de objetos, diagrama de componentes y diagrama de despliegue.
- **Diagramas de comportamiento**: diagrama de casos de uso, diagrama de secuencia, diagrama de actividad, diagrama de colaboración y diagrama (o máquinas) de estados.

- **Diagramas de organización del modelo**: diagramas de paquetes, subsistemas y de modelos.

Es momento de realizar una breve presentación de los dos tipos de diagramas que participan más activamente en mi proyecto: Máquina de Estados y Diagrama de Colaboración.

2.2.1. Máquina de estados

Una **máquina de estados** se utiliza para modelar los aspectos dinámicos de un sistema. La mayoría de las veces esto supone el modelado del comportamiento de objetos que reaccionan ante los eventos lanzados desde fuera de su contexto.

Los estados son los elementos primordiales de este tipo de diagramas. Un estado es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. En esta figura se muestra un diagrama de estados genérico:

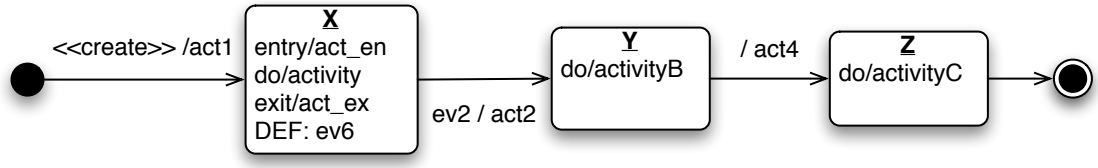

Figura 2.1: Ejemplo de Máquina de Estados.

2.2.2. Diagrama de colaboración

Un **diagrama de colaboración** es un diagrama de interacción que destaca la organización estructural de los objetos que se comunican entre sí. Los diagramas de interacción muestran las relaciones existentes entre un grupo de objetos, incluyendo los mensajes que se pueden enviar entre ellos.

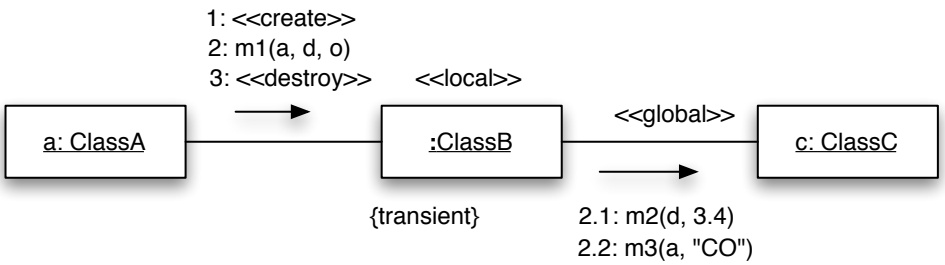

Figura 2.2: Ejemplo de Diagrama de Colaboración.

Uno de los elementos más importantes dentro del diagrama de colaboración será el *número de secuencia*, éste indicará la ordenación temporal del mensaje, este número se forma con un dígito que se irá incrementando secuencialmente por cada nuevo mensaje en el flujo de control. Para soportar el anidamiento, se utiliza la *numeración decimal de*

Dewey [9] en la que, el 1 representa al primer mensaje, el 1.1 será el primer mensaje dentro del mensaje 1, el 1.2 será el segundo mensaje dentro del mensaje 1, soportando cualquier nivel de profundidad.

Por último decir que los diagramas de colaboración y los diagramas de secuencia son semánticamente equivalentes, debido a que derivan de la misma información del metamodelo de UML, lo cual hace que presenten unas capacidades similares de expresividad.

2.2.3. Mecanismos de extensión

Estos mecanismos son proporcionados para posibilitar la expresión de todos los matices posibles de todos los modelos en cualquier dominio, en nuestro caso en el dominio de las prestaciones. Los mecanismos con los que contamos en este lenguaje son:

- **Estereotipos**: permite la creación de nuevos bloques de construcción que deriven de los existentes pero que sean específicos del sistema.
- **Valores etiquetados**: extiende las propiedades de un bloque de construcción de UML, para añadir nueva información en la especificación de ese elemento.
- **Restricciones**: extiende la semántica de un bloque de construcción.

Los elementos que utilizaremos para ser capaces de anotar los modelos de UML con características de prestaciones serán los estereotipos y los valores etiquetados.

2.3. UML-SPT

Desde su nacimiento UML ha sido utilizado en el desarrollo de multitud de sistemas con restricciones como tiempo, recursos, prestaciones, o seguridad. Con el paso del tiempo se comprobó que este lenguaje carecía de la expresividad suficiente como para poder modelar estas características, lo cual limitaba su expansión a campos como sistemas de tiempo real o sistemas empotrados.

A pesar de esto se vio que era posible utilizar los mecanismos de extensión (ver 2.2.3) que facilita con el fin de introducir este tipo de características dentro de nuestros modelos.

El OMG elaboró un documento denominado, **UML** Profile for **S**chedulability, **P**erformance and **T**ime Specification (**UML-SPT**) [22], en el cual se recogen los métodos comúnes para enriquecer un modelo de UML con información para su posterior análisis de prestaciones. Estos métodos son los que ha seguido ArgoSPE en su implementación.

La organización de este documento sigue un esquema lógico, cada capítulo nos instruye en un dominio concreto, por ejemplo el modelado general del tiempo, o el de la concurrencia. Todos los capítulos constan de dos partes, la primera introduce los conceptos teóricos del dominio al que nos estamos referiendo y la segunda explica como modelarlos con los elementos que nos proporciona UML, esta estructura queda reflejada en el siguiente esquema:

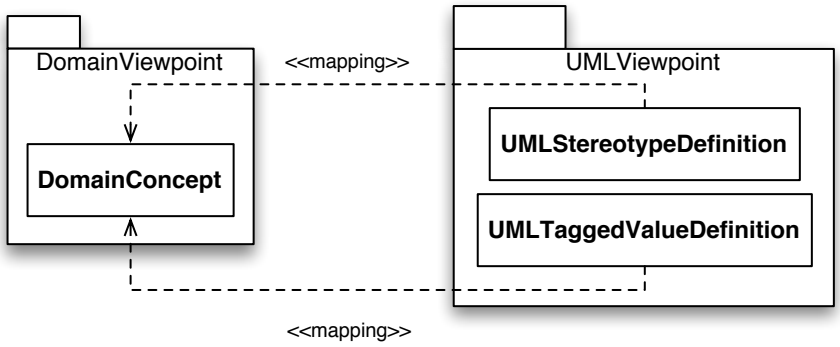

Figura 2.3: Relaciones entre el punto de vista del dominio y el de UML.

La figura 2.3 ilustra uno de estos conceptos del dominio en cuestión, el cual será representado por un par, estereotipo-valor etiquetado en UML. Los términos más interesantes desde el punto de vista del modelado de prestaciones según el UML-SPT son los siguientes:

- **Contexto**: representa una situación relevante para el diseñador, dentro de nuestro sistema. Está constituído por varios escenarios.
- **Escenario**: es una secuencia de 1 o más pasos, los cuales están ordenados de forma que podremos establecer relaciones de predecesores y sucesores entre los mismos. Se utilizan para explorar varias situaciones dinámicas que envuelven a un conjunto de recursos.
- **Paso**: incremento en la ejecución de un escenario particular utilizando ciertos recursos.
- **Recurso**: es visto como un servidor que posee un tiempo de servicio, podemos ver a los pasos como clientes de estos servidores.
- **Carga de trabajo**: indica la intensidad de la demanda para la ejecución de un escenario específico. Existen dos tipos: las *abiertas* que modelan un flujo de peticiones, y las *cerradas* que caracterizan un número constante de usuarios.

Dentro de la evaluación de prestaciones los escenarios juegan un papel importante puesto que las prestaciones son propiedades dinámicas de los sistemas, para modelarlos utilizando UML tendremos que utilizar algún elemento que represente también propiedades dinámicas, en nuestro caso, los **diagramas de colaboración** (ver 2.2.2). La representación concreta de los conceptos mencionados con anterioridad quedará expuesta en el anexo D.

2.4. Redes de Petri

En la investigación previa a mi trabajo se decantaron por las redes de Petri como formalismo para la evaluación de prestaciones. Después de una comparativa entre diferentes formalismos como por ejemplo las redes de colas [19] o las álgebras de procesos [15], se llegó a la conclusión de que las redes de Petri proporcionaban un nivel de detalle mayor

para modelar los sistemas que cualquiera de las anteriores. Ahora vamos a proceder a una breve explicación.

Las redes de Petri [26] (**RdP**) son una herramienta gráfica para la descripción formal de sistemas cuyos comportamientos dinámicos están caracterizados por la concurrencia, sincronización, exclusión mutua y conflictos.

La RdP está representada por un grafo dirigido bipartito en el cual los lugares se representan como círculos y las transiciones son dibujadas como barras o como cajas. Los lugares suelen describir estados locales del sistema, mientras que las transiciones representarían los eventos que modifican el estado 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, entonces decimos que la transición está sensibilizada, tras dispararse la transición eliminamos una marca de cada uno de los lugares de entrada y generamos una marca en sus *lugares de salida*.

Una de las evoluciones que se han producido de las RdP es la introducción del concepto del tiempo, por ejemplo a través de transiciones con tiempo, es decir, que ahora tenemos dos tipos de transiciones las inmediatas y las temporizadas. Una RdP con estas características la denominaremos estocástica o **SPN**.

Al introducir transiciones con tiempo también será necesario añadir prioridades a las transiciones, esto evita situaciones conflictivas dentro de la red [1], aquí sólo comentaremos que las prioridades serán números naturales asociados a las transiciones y que la transición con mayor prioridad se disparará primero si hay posibilidad de que otra también se pueda disparar.

Las **GSPN**'s serán SPN's (con prioridades) en las cuales las transiciones con tiempo tienen definido el retraso de su disparo como una variable aleatoria exponencialmente distribuída.

2.4.1. Operador composición

Para realizar la traducción automática de UML a GSPN's será necesario un operador que una varias GSPN's en una única red, para ello necesitaremos añadir etiquetas tanto a las transiciones como a los lugares de cada una de las redes que queremos fusionar, este nuevo tipo de redes las llamaremos GSPN etiquetadas o bien **LGSPN**'s.

El resultado de componer lugares y transiciones de dos LGSPN's será algo como lo siguiente:

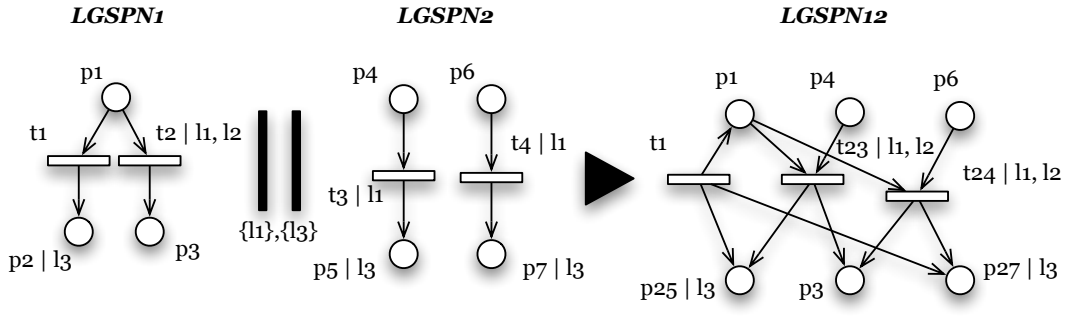

Figura 2.4: Funcionamiento del operador composición.

A continuación vamos a explicar con un poco más en detalle una de las etiquetas que forman parte de la figura 2.4, en concreto, `t2|l1,l2`, ésta consta de varias partes, la primera constituye el nombre de la transición, `t2` en este caso, lo siguiente es el separador, `|` , y tras éste nos encontraremos con una secuencia de etiquetas que utilizaremos para la **composición**, `l1` y `l2`.

2.5. ArgoSPE

ArgoSPE es una herramienta fruto de varios proyectos fin de carrera, que ya han sido comentados con anterioridad, además de varias becas de colaboración. Ha sido desarrollada bajo la licencia pública de GNU [12] (anexo E).

Esta aplicación tiene como fin poder evaluar las prestaciones de un sistema modelado con ArgoUML (por ahora[1]), implementa muchas de las características ofrecidas en los trabajos [5, 16, 18] y sigue la arquitectura sugerida en el UML-SPT, representada en la figura 2.5.

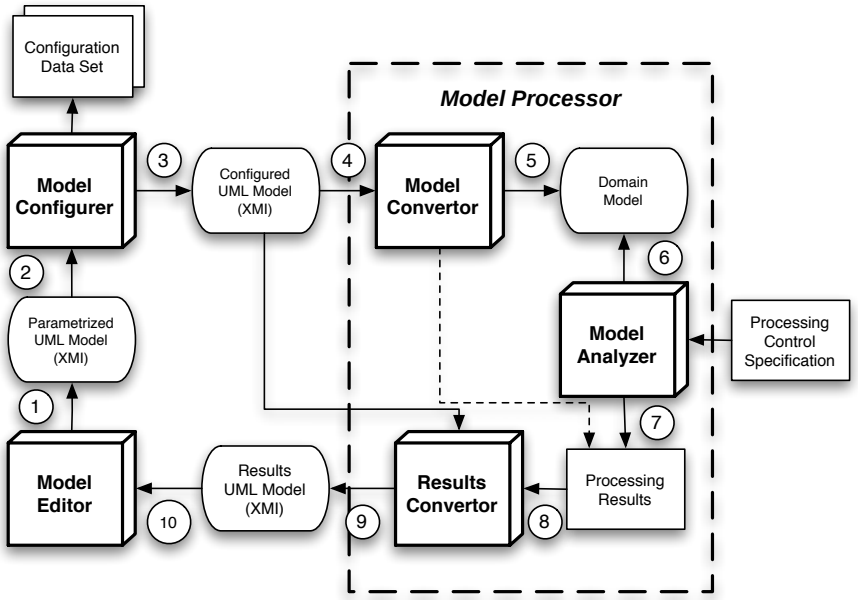

Figura 2.5: Arquitectura sugerida por el UML-SPT.

Para finalizar comentar que está diseñada como un conjunto de paquetes de Java y es implementada como un plug-in (o módulo) de ArgoUML, para la versión **0.18.1**.

Como queda reflejado en la figura 2.5 las principales partes de la arquitectura son el **editor**, el **configurador** y el **procesador del modelo**, este último está dividido en **conversor** y **analizador del modelo** y en el **conversor de resultados**.

Esta separación funcional de la arquitectura provoca que se pueda utilizar cualquier herramienta para que aporte la funcionalidad concreta, por ejemplo, se podrían utilizar diversos programas como analizadores del modelo, aunque para ello deberíamos tener

[1]Futuras implementaciones de nuestro módulo podrán utilizar cualquier herramienta CASE (Computer Assisted Software Engineering) como Editor del Modelo, dado que lo permite esta arquitectura flexible.

que representar el modelo a analizar en un formato estándar y comprensible por dichas herramientas.

Incluso podríamos considerar el cambio de formalismo utilizado para el análisis, lo que da idea de la potencia de la arquitectura de ArgoSPE.

En la siguiente figura podemos observar cómo, la estructura de paquetes que constituyen parte de los fuentes de nuestro módulo, representa la arquitectura propuesta en el UML-SPT.

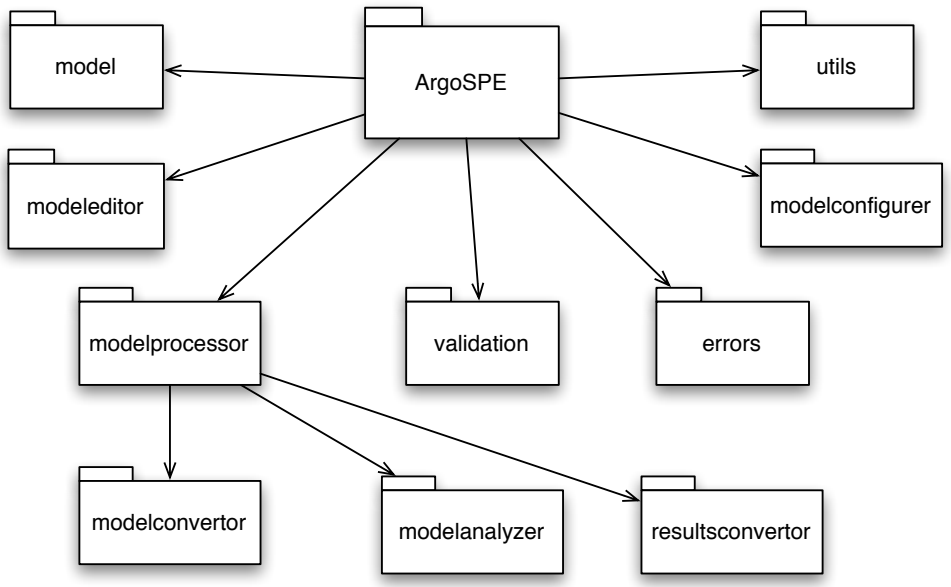

Figura 2.6: Distribución en paquetes de la arquitectura de ArgoSPE.

Las funcionalidades de los principales componentes que conforman la arquitectura son [13]:

- **Editor del Modelo**: este módulo debe ofrecer la posibilidad de crear, editar y anotar, usando el **T**agged **V**alue **L**anguage (**TVL** o Lenguaje de Valores Etiquetados), modelos software en UML.
- **Configurador del Modelo**: su funcionalidad consiste en sustituir los valores etiquetados que poseen expresiones en TVL, por el resultado de evaluar las expresiones, gracias a un fichero de configuración. Pasando a tener un modelo con el XMI configurado.
- El **Procesador del Modelo** como vemos en la figura 2.6 está dividido en:
  - **Conversor del Modelo**: es la parte encargada de pasar de un modelo en formato XMI al mismo modelo representado con GSPN's.
  - **Analizador del Modelo**: este componente ejecuta las consultas realizadas por el usuario utilizando para ello la aplicación GreatSPN.
  - **Conversor de resultados**: su objetivo es hacer regresar los valores obtenidos por el analizador, al editor del modelo, para que puedan ser recibidos por el usuario.

Capítulo 3

Planificación del trabajo

3.1. Diagrama de actividades

En mi opinión este proyecto no cuadra con el esquema de un ciclo de vida convencional, esto se debe a que nuestro trabajo consiste en la extensión de una herramienta ya existente, como es ArgoSPE. Se ha considerado más adecuado describir el modo en que se ha desarrollado mi trabajo gracias a un esquema que tiene mucha similitud con un ciclo de vida en cascada mejorado. La siguiente figura representa el orden cronológico de las fases (ver 1.6 y 4.2) en las se organizó mi proyecto desde el primer momento.

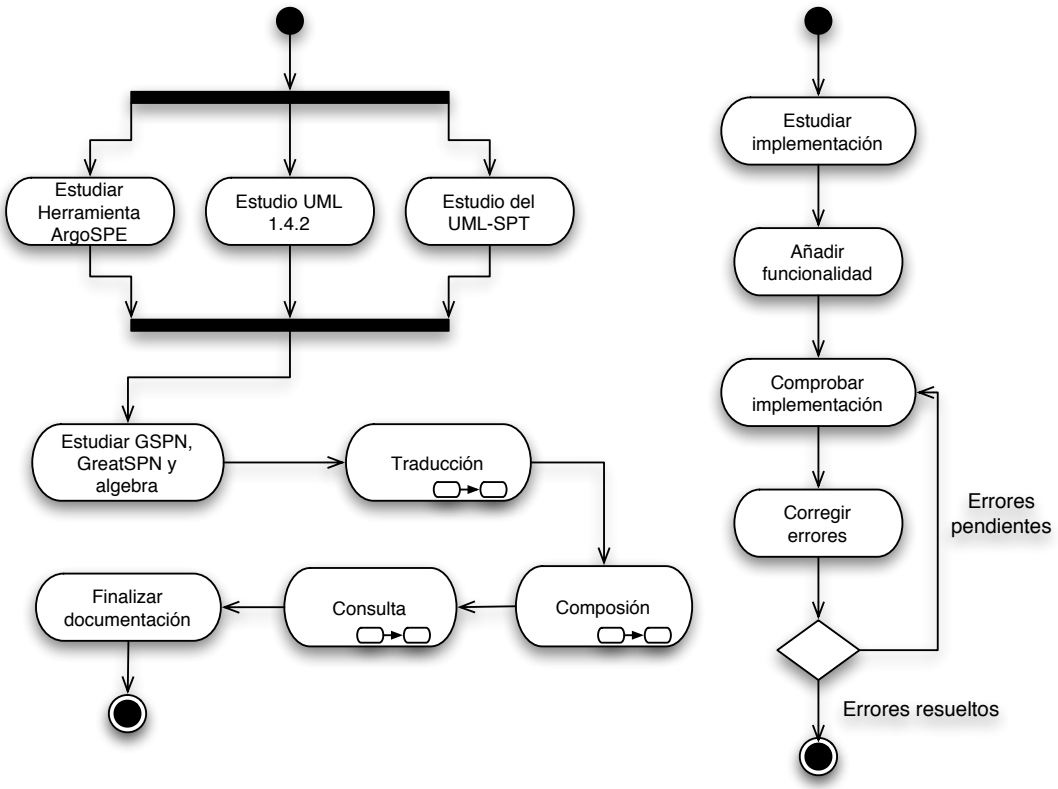

Figura 3.1: Diagramas de actividades de la planificación.

3.2. Diagrama de Gantt

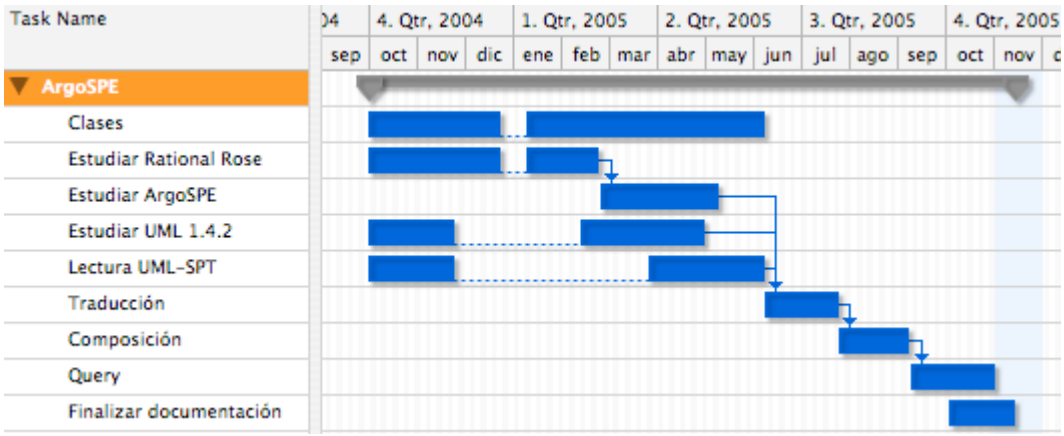

Figura 3.2: Diagrama de la cronología real.

En el diagrama anterior queda reflejado (tarea Clases) el hecho de que durante la realización de este proyecto he tenido que terminar las últimas asignaturas de mi carrera, factor que ha supuesto la prolongación, más allá de lo esperado, de algunas tareas relacionadas con mi trabajo.

Otro hecho muy importante que ha marcado el transcurso de mi trabajo fue **estudio de Rational Rose**[1] [25], que consisitió en la búsqueda y lectura de toda la información disponible referente a la extensión de esta aplicación.

La investigación realizada concluyó con, la implementación de una pequeña extensión, en forma de submenú, y con la idea clara de que era **imposible**[2] llegar a desarrollar una extensión completa, a menos que fuéramos socios tecnológicos de la empresa[3] que se dedicaba a la implementación de la herramienta, puesto que la documentación necesaria para ello solamente era distribuída a estos últimos.

El resultado de la investigación hizo que el objetivo de mi proyecto fuera completamente diferente al que en un principio se me había propuesto, a partir de ahora nuestro trabajo iba a consistir en la ampliación de la herramienta ArgoSPE.

A parte de hacer inservible la gran parte del trabajo elaborado en la primera fase, estos cambios produjeron una reestructuración completa de las tareas que se debían realizar para conseguir los nuevos objetivos, estas nuevas tareas son las que aparecen en el diagrama anterior a partir de Febrero de 2005.

El gráfico anterior pone de manifiesto las dependencias existentes entre las tareas en las cuales se ha organizado el trabajo, así se puede observar que el estudio de la teoría precede a la implementación de cada una de las partes. También observamos que, existe una dependencia entre la traducción, la composición y la consulta, la cual ha sido respetada.

[1] El objetivo original del proyecto era modificar ArgoSPE para que trabajara con Rational Rose.

[2] Este término refleja el hecho de que no disponer de la documentación adecuada prolongaría en exceso una posible extensión, haciendo inviable esa opción.

[3] Rational, la empresa desarrolladora, fue adquirida por IBM.

Capítulo 4

Resultados y conclusiones

4.1. Dificultades encontradas

Cumplir los objetivos marcados al inicio de este trabajo no ha sido un tarea precisamente sencilla. En esta sección se hará una breve descripción de las dificultades y problemas que nos hemos encontrado y que han sido superados.

Este proyecto utiliza y amplía una herramienta ya implementada, ArgoSPE. Durante mi período de formación en la carrera se nos ha inculcado la creencia de que la **comprensión de código ajeno**, es una de las tareas más complicadas con las que nos podemos enfrentar, de ahí que ayude enormemente una buena documentación del mismo. En nuestro caso nos hemos encontrado con partes que no han sido adecuadamente documentadas, lo que ha llevado a realizar una gran inversión de tiempo para comprender dicha implementación.

El primer obstáculo que hemos tenido que salvar fue la comprensión de una **parte teórica realmente compleja**. Para empezar tuvimos que estudiar el estándar UML 1.4.2[1] que nos ayudaría a poder modelar los ejemplos que más tarde utilizaríamos para la comprobación del funcionamiento de nuestra aportación.

Más adelante adquirimos los conceptos necesarios para modelar las prestaciones que deberíamos anotar en nuestros modelos, gracias al UML-SPT[2], documento que encierra cantidad de conceptos con un alto grado de abstracción.

Por último, y no por ello lo más simple, entender la teoría que rige la traducción de diagramas UML al dominio de las LGSPN's conlleva un gran esfuerzo, basta decir que es el trabajo investigado en la tesis de mi director [17], y que además ha suscitado numerosos artículos de investigación.

Aprender a utilizar la herramienta ArgoSPE no ha resultado algo trivial, el motivo principal es la falta de documentación en relación a su uso, principalmente en el apartado de la anotación de los diagramas UML, tema indispensable cuando tratamos de comprobar el funcionamiento del programa.

La naturaleza de las ampliaciones que he tenido que realizar dentro de ArgoSPE me han obligado a meterme de lleno con cada una de las partes de la arquitectura que compone la aplicación, lo cual me ha supuesto observar con detalle la mayoría del código implementado en cada uno de los **167 ficheros fuente** que constituyen nuestra

[1] El documento de sus especificaciones técnicas cuenta con 736 páginas de extensión.
[2] Este documento abarca un total de 232 páginas.

herramienta.

Unido a esto ha sido necesario realizar modificaciones sobre partes del código que ya estaban implementadas, estas modificaciones han sido motivadas fundamentalmente por dos razones, la primera ha sido la corrección de algunos errores en la traducción de las máquinas de estado, errores que serán explicados en los anexos con detalle. Y la segunda fue la transformación de la traducción de los diagramas de estado para posibilitar la composición entre las RdP's resultantes. Como es comprensible esto ha llevado al estudio tanto teórico como práctico del proceso de traducción de estos diagramas.

4.2. Trabajo realizado

Para lograr alcanzar los objetivos marcados al principio de este trabajo (sección 1.4) hemos tenido que realizar una dura labor implementando tanto la traducción, como la composición, sin olvidarnos de la consulta a nivel UML. Los resultados obtenidos son explicados a continuación.

4.2.1. Traducción

Como hemos comentado con anterioridad la traducción consiste en el paso de un modelo en UML, en este caso, de los diagramas de colaboración, al dominio de las LGSPN's. Vamos a representar en un dibujo la relación que existe entre los dos modelos:

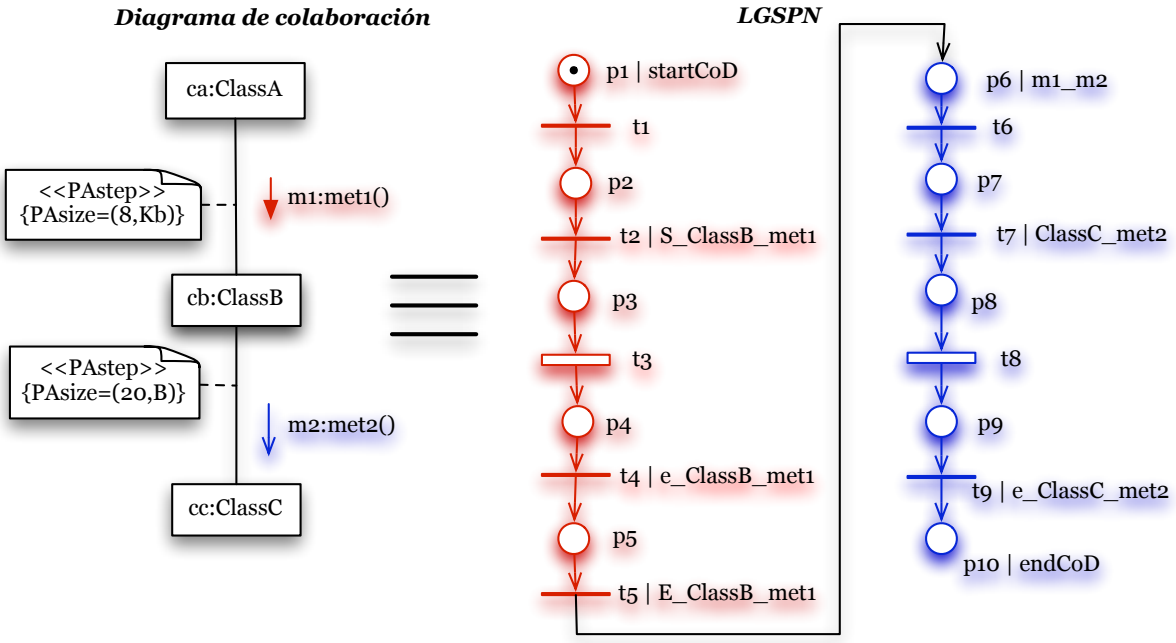

Figura 4.1: Representación del resultado del proceso de traducción.

El proceso de traducción de un diagrama de colaboración, se basa en la traducción por separado de cada uno de los mensajes que componen la colaboración.

Esta traducción será diferente dependiendo de los elementos que caractericen a ese mensaje, cada mensaje generará una pequeña red que deberá ser fusionada con la RdP

generada por el mensaje anterior, una vez finalizada la composición de las RdP's asociadas a los mensajes habremos obtenido como resultado una RdP que representará el comportamiento modelado por nuestro diagrama.

La información necesaria para determinar la traducción de un mensaje será recibida a través de un fichero XMI, que es el formato en el que se almacenará la información de nuestro modelo, de aquí tomaremos si el mensaje es síncrono o asíncrono, si ha sido anotado especificando su tamaño o no, así como otras cuestiones de menor importancia.

En la figura 4.1 observamos que el mensaje *m1* (coloreado en **rojo**) tiene una traducción diferente (RdP **roja**) al mensaje *m2* (en **azul**, como su RdP), el primero es síncrono, (lo cual viene reflejado por la representación de su flecha, los mensajes síncronos se representan por una flecha con la punta rellena) y está etiquetado con un tamaño, el segundo es asíncrono, (por lo que está representado por una flecha con punta sin rellenar).

Este esquema representa la salida del traductor que he implementado ante ese diagrama de colaboración, además tenemos que señalar que el traductor soporta cualquier elemento que pueda ser modelado dentro de un diagrama de colaboración por ArgoUML, y ha sido integrado con éxito formando ya parte de ArgoSPE. Con lo que consideramos como cumplido el objetivo de conseguir la traducción de estos diagramas.

4.2.2. Composición

El proceso de composición es difícil de explicar sin ayudarnos de complicadas fórmulas, por lo que vamos a mostrar a continuación un esquema que muestra el resultado final para intentar facilitar la comprensión del mismo:

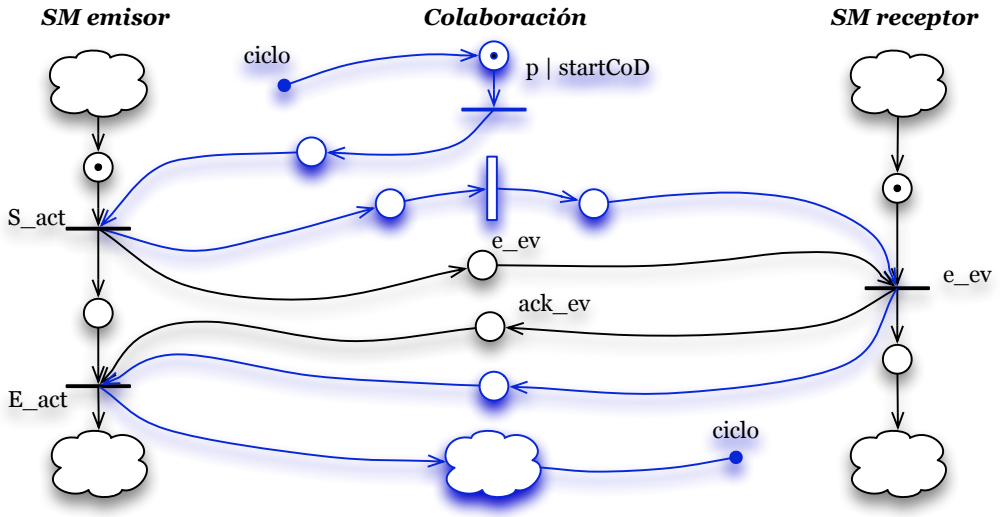

Figura 4.2: Composición entre máquinas de estado y diagramas de colaboración.

Al observar el gráfico anterior con atención tenemos que fijarnos que existen varios elementos diferenciados, en la parte izquierda tenemos la RdP obtenida de la traducción de la máquina de estados de la clase que ha generado el mensaje, el cual deberá estar representado en el diagrama de colaboración, en la parte derecha estará la RdP de la máquina de estados de la clase que recibe el mensaje. Por último en la parte central aparece un fragmento de la RdP generada al traducir el diagrama de colaboración.

En relación con el dibujo tenemos que destacar que se ha considerado oportuno resaltar, coloreándola de **azul**, la RdP generada por el mensaje (del diagrama de colaboración), con el fin de facilitar la comprensión del esquema por parte del lector. Con esta medida también conseguimos aislar el resultado producido por la composición de las RdP de las máquinas de estado, las cuales, como se puede observar tienen en los eventos de igual nombre su nexo de unión. Este es el mismo mecanismo utilizado en el gráfico de la traducción para diferenciar la RdP generada por cada uno de los mensajes.

Para entender el porqué de este proceso, tenemos que explicar una serie de cuestiones, lo primero de todo es que las clases son los elementos que definen la estructura de nuestro diseño, la máquina de estados de cada una de ellas modela la dinámica seguida por una instancia suya, por lo tanto el comportamiento del sistema queda descrito gracias a todas las máquinas de estados de las clases que participan en nuestro modelo.

Hasta ahora nuestra herramienta lo que hacía era traducir las máquinas de estados de un sistema y fusionarlas, con lo cual se obtenía un RdP gigantesca que caracterizaba el comportamiento global de nuestro sistema.

Los diagramas de colaboración reflejan las interacciones que realizan unas determinadas clases con otras definiendo un escenario concreto para nuestro diseño, es como si sacaramos una foto de como trabajan las clases para llevar a cabo una tarea.

Entonces al traducir el diagrama de colaboración a RdP y fusionar esta red con la RdP completa de nuestro sistema, obtenemos una red que describirá el comportamiento del sistema completo en una determinada situación (o escenario), que viene a ser la definida por nuestro diagrama.

Para lograr una perfecta composición entre las RdP anteriores, se tuvieron que realizar una serie de modificaciones en la traducción de las máquinas de estados, debido a que las etiquetas generadas por las RdP, tanto de las referidas a los diagramas de estados como de las del diagrama de colaboración, no eran iguales, condición sin la cual no se puede realizar la composición. Además contábamos con problema añadido de que GreatSPN tiene una longitud relativamente pequeña para almacenar las etiquetas, lo que nos obligó a definir un formato que fuera lo suficientemente pequeño y que además fuera único para cada uno de los elementos que representara.

Una vez subsanados todos los problemas simplemente tenemos que pasar las RdP generadas a **algebra**, (paquete que acompaña al programa GreatSPN y aporta la composición de GSPN's y la elminación de etiquetas dentro de las redes), el cual será el encargado final de realizar la composición. Este programa funciona en línea comandos y la llamada sería algo parecido a esto:

```
rdp99:~/ algebra -no_ba net1 net2 t eti_file net12 1
```

El primer parámetro *–no_ba* evita que los arcos sin destino o sin origen (en definitiva, rotos) sean dibujados en la red final. El fichero de etiquetas almacenará todas las etiquetas de los elementos que tendrán que fusionarse. Debemos mencionar además que la **t** que aparece como parámetro, se refiere a que la composición será meramente de transiciones ya que es lo único que se precisa en nuestro caso.

El último parámetro que acompaña a la herramienta es un número, en nuestro caso el 1, que será utilizado para indicar la colocación final de las RdP dentro del fichero resultado, el 1 significa que se colocarán horizontalmente, de izquierda a derecha, y si hubiéramos utilizado el 2 la ordenación hubiera sido vertical.

Como es lógico *net1* y *net2* representarán los ficheros en los que se encuentran las

RdP's que queremos fusionar, y *net12* será el nombre del fichero en el que queremos que se almacene el resultado de la composición.

Una última cuestión que no se ha mencionado anteriormente es el hecho de que ha resultado especialmente difícil la implementación, tanto de la traducción como de la composición, ya que debíamos de mantener intacta la funcionalidad que ya era proporcionada por la herrramienta, así que las modificaciones requiridas en el código existente se han realizado como comúnmente se suele decir, *con pies de plomo*.

4.2.3. Consulta

Todo lo implementado anteriormente quedaría sin sentido si no proporcionásemos algún modo de sacar partido de ello.

Pensando detenidamente, vemos que el objetivo de anotar nuestro modelo con características que definan las prestaciones de ciertos elementos, es el obtener una información que sea útil durante la fase de modelado a la persona que está realizando el diseño del sistema.

Una de las cuestiones más interesantes que podemos preguntarnos cuando estamos diseñando un sistema sería, ¿cuánto tiempo nos cuesta ejecutar una determinada tarea? Pues bien esa es la pregunta que podemos responder gracias al proceso de traducción y composición.

En nuestro caso vamos a denominar a la consulta que añadimos a ArgoSPE, **Response Time** y vendrá definida como el tiempo medio de respuesta de un escenario concreto, la consulta se podría decir que calcula el tiempo de respuesta del diagrama de colaboración que define dicho escenario.

Para poder calcular esta característica en nuestro sistema fue preciso realizar unas pequeñas transformaciones a la RdP que generábamos en el proceso de traducción, éstas implican crear una transición etiquetada como *close*, que hará de puente entre el lugar final de la RdP del diagrama de colaboración y su lugar inicial, obteniendo de esta manera una red cerrada necesaria para el análisis en el estado estacionario.

4.2.4. Pruebas

Al finalizar cada una de las etapas de implementación se pasaba a un período de comprobación, este proceso era de una gran importancia debido a que todas las partes dependían las unas de las otras para su correcto funcionamiento.

Para comprobar el buen comportamiento del código creado para realizar la traducción se tenía que modelar ejemplos de sistemas, en ArgoUML, que cumplieran los requisitos necesarios para poder generar una RdP.

Estos requisitos pasaban por diseñar, al menos dos clases, cada una con sus respectivos diagramas de estados, y en las que teníamos que enviar y recibir al menos un evento, lo que quiere decir que, en una de las máquinas de estados una acción sería la llamada de uno de los métodos de la otra clase y en la máquina de estados de la clase llamada recogeríamos la llamada, gracias al modelado del evento disparador (o trigger). Esta misma interacción era representada a su vez con un diagrama de colaboración. Una vez ideado el sistema teníamos que anotar el diseño con las anotaciones conforme a la tabla D.2.

Con todo esto ya podríamos generar una RdP para nuestro diagrama de colaboración

con nuestro traductor, la cual debería ser comparada con una RdP que habíamos elaborado a mano previamente, con la estructura correcta.

Este mismo proceso también es seguido para la comprobación del adecuado funcionamiento del código de la composición, aunque en este caso se hace mucho más complicada la verificación debido al gran número de lugares y transiciones existentes en la red, recordemos que estamos formando una única red a partir de todas las RdP de las máquinas de estado y de la del diagrama de colaboración seleccionado.

Con todo lo dicho queda claro que la realización de pruebas es una tarea bastante complicada.

4.3. Valoración del trabajo

4.3.1. Aplicación del trabajo desarrollado

Uno de los logros más significativos de este proyecto es la contribución realizada a una aplicación, ArgoSPE, que tiene como objetivo mejorar el proceso de desarrollo del software, introduciendo los criterios de prestaciones desde la fases iniciales del ciclo de vida del sistema, con el fin de ahorrar tiempo evitando diseños que no cumplirán los requisitos que nos hemos marcado.

No nos podemos olvidar que al mejorar ArgoSPE, también estamos posibilitando el aumento las características ofrecidas por ArgoUML, cuya versión 0.18.1 fue bajada 98.143 veces (datos hasta Agosto de 2005), todos estos usuarios al fin y al cabo son usuarios potenciales de nuestro módulo.

Es más, a pesar de sus deficiencias y de que aún no existe una versión totalmente estable, ArgoUML es la herramienta CASE, gratuita y de código libre, más utilizada en todo el mundo.

4.3.2. Trabajo futuro

Una de las ventajas de haber trabajado en la mayoría de las componentes que constituyen la aplicación es que se tiene una visión global de la funcionalidad que ofrece y de las posibles mejoras que podrían llegar a realizarse.

En mi opinión sugerir nuevas funcionalidades para una aplicación sin conocer, de una manera precisa, el estado de las funcionalidades que ahora mismo aporta no es algo razonable. Por este motivo la primera ampliación que propondría sería la realización de un estudio del proceso de traducción, que refleje las situaciones que son tenidas en cuenta y las que no soporta.

Una vez realizado ese estudio, enfocaría los esfuerzos en subsanar todos los defectos encontrados en la traducción, así como la implementación de cada nuevo aspecto que se han tenido en cuenta en el trabajo [6], ya que la considero un punto clave sobre el que se fundamenta el buen funcionamiento de nuestra herramienta.

Basándonos ahora en el enfoque de ampliar la funcionalidad de ArgoSPE, cabe señalar la realización de un estudio de herramientas CASE que puedan ser utilizadas con nuestro módulo. El objeto de cambiar ArgoUML por otra aplicación no es otro que proporcionar un mayor abanico de diagramas con los que pueda trabajar nuestra herramienta, a la vez que contar con un mayor número de elementos implementados dentro de cada tipo de diagrama.

Otra ampliación que resultaría de gran utilidad sería sustituir el formato de las RdP que genera el traductor, ya que actualmente las RdP se generan con formato GreatSPN, formato que no es estándar, por lo que limita la herramienta con la que podemos analizar las redes obtenidas. El mejor candidato como formato estándar de RdP es **PNML** (**P**etri **N**et **M**arkup **L**anguage) [23].

Por supuesto la funcionalidad que ArgoSPE ofrece, viene dada por las consultas que proporcione sobre los modelos de nuestros sistemas, por eso consideramos una buena opción añadir más consultas a las ya implementadas por nuestra aplicación.

Para finalizar me gustaría proponer la investigación de las características menos documentadas de esta aplicación, lo cual implicaría la lectura detallada de todos los documentos relacionados con ella que existen, como artículos memorias de proyectos fin de carrera y demás y tratar de aclarar todo aquello que sea susceptible de ser necesitado por futuros desarrolladores.

4.4. Conclusiones personales

Como resultado de este proyecto la principal conclusión que se puede extraer es lo valioso que resultan la experiencia y los conocimientos adquiridos a lo largo de su desarrollo.

Por un lado he conseguido incrementar ampliamente mis conocimientos de Java, uno de los lenguajes más extendidos en la actualidad. Además he asentado y aumentado mi entendimiento sobre un estándar tan extendido e importante como es UML. Por otra parte también he sido capaz poner en práctica algunos de los conocimientos adquiridos a lo largo de mi período de formación.

Por otro lado he aprendido a utilizar con soltura herramientas básicas en el desarrollo de proyectos como CVS, Ant (para desarrollo de Java) y otras herramientas proporcionadas por el sistema operativo GNU/Linux, además del perfeccionamiento en el manejo del editor de textos Vim.

La envergadura de este tipo de trabajos me ha aportado también soltura a la hora de planificar, documentar y gestionar las diferentes etapas que lo componían.

Un concepto importante que he aprendido es que una adecuada organización inicial, una correcta planificación y unos conocimientos previos bien asentados (no necesarios pero desde luego deseables) son elementos muy a tener en cuenta ya que facilitan en gran medida el trabajo desarrollado a posteriori y ahorran gran cantidad de tiempo y esfuerzo.

Gracias a la observación de otros proyectos como ArgoUML se han adquirido nociones de cómo se organiza y planifica un proyecto software, en el cual colaboran decenas de personas (siempre dentro del marco de un proyecto de código libre). El hecho de que toda la gestión del proyecto se realice vía web le da aún más valor a este asunto.

En último lugar me gustaría expresar que estoy profundamente satisfecho con el resultado de este proyecto y espero que los conocimientos y mecanismos aprendidos durante su desarrollo me resulten muy útiles en un, esperemos próximo, entorno laboral.

Parte II

Anexos

Apéndice A

Análisis de UML

A.1. Diagramas de estados

Un **diagrama de estados** se utiliza para modelar aspectos dinámicos de un sistema. La mayoría de las veces esto supone el modelado del comportamiento de objetos reactivos. Un **objeto reactivo** es aquél para el que la mejor forma de caracterizar su comportamiento es señalar cuál es su respuesta a los eventos lanzados desde fuera de su contexto.

Los diagramas de estados pueden asociarse a las clases, los casos de uso, o a sistemas completos para visualizar, especificar, construir y documentar la dinámica de un objeto individual.

Supongamos que hemos modelado una clase que representa un sistema de registro (login) de usuarios. El comportamiento dinámico de cada objeto que instancia dicha clase se modela con el ejemplo de la figura A.1, que además nos sirve como ejemplo de la notación gráfica empleada en un diagrama de estados:

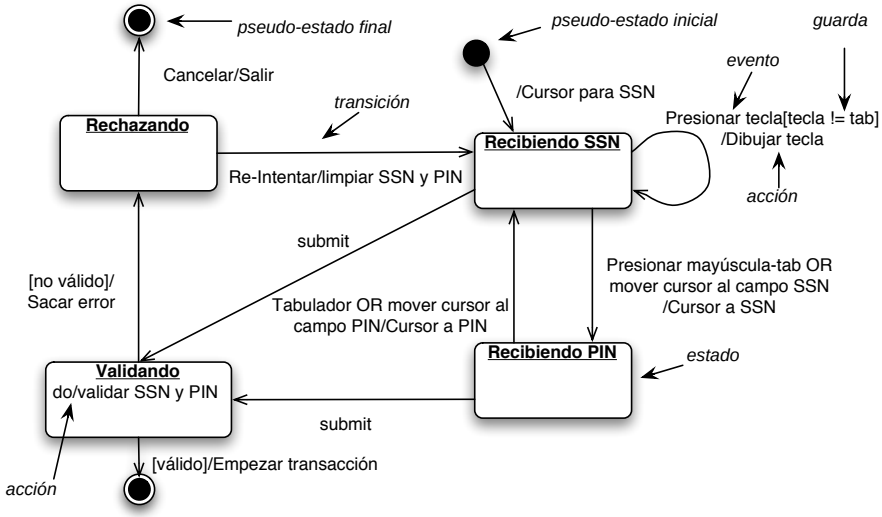

Figura A.1: Ejemplo de diagrama de estados.

También en este ejemplo vemos algunos de los elementos más representativos, las relaciones que se establecen entre ellos y varios adornos. Los elementos más comunes

dentro de un diagrama de estados son los estados y las transiciones.

Un **estado** es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Cuando la máquina de estados de un objeto se encuentra en un estado dado, se dice que el objeto está en ese estado y que ese estado está **activo**.

Un estado tiene varias partes: un nombre (cadena de texto que distingue al estado de otros estados), **acciones de entrada** (acciones ejecutadas al entrar en el estado e identificadas con la palabra clave *entry*), **acciones de salida** (acciones ejecutadas al salir de un estado e identificadas con la palabra clave *exit*), **transiciones internas** (transiciones que se manejan sin causar un cambio en el estado), **actividades** (actividades que son ejecutadas mientras se encuentra activo el estado que las contiene y que se identifican con la palabra clave *do*) y **eventos diferidos** (lista de eventos que no se manejan en este estado sino que se posponen y se añaden a una cola para ser manejados por el objeto en otro estado).

Además de los estados simples podemos encontrar otros tipos de estados como los estados compuestos, los estados **de sincronización**, los estados de submáquina, los pseudoestados, etc.

En la figura del ejemplo vemos dos tipos de estado más muy comunes que son el **estado inicial**, que indica el punto de comienzo por defecto para una máquina de estados o el subestado, y el **estado final**, que indica que la ejecución de la máquina de estados o del estado que lo contiene ha finalizado.

Los estados **compuestos** son también elementos comunes de los diagramas de estados. Mientras que un estado simple no tiene una subestructura, un estado compuesto es capaz de contener a otros estados. Los estados anidados dentro de un estado compuesto reciben el nombre de subestados. Cuando un subestado está activo, también lo está el estado compuesto que lo contiene. Además, los subestados se pueden anidar a cualquier nivel.

Una **transición** es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones específicas. Cuando se produce ese cambio de estado se dice que la transición se ha disparado. Una transición tiene cinco partes:

- **Estado origen**: El estado afectado por la transición; si un objeto está en el estado origen, una transición de salida puede dispararse cuando el objeto reciba el evento de disparo de la transición, y si la condición de guarda, si la hay, se satisface.
- **Evento de disparo**: el evento cuya recepción por el objeto que está en el estado origen provoca el disparo de la transición si se satisface su condición de guarda.
- **Condición de guarda**: Una expresión booleana que se evalúa cuando la transición se activa por la recepción del evento de disparo; si la expresión toma el valor cierto, la transición se puede disparar; si la expresión toma el valor falso, la transición no se dispara y si no hay otra transición que pueda ser disparada por el mismo evento, éste se pierde.
- **Acción**: Una computación atómica ejecutable que puede actuar directamente sobre el objeto asociado a la máquina de estados, e indirectamente sobre otros objetos visibles al objeto.
- **Estado destino**: El estado activo tras completarse la transición.

A.2. Diagramas de colaboración

Un **diagrama de colaboración** es uno de los dos tipos de diagrama de interacción que podemos encontrar, éstos últimos están dedicados a describir cómo colaboran un grupo de objetos para conseguir la realización de una tarea. En ellos se muestra los mensajes intercambiados entre los objetos dentro de un caso de uso, recogiendo de esta manera el comportamiento del mismo, o de un diagrama de clases.

Los dos tipos de diagramas de interacción muestran la colaboración entre los objetos aunque cada uno hace énfasis en una cualidad diferente de esta relación, así por ejemplo los **diagramas de secuencia**, que son los otros diagramas pertenecientes a este grupo, remarcan la ordenación temporal de los mensajes, mostrando la secuencia de mensajes intercambiados a lo largo de la línea de vida de cada uno de los objetos participantes.

Por el contrario los diagramas de colaboración enfatizan la organización estructural de los objetos que componen la interacción, en detrimento de la representación del orden temporal de los mensajes, no obstante, se puede conocer la ordenación de éstos simplemente fijándose en los números de secuencia que aparecen representados en el diagrama. A partir de este punto vamos a centrarnos en el diagrama de colaboración, ya que es el diagrama de interacción que es relevante para nuestro trabajo.

En el metamodelo de UML el paquete que define los elementos que utilizaremos en los diagramas de colaboración, es el Collaborations package (o paquete de Colaboraciones), éste se relaciona con otros paquetes que forman parte del metamodelo, siguiendo la estructura representada en la figura:

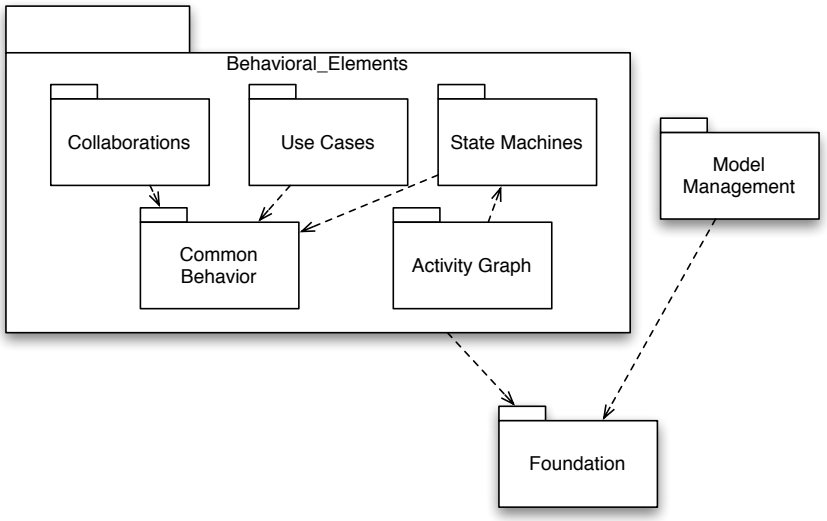

Figura A.2: Paquetes del metamodelo de UML.

El paquete **Behavioral_Elements** (Elementos de comportamiento) es una estructura que especifica el comportamiento dinámico. El paquete Collaborations, como vemos, forma parte de este super-paquete, y está dividido en dos partes: la **colaboración** y la **interacción**.

Colaboración define los roles que juegan (o los papeles representados por) un conjunto de instancias cuando realizan una tarea en particular, como una operación

o un caso de uso.

Interacción Es una colección de comunicaciones producidas entre instancias, en las que se incluyen todos los tipos de formas en las cuales podemos afectar en una instancia, como la invocación de una operación, o la creación y destrucción de éstas. Las comunicaciones están parcialemente ordenadas (en tiempo). Una interacción expresa un patrón de comunicación modelado por las instancias que representan los roles de una colaboración.

A.2.1. Interacciones

Los elementos que forman parte de una interacción son los siguientes:

- **Instancia** (objeto, valor de datos, instancia de un componente): es una entidad con una única identidad y para la cual tenemos un conjunto de operaciones que pueden ser aplicadas (o señales ser enviadas) y que tiene un estado que almacena los efectos de las operaciones.
- **Acción**: una especificación de una sentencia ejecutable. Existen diferentes tipos de acciones que están predefinidas, por ejemplo acción de creación, de llamada, de destrucción, y la acción no interpretada.
- **Estímulo**: representa una comunicación entre dos instancias. La notación utilizada para definir un estímulo es una flecha.
- **Operación**: una declaración de un servicio que puede ser consultado desde una instancia para conseguir un comportamiento.
- **Señal**: una especificación de un estímulo asíncrono comunicado entre instancias.
- **Enlace**: una conexión entre instancias.

A.2.2. Colaboración

Una colaboración, a su vez, también está constituída por un conjunto de artefactos que le proporcionan toda la expresividad que necesitamos:

- **Colaboración**: describe un clasificador como, una operación, una clase, o un caso de uso, es realizado por un conjunto de clasificadores y asociaciones utilizados de una determinada manera. Se representa con esta sintaxis:

Figura A.3: Representación de una colaboración.

- **ClassifierRole** (papel representado por el clasificador): es un rol específico que juega un participante en una colaboración. Especifica un *punto de vista restringido* de un clasificador, definido por lo que es requerido en la colaboración.

- **Mensaje**: expresa una comunicación entre instancias. Es una parte del patrón de comunicación representado por una interacción. La notación utilizada para definir un mensaje es una flecha.

- **AssociationRole** (asociación de un rol): una *association role* es un uso específico de una asociación necesaria en una colaboración. Concreta un punto de vista limitado de una asociación entre clasificadores.

- **Generalización**: es una relación taxonómica[1] entre un elemento general y otro elemento más específico. Se representa de esta manera:

Figura A.4: Representación de una generalización.

A la hora de representar tanto los mensajes como los estímulos existen diferentes tipos de flechas, las cuales describen diferentes tipos de flujo de control, entre éstas las más conocidas son:

Figura A.5: Tipos de flechas.

La primera flecha, empezando por la izquierda, representa una llamada a un procedimiento, aunque también puede ser utilizada para denotar que existen instancias activas y una de ellas envía una señal , esperando una secuencia de llamadas que complete el comportamiento antes de continuar, se distingue perfectamente porque tiene la punta rellena. La siguiente flecha denota una comunicación asíncrona, es decir, el emisor dispara el estímulo e inmediatamente continúa con el siguiente paso de la ejecución.

Y por último la flecha discontínua que refleja el hecho de regresar de la llamada a un procedimiento, como es lógico estará emparejada con una flecha del primer tipo, la flecha de retorno sería obviada cuando estemos al final de la activación de un objeto.

Las **etiquetas** que acompañan a las **flechas** que representan tanto los mensajes como los estímulos tienen un formato determinado:

```
predecesor '/' guarda nº_de_secuencia valor ':=' nombre_msj argumentos
```

Para que veamos con un poco más de claridad los diferentes tipos de comunicación que podemos tener utilizando este tipo de flechas vamos a representar unos diagramas de secuencia:

[1] Taxonomía: Ciencia que trata de los principios, métodos y fines de la clasificación.

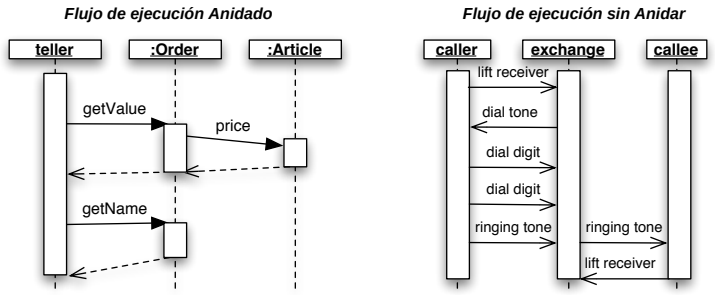

Figura A.6: Tipos de comunicación.

A.2.3. Tricotomía Clasificador-Instancia-Rol

Una de las cosas que se puede hacer más complicado a la hora de la comprensión de los diagramas de interacción, en general, es diferenciar entre los conceptos de instancia, clasificador y papel que juega un clasificador dentro de la colaboración, lo que denominamos *rol*.

Una **instancia** es una entidad con comportamiento y un estado, y además tiene un identidad única. Un **clasificador** es una descripción de una instancia. Un **rol del clasificador** define un uso (una abstracción) de una instancia. La relación existente entre estos elementos queda patente en la figura A.7.

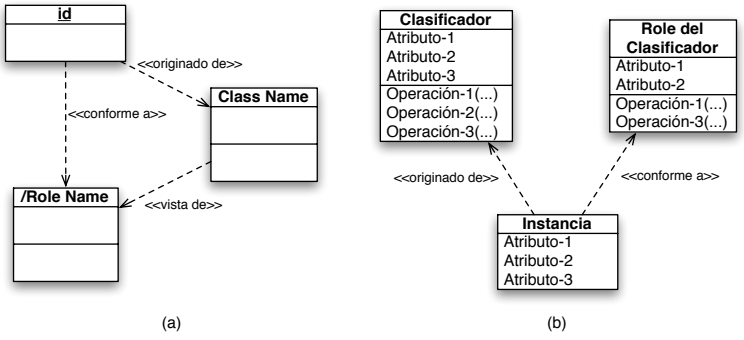

Figura A.7: Tricotomía Clasificador-Instancia-Rol.

Los atributos de una instancia corresponden a los atributos que posee su clasificador. Todos los atributos requeridos por el *rol del clasificador* tienen su atributo correspondiente en la instancia. Todas las operaciones definidas en el clasificador de la instancia pueden ser aplicadas a la instancia. Todas las operaciones requeridas por el rol del clasificador, son aplicables a la instancia.

Otra situación un tanto confusa que podemos encontrarnos es la diferencia existente entre la asociación de un rol y la asociación que relaciona los clasificadores, como se puede observar en el siguiente modelo existen sutiles diferencias entre ambas:

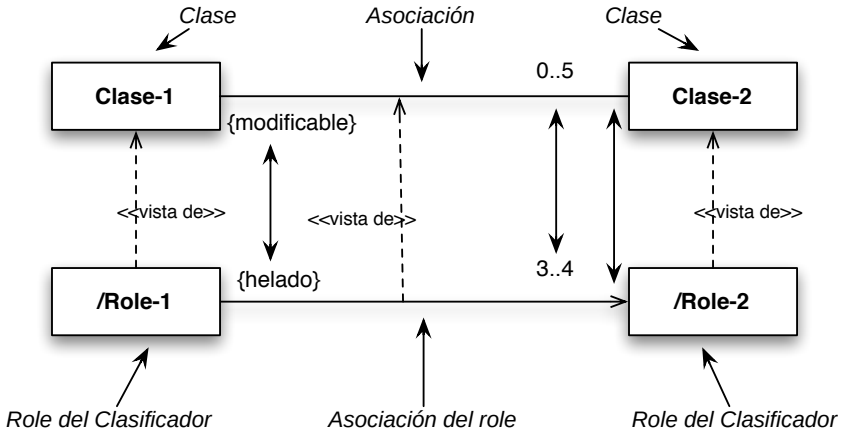

Figura A.8: Asociación y asociación de los roles.

Apéndice B

ArgoUML

B.1. Deficiencias de los diagramas de colaboración

Como hemos comentado con anterioridad ArgoUML es un programa de software libre, la mayor problemática existente con este tipo de proyectos es que la dedicación que los desarrolladores invierten en ellos, está muy determinada por su tiempo libre ya que, como norma general, estos productos no son implementados con ánimo de lucro, sino para cubrir una necesidad detectada por una cierta comunidad de usuarios, que serán los que a la postre lleven el peso de la implementación.

Con todo esto es lógico que algunos de estos proyectos manifiesten ciertas deficiencias en su implementación, ArgoUML no es una excepción a todo esto, es más, las librerías utilizadas por éste también han sido desarrolladas de la misma forma, como consecuencia de ello presenta algunas carencias que nos afectaron a la hora de la realización de nuestro trabajo.

Todas estas deficiencias se engloban en varios grupos:

- **Diagramas enteros sin implementar**: Como es el caso de los diagramas de secuencia, aunque según queda reflejado en la página de ArgoUML se espera que en la versión 0.20 vuelva a estar soportado por la herramienta.

- **Elementos incorrectamente implementados**: Normalmente esto se debe a una incorrecta interpretación de las características descritas en la especificación de UML.

- **Funcionalidades poco desarrolladas**: Ésto se ve reflejado, por ejemplo, en que la impresión de los diagramas no satisface grandes expectativas, lo mismo que ocurre con la generación de código y algún que otro fallo de programación, como excepciones no capturadas debidamente.

- **Diagramas que no contemplan varios elementos** que se encuentran descritos en el estándar UML dentro de ellos, este último grupo será el que más nos afecte a la hora de poder realizar nuestro trabajo, en concreto, en el proceso de traducción.

Es importante tener en cuenta el hecho de que ArgoUML soporta en estos momentos la versión 1.3 de UML, esta versión ya quedó obsoleta a finales de 2003, este retraso en cuanto a la evolución de UML es debido a la librería Java que proporciona a ArgoUML una implementación del metamodelo de UML: la librería **NSUML** [20].

Esta librería implementa la versión de UML con la que trabaja Argo, se encuentra inacabada con diversos fallos y en un estado de abandono en su desarrollo por parte de la empresa que inició este proyecto, la documentación que proporcionan es muy deficiente lo que hace más complicado la actualización de dicha librería.

A continuación pasaremos a citar las carencias detectadas a lo largo del desarrollo de nuestro proyecto, éstas están agrupadas en dos apartados que señalaremos a continuación:

- **Elementos no implementados en el Diagrama de Colaboración**:
  - No es posible incluir un objeto múltiple (o multiobjeto) en uno de estos diagramas. Un **multiobjeto** representa un conjunto de instancias de una determinada clase, es usado para mostrar que, ciertas operaciones y señales, van dirigidas a un conjunto, en vez de a una única instancia como ocurre habitualmente. Su representación sería algo como esto:

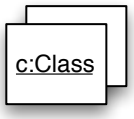

Figura B.1: Representación gráfica de un objeto múltiple.

  - ArgoUML no permite añadir un objeto activo dentro de una colaboración. Un **objeto activo** es aquel que posee el hilo de control e iniciaría la actividad de control, dicho de otra forma, será el encargado de iniciar la colaboración. Normalmente se suele representar del mismo modo que un objeto común, únicamente que aparecería resaltado en negrita como puede observarse en la figura B.2:

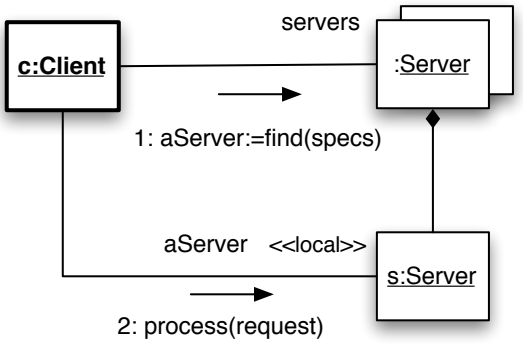

Figura B.2: Representación gráfica de un diagrama con un objeto activo.

  - No podemos crear elementos a nivel de instancia, sólo a nivel de especificación. Esto impide que utilicemos instancias (en lugar de clasificadores), al igual que ocurre con los estímulos (en vez de los mensajes), por supuesto tampoco se puede modelar enlaces entre objetos.
  - La herramienta CASE no permite representar actores dentro de un diagrama de colaboración, por lo tanto no podríamos modelar un diagrama de este estilo:

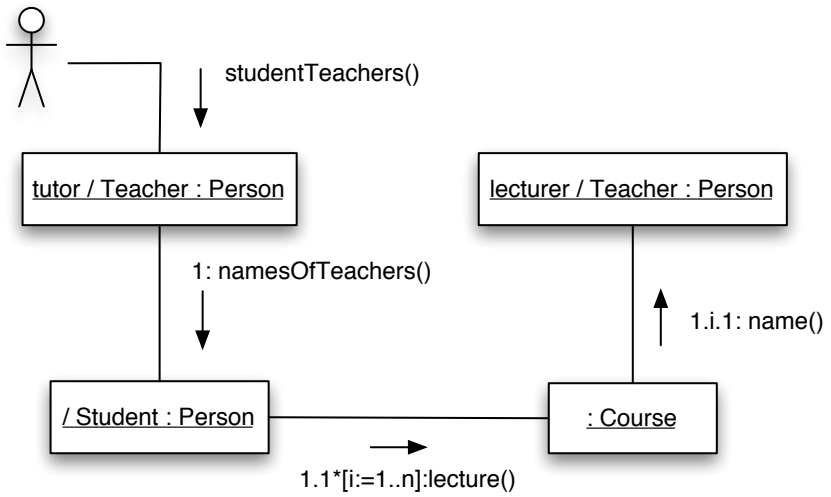

Figura B.3: Representación gráfica de un actor en un diagrama de colaboración.

- En ArgoUML no aparece el símbolo de la colaboración (ver A.3), por lo que tendremos un menor número de posibilidades de relacionar una colaboración con otros elementos de nuestro modelo, como por ejemplo con clasificadores, con instancias o con casos de uso:

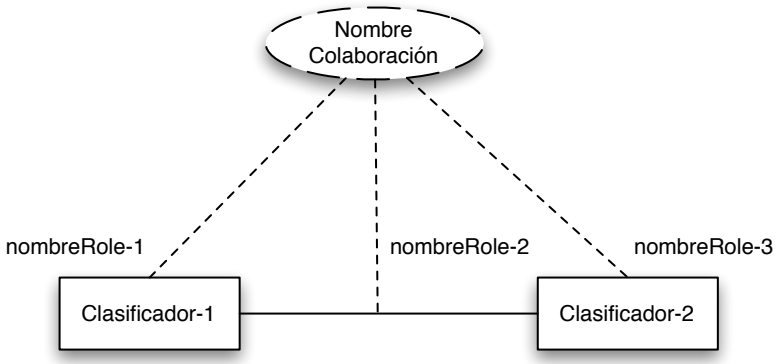

Figura B.4: Diagrama que representa una colaboración y sus clasificadores.

- Es importante señalar que la implementación del diagrama de colaboración de Argo no soporta la representación de la **ejecución** de una acción **condicionada** al cumplimiento de una situación. Al igual que tampoco ofrece la posibilidad de expresar la **ejecución** de un mensaje de forma **iterativa**. Estas dos carencias han hecho que no nos sea posible utilizar todas las notaciones que disponíamos para este diagrama.

- **Elementos implementados que presentan deficiencias**:

  - No se puede establecer una asociación entre el papel representado por un clasificador y una clase existente dentro de nuestro modelo, es decir, no podemos reflejar la relación clasificador y su rol.

- Los diagramas de colaboración no pueden ser creados sin que estén asociados a un caso de uso (más concretamente a un actor de un caso de uso) o a una clase (perteneciente a un diagrama de clases) que tengamos modelada en nuestro sistema previamente. Esta característica queda completamente fuera de la semántica que aparece en la especificación de UML, puesto que el diagrama de colaboración se encarga de representar la interacción entre objetos de diferentes clases a través de mensajes, no vemos que tenga sentido que para crearse estos diagramas tengan que estar asociados a alguna clase o a algún actor, por ejemplo en el esquema B.3, ¿a qué clase tendríamos que asociar ese diagrama de colaboración? A la clase *Person* o la clase *Course*. Según la especificación no tendríamos porque tomar esta decisión ya que no se establece una relación directa con ninguna de las dos clases, sino que el diagrama estaría al mismo nivel que el diagrama de clases o que los casos de uso.
- ArgoUML no sigue estrictamente la notación para representar los nombres de los clasificadores indicada por la especificación del estándar, puesto que el clasificador tiene un formato claramente definido:

  ```
  "nombre_del_objeto" / "nombre_del_rol" : "nombre_de_la_clase"
  ```

  En nuestro caso no es posible escribir el nombre de la clase a la cual va asociada.
- Los mensajes no pueden cambiar su orientación una vez han sido representados en una asociación entre clasificadores. Únicamente contamos con un tipo de mensaje, los mensajes síncronos (comunicación síncrona). El orden de los mensajes es generado automáticamente por la herramienta.
- Las acciones también muestran su falta de corrección en relación a lo apuntado en la especificación estudiada, éstas no pueden estar asociadas a los métodos de una clase, ni tampoco podemos pasarle argumentos a una acción.

Todas estas carencias nos llevan a la conclusión de que todavía falta bastante trabajo para que los diagramas de colaboración estén implementados de acuerdo a toda la expresividad que ofrece UML, aunque tenemos que decir que son los únicos representantes, por el momento, de los diagramas de interacción en la herramienta CASE que utilizamos.

B.2. XMI

Cuando la OMG publicó en 1997 un **RFP** (Request For Proposal) para un formato de intercambio de modelos basado en Stream [27](Stream-Based Model Interchange Format -**SMIF**) se presentaron Unisys, IBM, Oracle, y otras compañías con la propuesta de **XMI**.

Como su propio nombre indica XMI (**X**ML **M**etadata **I**nterchange) es un formato de intercambio de **metadatos** basado en **XML**. Antes de continuar, consideramos necesario realizar unos comentarios acerca de XML.

XML [31] significa e**X**tensible **M**arkup **L**anguage, es un subconjunto de **SGML** (**S**tandard **G**eneralized **M**arkup **L**anguage), la funcionalidad que convierte a XML en una herramienta tan poderosa es que es un **metalenguaje**, es decir, que es un lenguaje utilizado para definir otros lenguajes de marcado que se adecúen a un problema concreto.

Además de esta característica también podemos mencionar el hecho de que es un estándar del W3C [30], en su contenido (determinado por un **DTD** [11] o Definición del Tipo de Documentos) se combina tanto los datos como los metadatos para el intercambio de información, es flexible e independiente del sistema y las etiquetas forman una estructura en árbol (**DOM** [10] Modelo de Objetos del Documento). En la figura B.5 están representados todos los elementos que hemos mencionado.

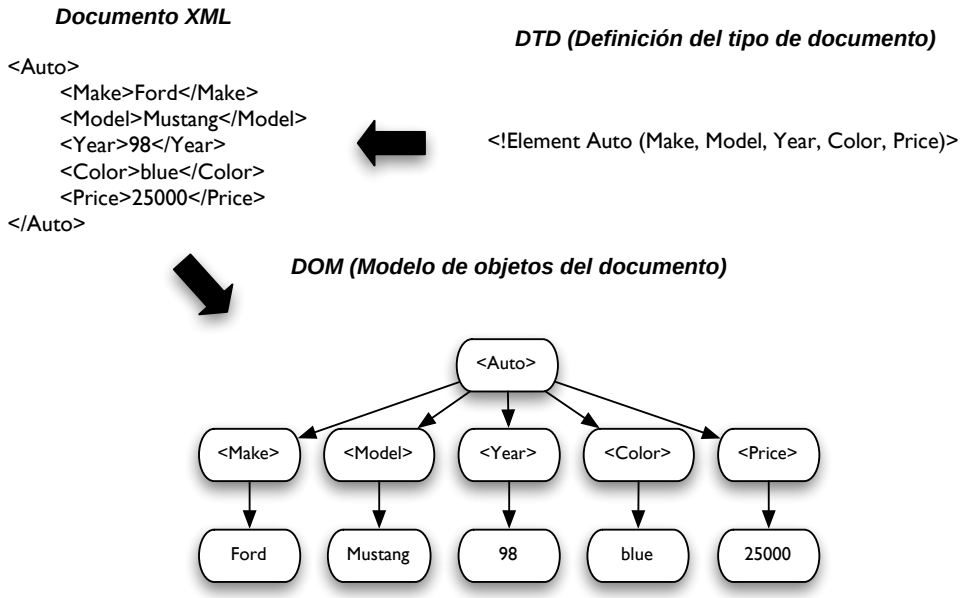

Figura B.5: Relación entre XML, DTD y DOM.

El objetivo último de la definición de un formato como XMI era la integración de herramientas aplicaciones y repositorios que trabajaban con metamodelos que eran conformes a **MOF** (**M**odel **O**bject **F**acility).

B.2.1. MOF

UML es un estándar utilizado para describir *modelos de objetos*. Desde un punto de vista arquitectónico el modelo de UML está estructurado en tres niveles:

- **Metamodelo**: El metamodelo de UML define un número de elementos tales como *Class*, *Operation*, *Attribute*, *Association*, y algunos más. Estos elementos se llaman **metaclases**.
- **Modelo**: Instancias de una *Class* de UML representan entidades, y procesos. Por ejemplo:

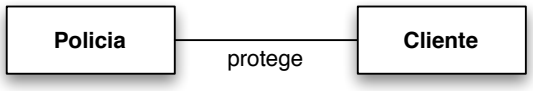

Figura B.6: Clase Policia y Cliente.

Se puede definir también instancias de la metaclase Association entre clases, tal como la asociación (*protege* ver B.6) entre Cliente y Policia. Estas instancias de metaclases de UML se denominan **metaobjetos**.

- **Objetos del usuario**: Son instancias de los elementos del modelo. Por ejemplo instancias de la clase *Cliente* son: c123:Cliente, c456:Cliente. Estos son los objetos. Los modelos de objetos de UML de dominios particulares, se denominan *modelos de objetos de dominio* o *modelos de dominio*.

Existen también otros metamodelos como son los metamodelos para sistemas de bases de datos. Un metamodelo de una base de datos define elementos básicos como base de datos, tabla, columna, clave, y demás. Que son usados para definir modelos de datos específicos.

Tantos los modelos de objetos de dominio como los modelos de datos son guardados en bases de datos especializadas llamadas **repositorios de metadatos**, algunas veces llamado simplemente como **repositorio**. Los administradores de los repositorios normalmente necesitan manejar modelos de dominios y modelos de datos de una forma unificada, pero la naturaleza tan dispar de los metamodelos sobre los cuales están basados constituye un gran obstáculo.

MOF provee una base común para esos metamodelos. Si dos metamodelos diferentes están de acuerdo con MOF, los modelos basados en ellos pueden residir en el mismo repositorio o intercambiarse por diferentes herramientas compatibles con él.

El metamodelo de UML está basado en el estándar MOF, (es decir que sus constructores están definidos en términos de los elementos *core* de MOF), y esto es una pieza clave de la estrategia de la OMG para soportar repositorios integrados. MOF define un conjunto de constructores que pueden ser usados para describir metamodelos.

MOF agrega un cuarto nivel a la clasificación que mostramos antes. MOF llama a estos niveles: M0, M1, M2, y M3.

- **M3** (MOF core): Define los elementos usados para especificar metamodelos, como por ejemplo MetaClass, MetaAttribute, MetaAssociation, Mof::Class, Mof::Association, Mof::Attribute.
- **M2** (Meta-Model): Un metamodelo definido en términos de los elementos core de MOF consiste de MetaClasses, MetaAttributes, y algunos otros. Como muestra, UML: Class, Attribute, Operation. Data Warehousing: Base de datos, tabla, fila.

- **M1** (Modelo): Un modelo expresado en términos de un metamodelo. Modela un dominio de información específico. Consiste de instancias de elementos de un metamodelo (es decir metaobjetos).

- **M0** (Objetos de usuario): Instancias de los elementos de un modelo.

XMI proporciona un mecanismo para derivar un XML-DTD que representa los constructores de un metamodelo compatible con MOF.

Una herramienta compatible con MOF puede por lo tanto representar y enviar un modelo de dominio basado en UML en la forma de XML cuya estructura está de acuerdo a un XMl DTD. El objetivo de quienes envían un XMI es que el XML sea usado para importar y exportar modelos desde y a un repositorio persistente.

Todo lo anterior justifica la sugerencia que realiza OMG en la definición de arquitectura propu en el profile [22], de la utilización de XMI como formato de intercambio entre las diversas partes, como el editor y el configurador del modelo.

Después de habernos puesto en situación y teniendo claras las ideas que se han comentado pasamos a representar gráficamente, cómo ArgoUML exporta los elementos más importantes (desde el punto de vista de nuestro trabajo) que podemos encontrarnos dentro de un modelo UML.

Lo primero que nos vamos a encontrar dentro de un fichero XMI generado con ArgoUML (0.18.1) es la siguiente cabecera:

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Novosoft UML Library</XMI.exporter>
      <XMI.exporterVersion>0.4.20</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>
```

Figura B.7: Cabecera del documento XMI.

La primera también conocida como *declaración XML*, define la versión de XML que estamos utilizando, además en nuestro caso también especificamos la codificación del documento, en nuestro caso UTF-8. Después de esta línea comienza información referente a la versión de XMI, cual es el metamodelo que mapea y su versión, UML v1.3 para nosotros, la línea que nos informa de esto es: `<XMI.metamodel xmi.name="UML" xmi.version="1.3"/>`.

Diremos que `XMI.metamodel` es el **elemento** de la etiqueta, `xmi.name` será un **atributo** del elemento anterior, lo mismo que xmi.version, y tanto `"UML"` como `"1.3"` serán los respectivos valores de los atributos. Un elemento podrá llevar anidados otros elementos, atributos o ambos. A continuación vamos mostrar cómo están serializados los diagramas en formato XMI.

B.2.2. Diagrama de clases

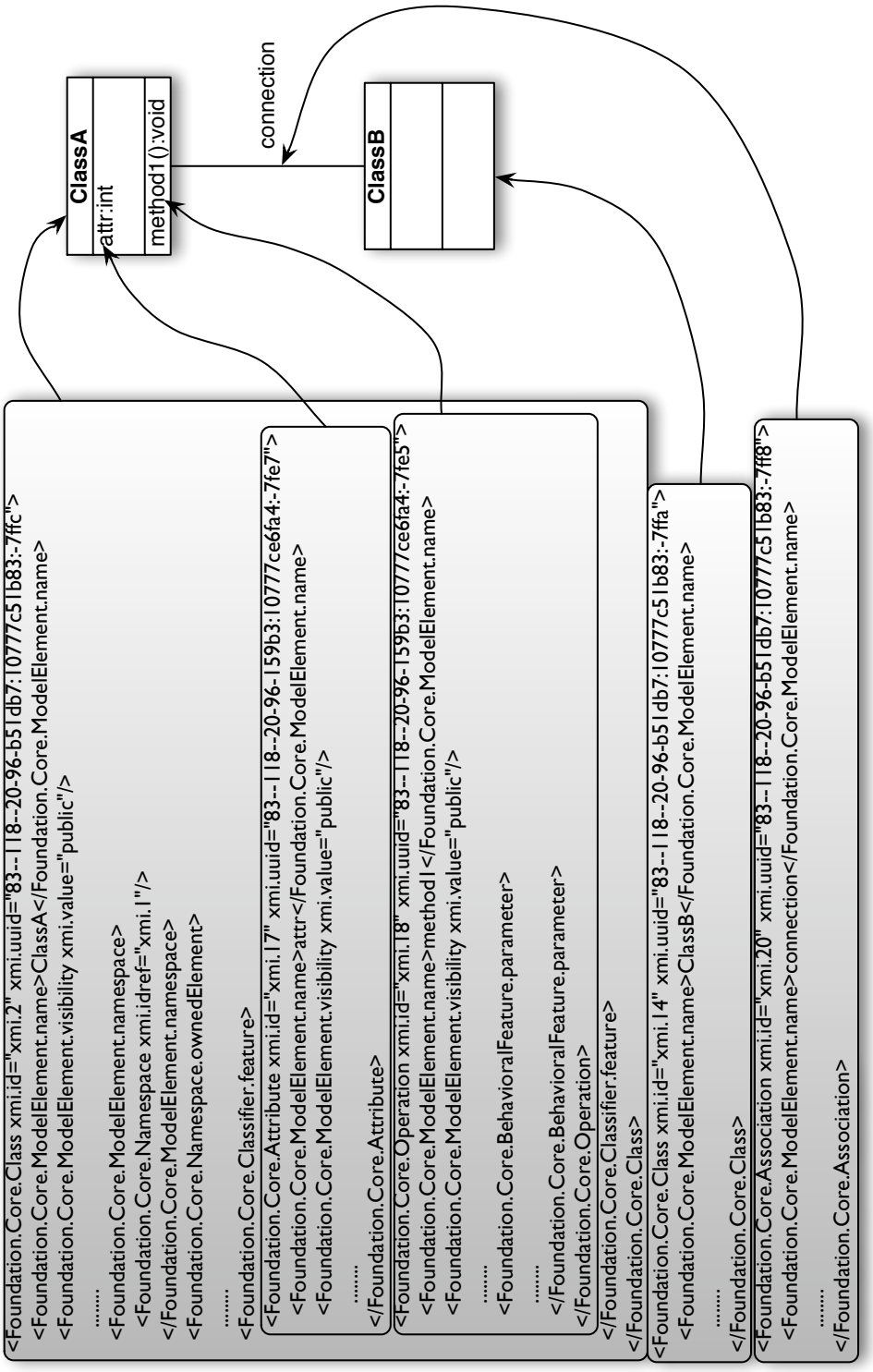

Figura B.8: Representación XMI de los elementos del diagrama de clases.

B.2.3. Diagrama de despliegue

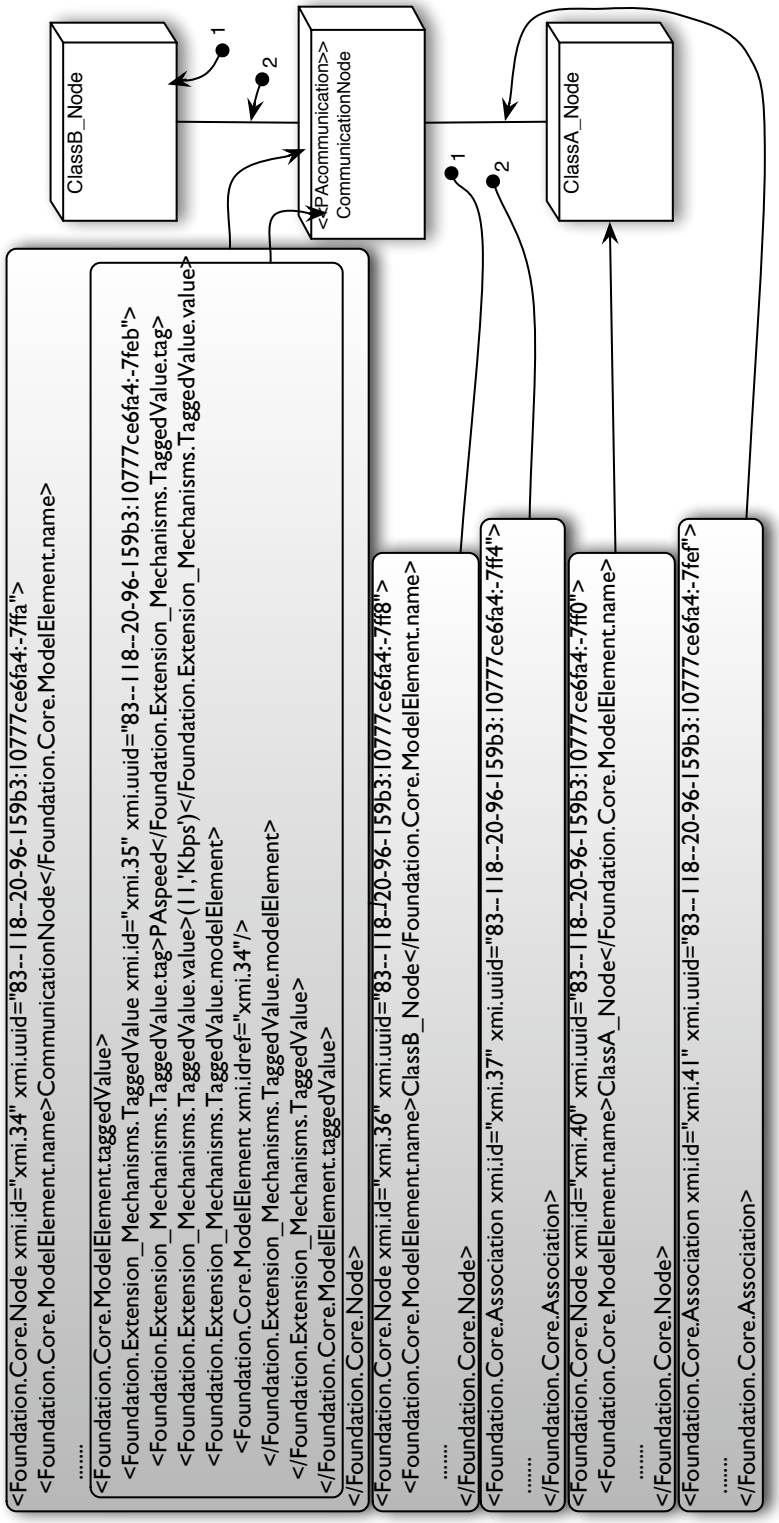

Figura B.9: Representación XMI de los elementos del diagrama de desarrollo.

B.2.4. Diagrama de estados

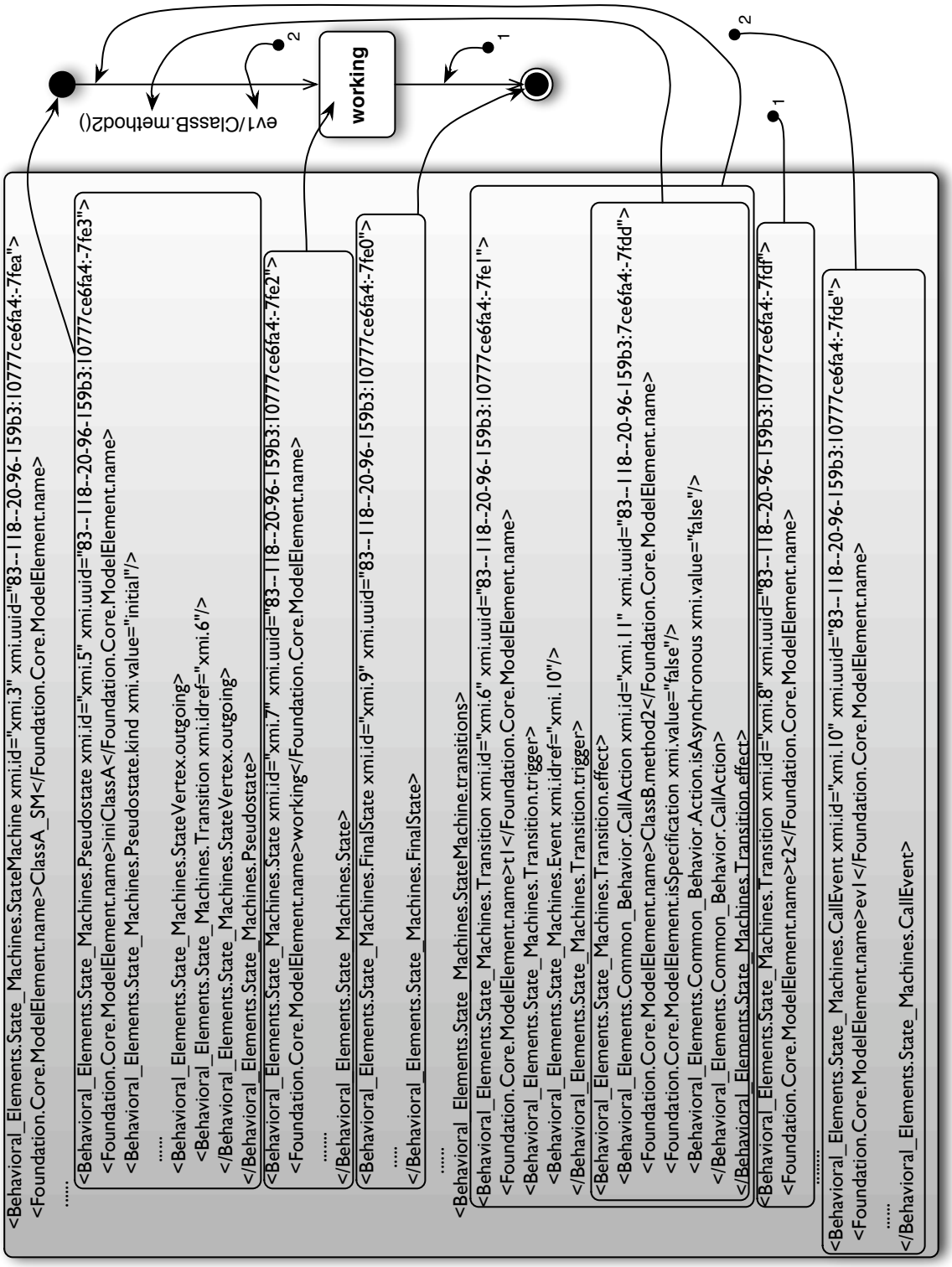

Figura B.10: Representación XMI de los elementos del diagrama de estados.

B.2.5. Diagrama de colaboración

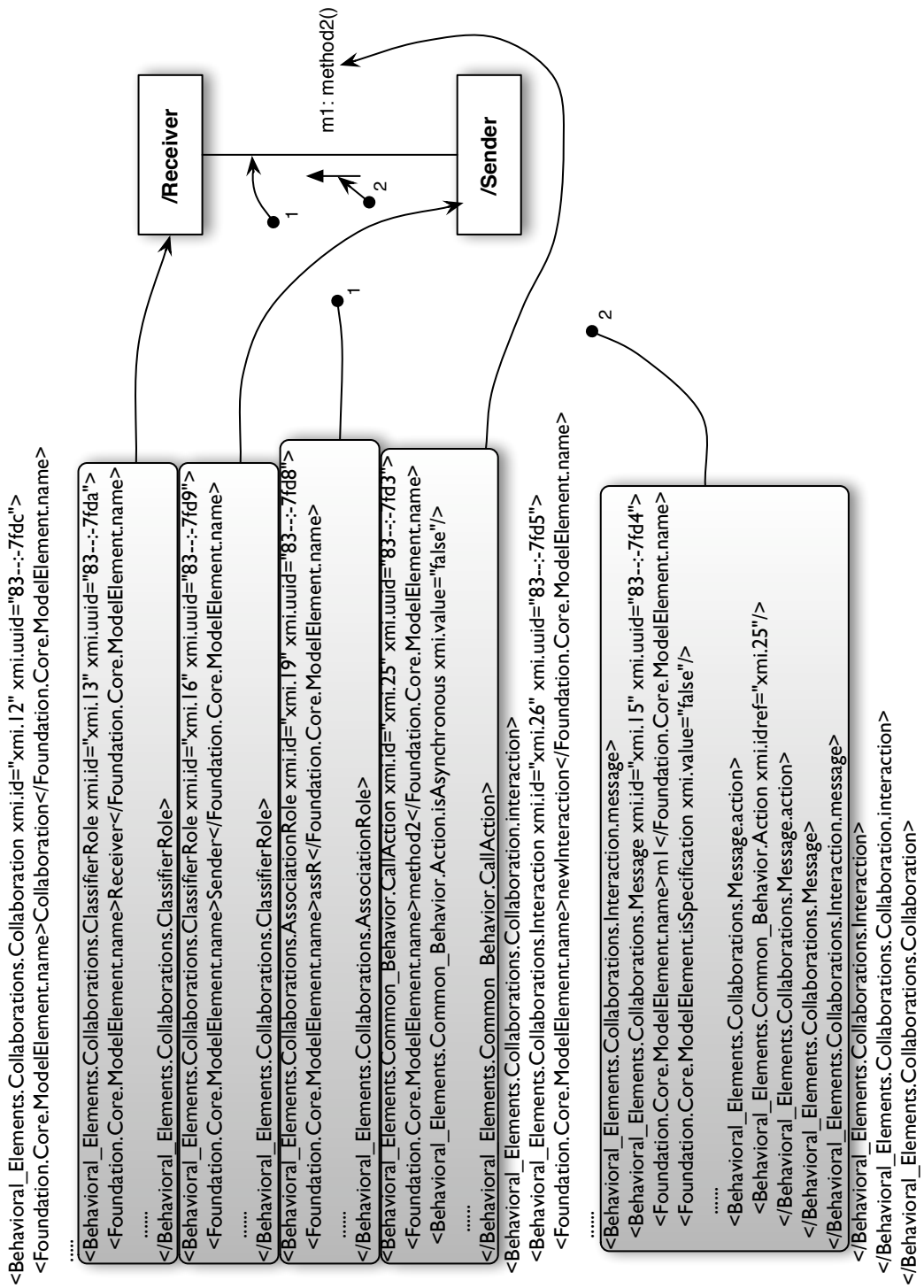

Figura B.11: Representación XMI de los elementos del diagrama de colaboración.

Apéndice C

Implementación de soluciones

A lo largo de este anexo vamos a exponer los detalles más importantes de la implementación de ArgoSPE que han estado directamente relacionados con mi proyecto, los cuales fueron estudiados para poder extender adecuadamente la herramienta, sin provocar ninguna modificación no deseada dentro del comportamiento de la aplicación.

En este apéndice comentaremos los fallos más relevantes que hemos encontrado, fundamentalmente en el proceso de traducción de máquinas de estados a LGSPN's. Para finalizar dejaremos constancia de cúal ha sido el resultado final tanto de los procesos de traducción de los diagramas de colaboración, como del mecanismo de composición con las LGSPN's de las máquinas de estado.

C.1. Errores

C.1.1. Modificación caso A

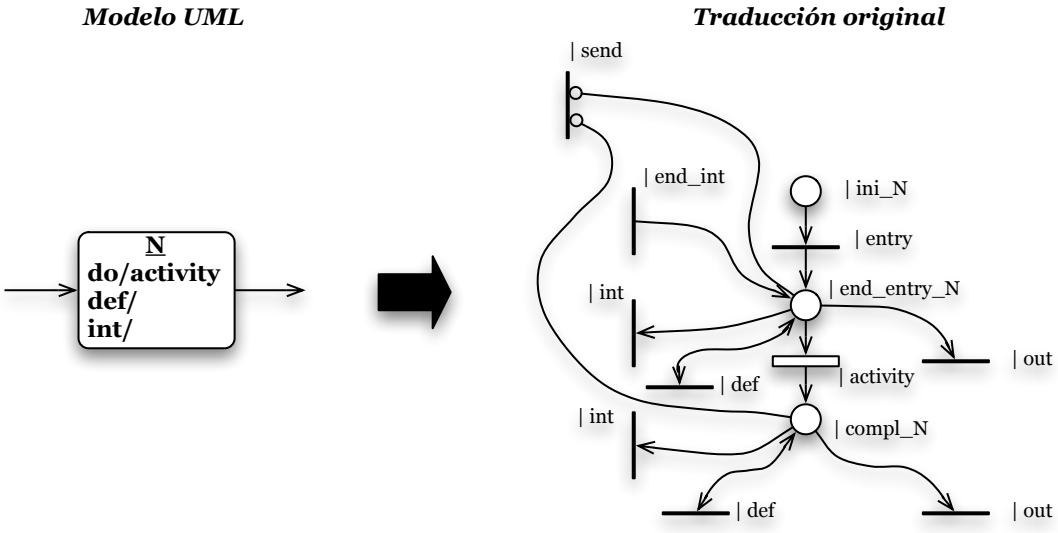

Figura C.1: Representación de la traducción original del estado N.

La figura anterior C.1 muestra la traducción propuesta en [17] para un estado que posee una *do/activity*[1], un evento diferido y una transición interna[2].

En el trabajo [4] su autor consideró oportuno realizar una serie de modificaciones con el fin de evitar la duplicidad de etiquetas que se producían con la transformación anterior, por ejemplo, en las transiciones que representan la transición interna del estado, etiquetadas con | `int`. Como resultado de estas transformaciones la red de Petri resultante del estado mencionado anteriormente quedaría algo como esto:

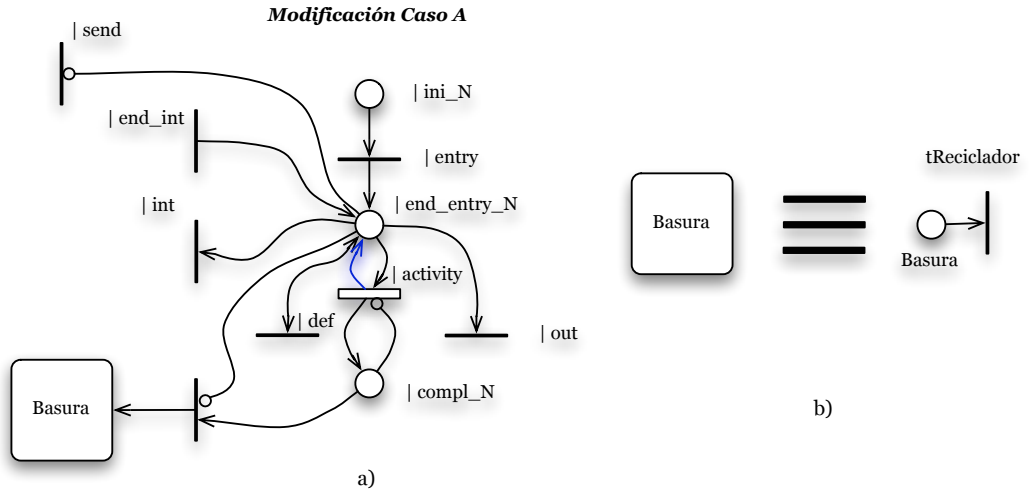

Figura C.2: Representación de la traducción modificada del estado N.

Se puede apreciar claramente en la figura que uno de los arcos dirigidos que hemos representado, tiene un color diferente del resto de los elementos que constituyen el esquema, en este caso el **azul**, gracias a las revisiones realizadas de nuestro módulo se advirtió la ausencia de esa transición que implicaba una modificación en el comportamiento que debía modelar la RdP.

También cabe destacar el hecho de la aparición de un *estado* al que hemos denominado *Basura*, la representación de este estado en elementos del dominio de las RdP viene reflejada en la parte derecha, la figura *b*, el lugar Basura recibe las marcas que representan los objetos que han terminado la ejecución de la acción del estado y que deben ser eliminados de la red para conseguir un funcionamiento correcto de la misma, ya que sino podría impedirse el disparo de la transición temporizada que describe la acción del estado.

En el esquema C.2 podemos comprobar que existen dos arcos con un círculo en uno de los extremos, estos arcos se denominan **arcos inhibidores**, estos arcos hacen que la transición que los contiene **no** pueda dispararse si en el lugar enlazado por cada uno de los arcos existen al menos tantas marcas como indica el peso del arco. Estos arcos son una extensión de las RdP para implementar la lógica negativa.

Después de presentar la modificación hemos considerado adecuado explicar brevemente el comportamiento dinámico modelado por la RdP para que el lector se pueda

[1] Acción que deberá realizar un objeto de esa clase cuando permanezca en el estado representado en el esquema.

[2] La diferencia entre una transición interna y una autotransición es que cuando se produce una transición interna no se ejecuta ni la acción de entrada ni la acción de salida.

hacer una idea más clara de los pasos que realiza una instancia cuando se encuentra en un estado, vamos a representar este recorrido a través de la figura C.3.

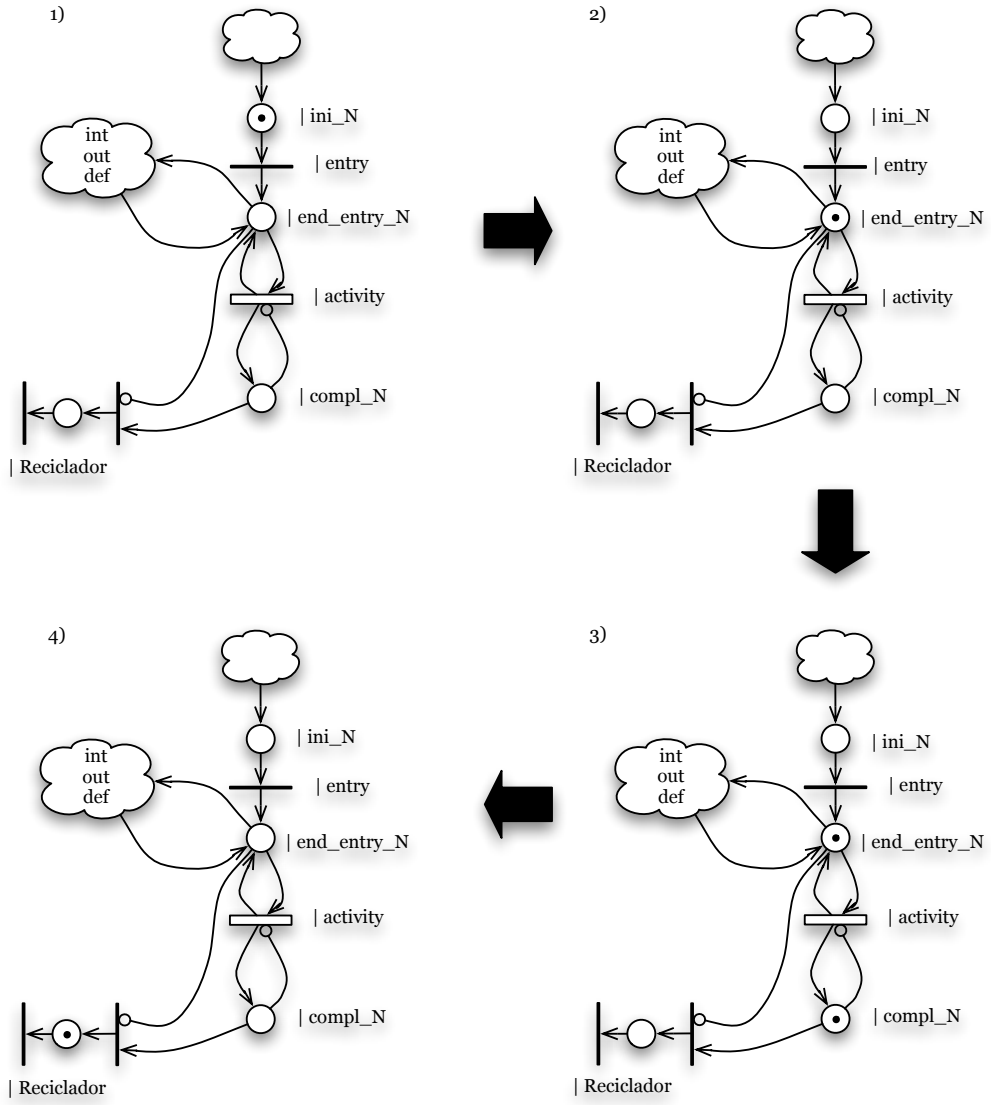

Figura C.3: Ejecución de la red de Petri modificada del estado N.

Lo primero que tenemos que comentar es que se ha comprobado que el comportamiento descrito por la RdP original y la modificada es el mismo, por lo que no perdemos nada de semántica con la transformación, así que consideraremos la modificación como **correcta**.

Dicho esto es preciso aclarar la existencia de dos nubes en cada una de las RdP que componen el esquema, la primera indica que esas RdP son parte de la RdP que representa la máquina de estados en la que se encuentra el estado N, pero que no hemos querido reflejar para evitar la complejidad que supone la representación de la RdP completa del diagrama de estados.

La otra nube tiene un propósito similar aunque esta vez, intentamos reflejar que

existen otros elementos pertenecientes a esa red que modelan tanto, la transición interna, como la transición de salida del estado, como el evento diferido.

La figura C.3 1), representa la situación en la que una instancia de la clase representada por la máquina de estados en la que se encuentra el estado N, ha entrado en ese estado. Lo primero que tiene que hacer nada más entrar será ejecutar la acción de entrada (*entry action*) asociada al estado, como nos podemos imaginar si nos fijamos en las etiquetas la *entry action* está modelada como una transición inmediata.

La ejecución de esta acción es recogida en la red de Petri con el disparo de la transición etiquetada como `| entry`, se puede observar que se cumplen todos los requisitos para el disparo de la misma, ya que todos los lugares de entrada, en este caso el etiquetado con `| ini_N`, tienen como mínimo el número de marcas que indica el peso del arco que lo enlaza con la transición, es decir, uno.

Como resultado del disparo de la transición (o ejecución de la acción de entrada), la marca pasa a estar situada en su lugar de salida, `| end_entry_N`, (ver C.3 figura 2)). Si nos fijamos con atención vemos que la transición temporizada tiene **dos** lugares de entrada, el lugar de la salida de la transición anterior y el lugar `| compl_N`, el primero contiene una marca, pero el segundo no contiene ninguna aunque esto es exactamente lo que necesitamos para que la transición se encuentre sensibilizada, ya que este lugar lo enlaza un arco inhibidor a la transición.

Por tanto el disparo de la transición se efectuará una vez se haya producido el retraso indicado por la anotación asociada a dicho estado dentro del modelo UML. El resultado de este disparo hace que el marcado de la red sea el presentado en la figura 3 del esquema C.3.

En este momento vemos que ha aparecido una marca en el lugar `| compl_N` y otra en `| end_entry_N`, lo cual produce que **ninguna** de las transiciones de la RdP, que estamos contemplando, esté sensibilizada, esto es debido a la acción de los arcos inhibidores.

La única manera que tenemos para que la red modifique su marcado será que se produzcan algunas de las opciones que han sido encerradas en la nube, es decir, que se produzca el evento que dispare o la transición interna, o la de salida, o bien sea un evento diferido.

Vamos a suponer que el evento que dispara la transición de salida ha sido generado desde el exterior de la máquina de estados, esto hará que la marca del lugar `| end_entry_N` desaparezca y se dispare la transición que introduzca la marca de `| compl_N` dentro del estado Basura. Con lo que obtendremos el marcado expuesto en la figura C.3 4), de esta manera la red queda lista para otra instancia de la clase.

C.1.2. Traducción del pseudoestado elección

Al comienzo de nuestro proyecto fueron notificadas dos situaciones que podrían producir un error en la traducción, la primera ha sido comentada en la subsección anterior y la segunda se refiere a la traducción que se realizaba de los pseudoestados elección (o *choice pseudostates*), elementos pertenecientes a las máquinas de estados.

Los **pseudoestados de elección** resultan de la evaluación dinámica de las guardas de sus transiciones de salida, por lo cual definen una ramificación condicional dinámica. Esto significa que la transición de salida a tomar depende de una función o resultado previamente calculado. Las guardas tienen un valor booleano, son exclusivas y al menos una de ellas debe devolver el valor cierto para que el modelo esté bien formado.

Una de las cosas más importantes para poder analizar si la traducción a RdP es correcta es comprender qué se está modelando desde el nivel de UML, para ello vamos a ver un sencillo ejemplo para explicar el significado de lo que modelamos.

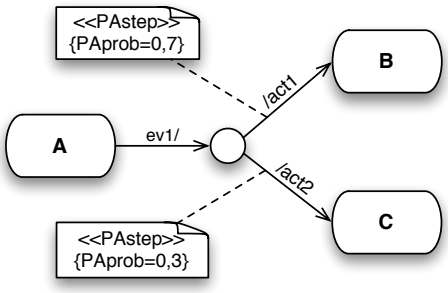

Figura C.4: Diagrama de estados con un pseudoestado choice.

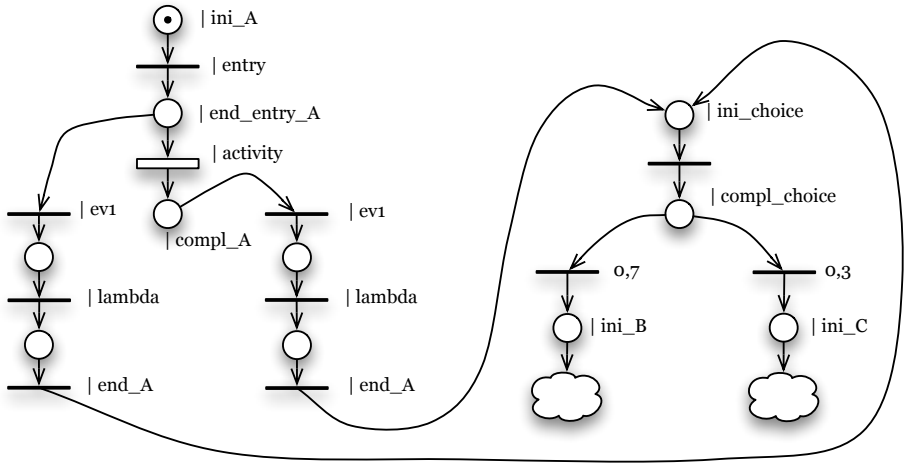

Figura C.5: Traducción del diagrama de estados de la figura C.4.

Podemos obsevar en la figura C.4 que es una de las máquinas de estados más simple que se puede modelar conteniendo un pseudoestado de elección. Cuando sea recogido un evento de tipo *ev1* por esta máquina de estados la transición de salida del *estado A* será disparada, entonces entraremos en el pseudoestado de elección, para representar la verificación de una guarda hemos colocado unas anotaciones en cada una de las transiciones de salida de la elección, esto indica la probabilidad con la que nos decantaremos por una de las transiciones de salida del *branch*, nombre por el que también es conocido el estado de elección.

Al utilizar este pseudo estado hemos variado el comportamiento normal de las transiciones de las máquinas de estados, puesto que ahora al dispararse la transición de salida no ejecutaremos la acción que hemos modelado como efecto de la misma, sino que se ejecutará el efecto (o acción) de la rama que se escoja en el *choice*.

Los eventos desencadenantes de las transiciones de salida del *branch*, también son inocuos debido a que una instancia no permanecerá en el pseudoestado elección como

podría hacerlo en un estado normal, por lo que el único evento que se tendrá en cuenta a la hora de cambiar de estado es el modelado en la transición de salida de *A*.

Debemos comentar que la RdP representada en la figura C.5 no es exactamente idéntica a la que aparece en la traducción de nuestra aplicación, ya que faltan algunas partes como la traducción de los estados *B* y *C*, pero en lo referente al pseudoestado de elección, que hemos denominado *choice*, es su representación **exacta**.

Como apreciamos en la figura C.5 la LGSPN resultante representa perfectamente lo que queríamos modelar con la máquina de estados inicial, con lo cual tenemos que concluir que la traducción realizada por nuestra aplicación es la adecuada.

C.2. Traducción

Este es uno de los puntos más importantes de nuestro trabajo, debido a que es la base de una correcta composición con las máquinas de estado, y guarda además una estrecha relación con la consulta que hemos implementado.

Antes de comenzar con la traducción de los diagramas de colaboración debemos dejar constancia de la representación de los diagramas de estados en el dominio de las GSPN's.

C.2.1. Máquinas de estado

Cada máquina de estados que compone el modelo UML que hemos diseñado será representada por una única RdP, el proceso de obtención de esta red está basado en la composición de pequeñas subredes que son obtenidas tanto de la traducción de los estados que componen la máquina como de las transiciones que unen a dichos estados.

La subred generada de la traducción de un estado UML dependerá de los elementos que presente dicho estado, como por ejemplo, la existencia de una *entry action*, de una *do activity* o de transiciones internas pueden hacer variar de una manera significativa la red obtenida.

Lo mismo ocurre en el caso de las transiciones, aunque esta vez tendremos que fijarnos en si la transición cuenta con un disparador (*trigger*) y un efecto (*effect*). Estos dos elementos son los que utilizaremos como puntos de conexión con otros elementos del modelo.

Para nuestro trabajo ha resultado muy práctico conocer al detalle la implementación de cómo estaban traducidos los diagramas de estados dentro de la herramienta, en primer lugar para poder reparar los defectos que habían sido encontrados y posteriormente para realizar unas cuantas modificaciones, que nos facilitarían más tarde la tarea de la composición.

Lo anteriormente dicho pone de manifiesto la importancia de conocer al detalle la traducción de este tipo de diagramas, motivo por el cual intentaremos dar la mayor precisión posible a esta sección. Para ello creemos necesario modelar una máquina de estados con la gran parte de los elementos que podemos encontrar en ella, para posteriormente ir asociando estos elementos con su traducción. El modelo de UML utilizado será:

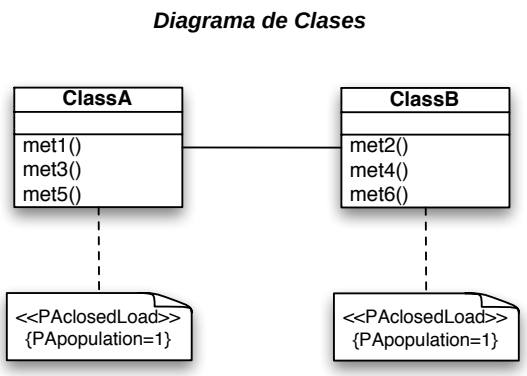

Figura C.6: Diagrama de Clases a traducir.

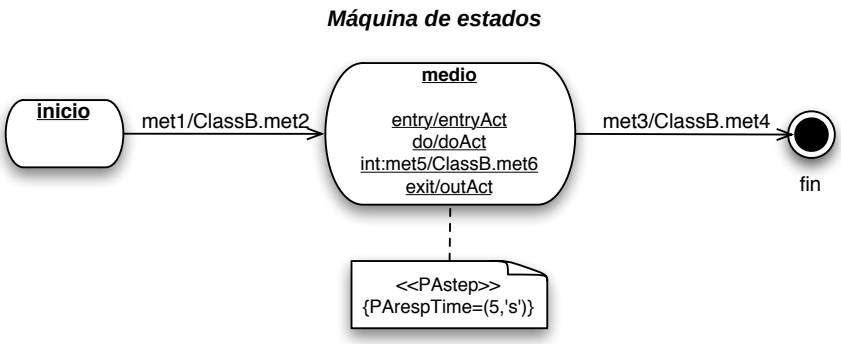

Figura C.7: Máquina de estados a traducir.

Con el fin de representar lo más claramente posible la RdP resultante de la traducción del diagrama de estados vamos a representar por separado cada una de las RdP de cada estado.

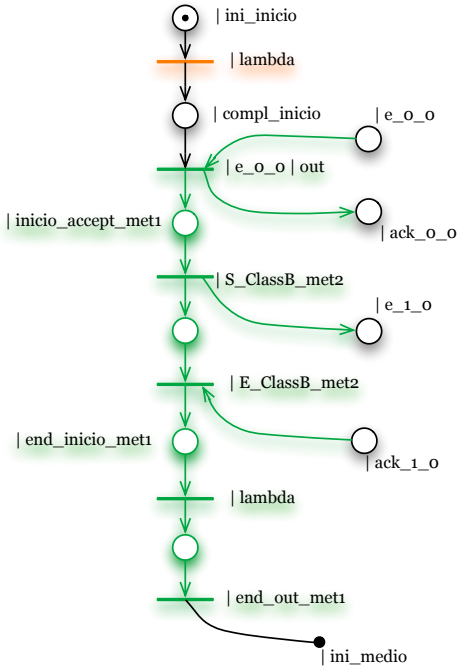

Figura C.8: GSPN obtenida del estado inicio.

En esta primera red C.8 queda representado como es traducido el estado *inicio* en el que únicamente contamos con un elemento relevante, la transición de salida, esta transición posee un evento disparador y un efecto, la parte **verde** de la red representa los elementos de la RdP asociados a la transición de salida, y los elementos de color **naranja** serán los de la acción de entrada.

En la siguiente red también hemos utilizado otros colores con el fin de distinguir qué partes están relacionadas con ciertos elementos de UML, como por ejemplo el **morado** con la *do activity* y el **azul** con la transición interna.

También podemos encontrar elementos en un color **negro**, estos elementos no están relacionados con ningún componente concreto de UML sino que forman parte de la RdP, por ejemplo, los lugares cuyas etiquetas empiezan por `| e_`, o `| ack_`, son lugares de enlace con otras máquinas de estado. Cabe destacar la etiqueta `lambda` que será utilizada cuando no exista el elemento que representa el lugar o la transición en la cual se coloque.

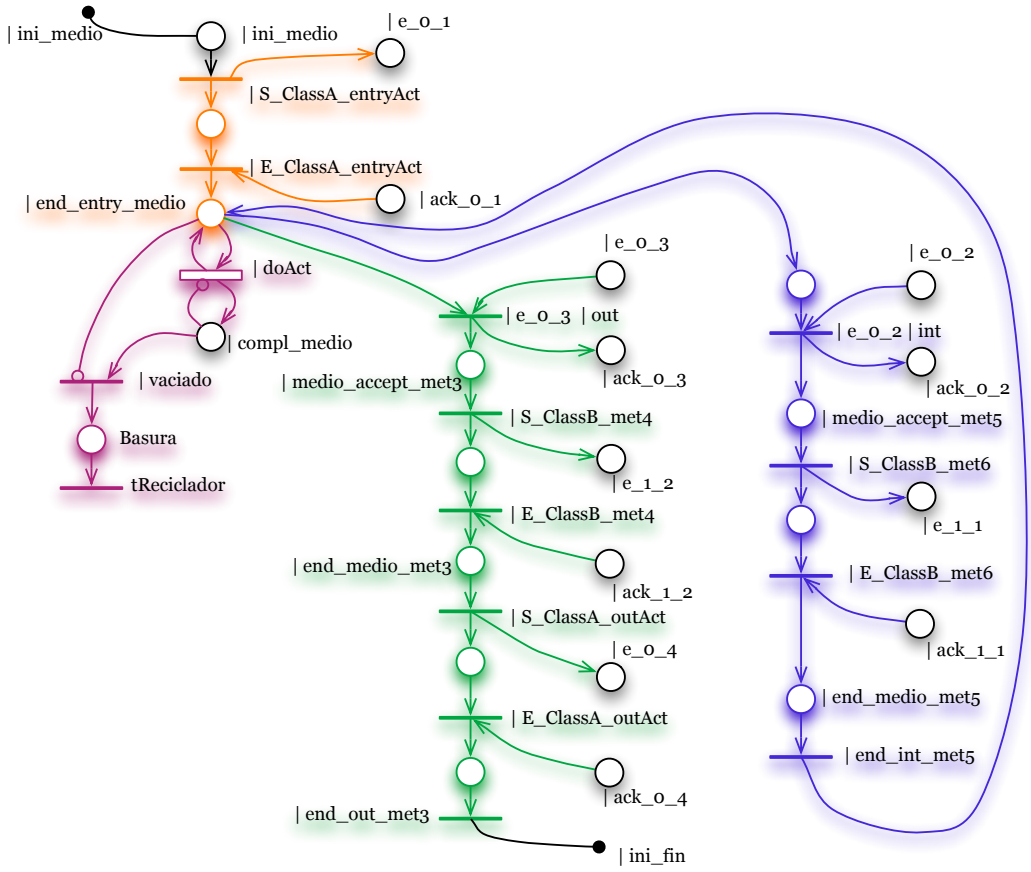

Figura C.9: GSPN resultante del estado medio.

Se puede apreciar claramente cómo dependiendo de los elementos que aparecen en UML la RdP cambia sustancialmente, este hecho queda reflejado en las diferencias entre la traducción del estado *inicio* y la del estado *medio*, este último posee acción de entrada, *do/activity*, transición de salida y transición interna, se puede decir que modela prácticamente todos los elementos que puede albergar un estado.

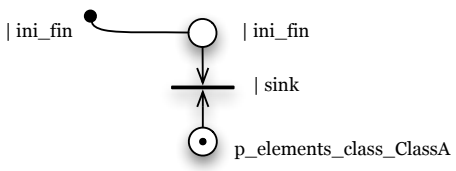

Figura C.10: GSPN representante del estado fin.

A simple vista vemos que la RdP del pseudoestado final de la máquina de estados es la más sencilla de todas las que hemos presentado, esto se debe a la propia naturaleza de este elemento que simplemente expresa el fin de la vida de la instancia, su último lugar el etiquetado con `| p_elements_class_ClassA` posee tantas marcas como población tenga la clase cuya máquina de estados estamos contemplando.

C.2.2. Diagramas de colaboración

En algunas secciones anteriores de este documento (ver 4.2.1) hemos mostrado con algún ejemplo simple cómo se traducían los diagramas de colaboración al dominio de las GSPN's. En esta sección queremos mostrar los cuatro tipo de traducciones posibles que podemos encontrar para los mensajes que seríamos capaces de modelar en un diagrama de colaboración de ArgoUML.

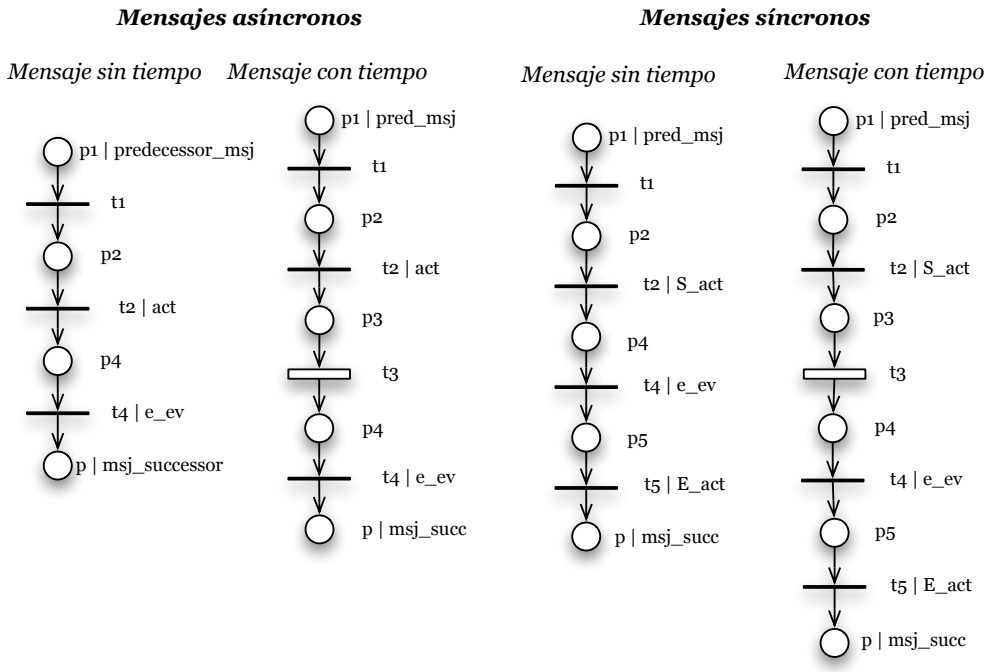

Figura C.11: GSPN's representantes de todos los tipos de mensajes.

Tenemos que puntualizar una serie de detalles sobre esta traducción, la estructura de las RdP está seguida exactamente como queda representada en la figura C.11, aunque existen unas pequeñas variaciones en cuanto a dos etiquetas de dos lugares.

Cuando estamos traduciendo el primer mensaje de una interacción el primer lugar de éste también será el primer lugar del diagrama de colaboración con lo que la etiqueta de este será `| startCoD`, y no `| _msj` como cabría esperar, lo mismo ocurre cuando estamos traduciendo el último mensaje de la interacción reflejada en el diagrama de colaboración, su último lugar también será el último lugar del diagrama de colaboración por lo que quedará etiquetado como `| endCoD`.

Para que la traducción de un diagrama de colaboración puede llevarse a cabo se tienen que cumplir una serie de condiciones en el modelo que queremos analizar, la primera de ellas indica que cada una de las clases que participa en la interacción descrita por el

diagrama de colaboración deberá describir su comportamiento por medio de un diagrama de estados. La segunda es referente a los mensajes y expresa que todos los mensajes que están recogidos dentro de un diagrama de colaboración, tienen en la máquina de estados del emisor una acción que causa el envío del mismo, y en el diagrama de estados del receptor existirá al menos una transición que modele la respuesta a ese evento.

Esta segunda hipótesis hace que debamos modificar el algoritmo de traducción que utilizaba la herramienta ArgoSPE antes de soportar la traducción de los diagramas de secuencia, está claro que tenemos que poseer toda la información de todas máquinas de estados que existen en el modelo para poder asegurar el cumplimiento de la segunda hipótesis, de otra manera sería imposible.

Para representar el algoritmo de traducción antiguo y el modificado vamos a utilizar unos diagramas de actividades:

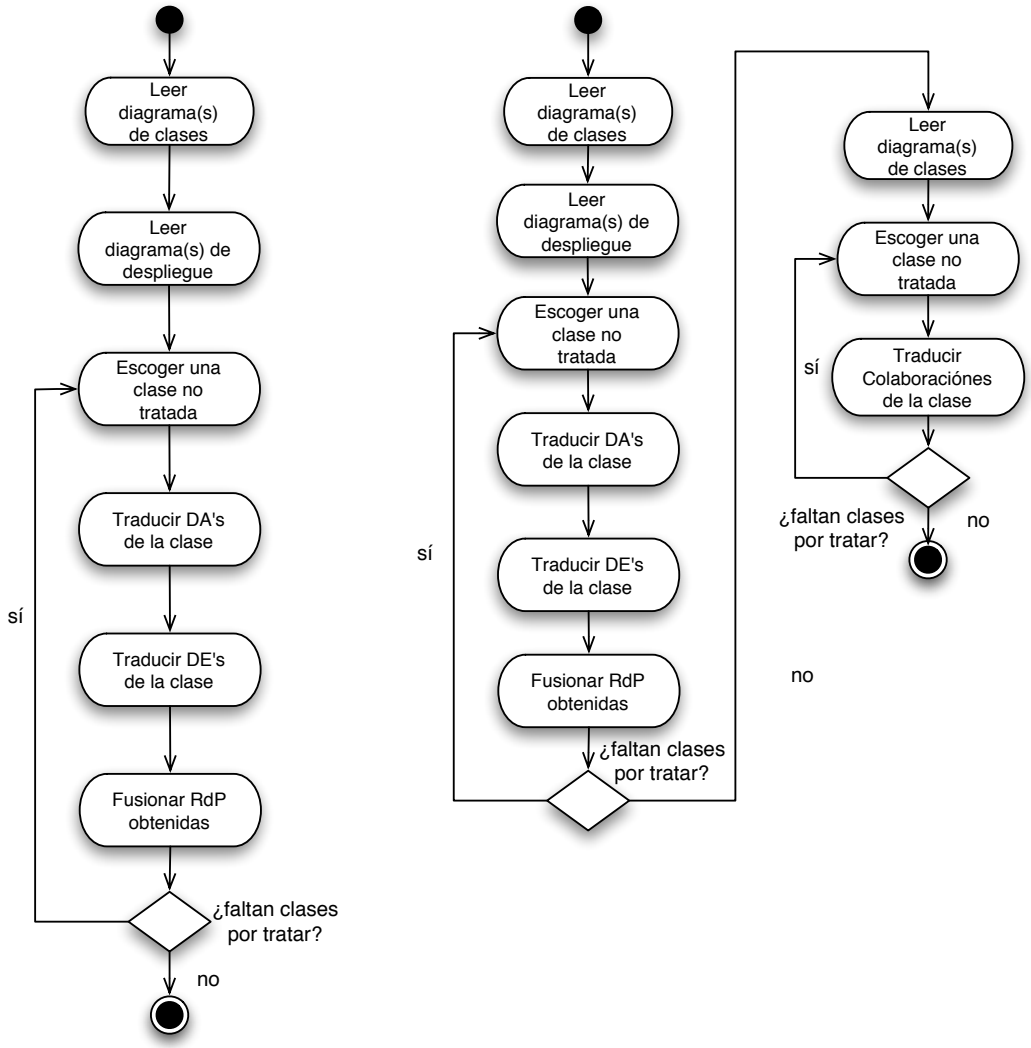

Figura C.12: Diagramas de actividades de los algoritmos de traducción de ArgoSPE.

Como es evidente el algoritmo antiguo será el representado por el diagrama de actividades de la parte izquierda y el más reciente estará reflejado en la parte derecha de la

figura. Vemos que simplemente hemos tenido que sacar del bucle principal el recorrido de los diagramas de colaboración que poseen cada una de las clases que constituyen el modelo.

C.3. Composición

El proceso de composición envuelve a la RdP resultante de la composición de las RdP de las máquinas de estados y de los diagramas de actividad, con la RdP de uno de los diagramas de colaboración que han sido modelados por el diseñador del sistema software.

Esto quiere decir que realmente se tienen que realizar dos composiciones para poder conseguir la representación, en el dominio de las RdP, de la situación descrita por el diagrama de colaboración seleccionado del sistema.

Lo lógico será pues comenzar por representar cómo se realiza la primera de las composiciones, principalmente nos interesará conocer cómo se fusionan las máquinas de estados para poder obtener una representación del sistema completo.

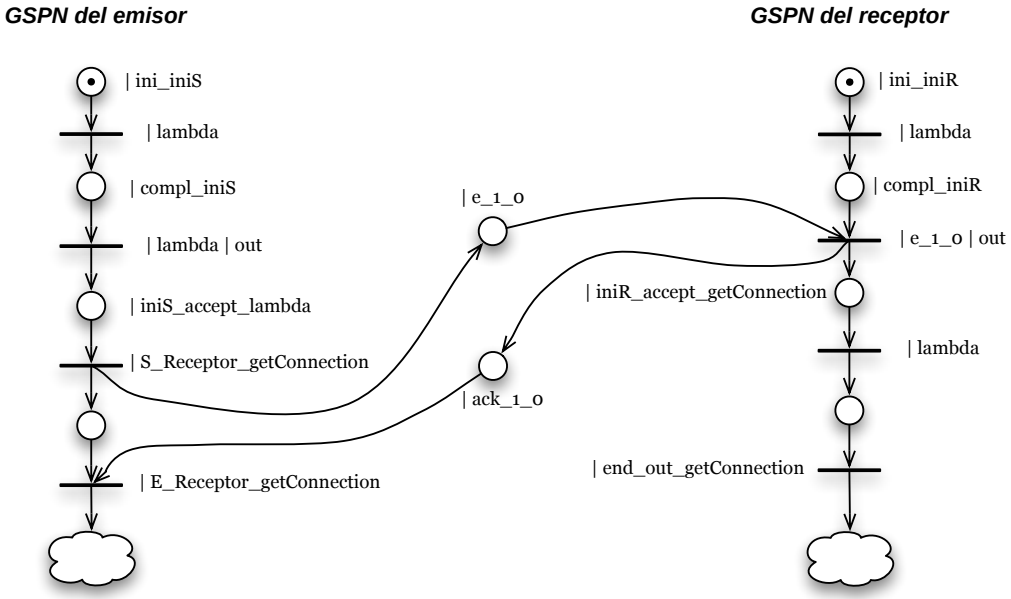

Figura C.13: Esquema de la composición de dos máquinas de estados.

Al observar con un poco de detenimiento la figura C.13 vemos representado en la RdP cómo, la clase *Emisor* ha modelado en la transición de salida inmediata del estado *iniS* el lanzamiento de un evento que será recogido por la máquina de estados de la clase *Receptor*.

Dicho de otra manera en la máquina de estados de la clase Emisor existe una transición que tiene como efecto la ejecución de la operación `Receptor.getConnection`, método de la clase *Receptor*, que generará un evento que podrá ser recibido por la máquina de estados para que ésta modifique su estado. Esto produce que exista el efecto `Receptor.getConnection` en la transición de salida del estado *iniS* y que a su vez tengamos un evento disparador en la transición de salida del estado *iniR*.

Una vez explicada de forma breve la composición entre máquinas de estado vamos a suponer que tenemos un diagrama de colaboración como este:

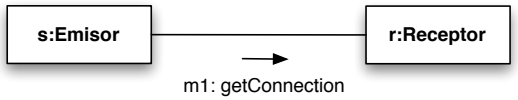

Figura C.14: Diagrama de colaboración con un mensaje.

Para poder observar más claramente la composición entre las RdP de las máquinas de estados y la del diagrama de colaboración hemos optado por un diagrama con un solo mensaje, ya que si se entiende el mecanismo con un solo mensaje extender la idea a diagramas de colaboración más complicado se hará mucho más sencillo. Por tanto la composición final quedará de la siguiente manera:

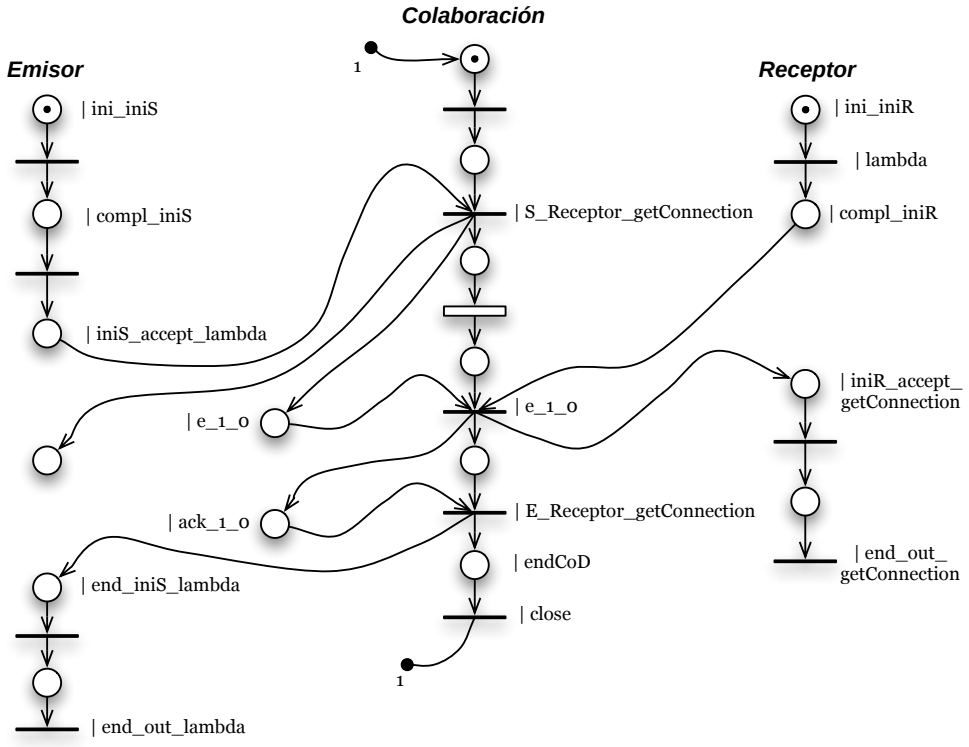

Figura C.15: GSPN representante del escenario modelado con C.14.

Éste es el resultado que muestra GreatSPN (después de unos cuantos retoques) cuando abrimos la RdP generada con ArgoSPE tras las implementaciones que hemos ido explicando.

Nos queda un detalle que no ha sido explicado y es el hecho de que con la traducción original de las máquinas de estados no se podía realizar adecuadamente la composición con el diagrama de colaboración, esto era debido al formato utilizado en la construcción de las etiquetas de las transiciones que representaban los eventos, por ejemplo, un evento que desencadenaba una transición de salida era etiquetado así:

```
| out_e_0_1
```

El formato utilizado refleja que estamos en el caso de un *trigger event* de una transición de salida, por, `out`, que pertenece a la clase cuyo identificador interno es el `0`. El identificador interno del evento en cuestión será el `1`.

El problema de esta etiqueta era que para poder componerla con su semejante del diagrama de colaboración deberíamos tener exactamente la misma etiqueta en las dos transiciones, pero desde el diagrama de colaboración desconocíamos si un evento venía de una transición de salida, o desde una interna, o de cualquier otra fuente, solamente conocíamos el nombre de la clase destino y el nombre del evento.

La solución adoptada fue la de modificar la etiqueta generada desde el traductor de las máquinas de estados para que encapsulara la misma información que tenía antes pero con un formato tal que pudiera ser reproducido desde el traductor de los diagramas de colaboración.

El formato que implementamos era el siguiente:

```
| e_0_1 | out
```

Esto hace que tengamos dos etiquetas dentro de una transición, pero simplemente tenemos que hacer que la etiqueta con la que queramos componer se encuentre en el fichero de etiquetas de su correspondiente red de Petri. Por supuesto nos aseguramos de que la modificación del formato de esas etiquetas no interfería ni en el proceso de traducción ni en el de composición de las máquinas de estado.

Apéndice D

Caso práctico

A lo largo de este anexo vamos a reunir todos los conocimientos que he ido adquiriendo a lo largo del desarrollo de nuestro proyeto, sobre el funcionamiento de la herramienta ArgoSPE, (modelado de diagramas, ejecución de consultas) para ello vamos a explicar cómo se utilizaría ArgoSPE con un ejemplo de un sistema software concreto.

D.1. Modelado

Si pensamos un poco en ello, es importante elegir bien el ejemplo que vamos a utilizar dentro de este capítulo ya que tiene que ser lo suficientemente amplio como para abarcar el mayor número de características de la aplicación, pero también ha de ser conciso para no perdernos en detalles que no nos aporten nada.

Una muestra del tipo de ejemplos que serían útiles en este punto sería el *WatchDog Timer* [1] del proyecto DepAuDE (**Dep**endability for embedded **Au**tomation systems in **D**ynamic **E**nvironments with intra-site and inter-site distribution aspects) [8].

D.1.1. Funcionamiento del WatchDog Timer

El WatchDog Timer, **WT** a partir de ahora, es un mecanismo de tolerancia a fallos (**FT**) que ha sido diseñado e implementado dentro del proyecto europeo DepAuDE. El principal objetivo de este proyecto es proporcionar una *framework* (o marco de trabajo) que incremente la fiabilidad de la automatización de los sistemas software embebidos distribuídos.

El WT es un componente configurable por el usuario, los parámetros regulables son la duración de la alarma y su localización espacial, éste detecta violaciones en tiempo de ejecución del proceso de una aplicación. Este sistema se basa en un *timer* (temporizador) que es inicializado por la aplicación que estamos monitorizando antes de que finalice su cuenta atrás.

La ejecución de la inicialización del *timer* es realizada por el envío de un mensaje `"I'm alive"` (`"Estoy vivo"`), destinado al WT. Si por cualquier motivo la aplicación no es capaz de enviar este mensaje al WT automáticamente se generará un error que será transmitido a un componente del *Backbone* (**BB**).

[1] Se va a mantener el nombre en inglés puesto que la traducción resulta un tanto extensa.

D.1.2. Diagramas UML

Para conseguir representar con UML el comportamiento que hemos expuesto en la subsección anterior tenemos que utilizar diversos diagramas, el diagrama de clases, diagrama de colaboración, diagramas de estados, y el diagrama de despliegue.

Los diagramas correspondientes a nuestro sistema son los siguientes:

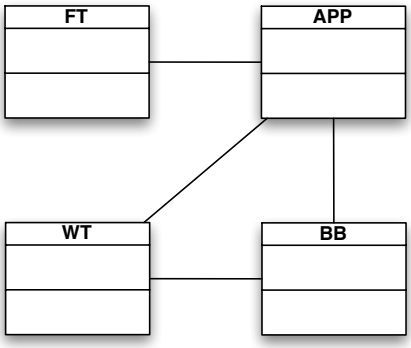

Figura D.1: Diagrama de clases de WT.

Normalmente cuando modelamos un sistema software suele ser recomendable empezar por el diagrama de clases, ya que será el que defina los componentes estructurales (clases) de nuestro diseño.

En nuestro caso contaremos con cuatro clases, *WT*, la cual representa al *WatchDog*, *APP* será la aplicación de usuario que observaremos, *BB* describirá el *Backbone* y por último *FT* modelará un fallo de la aplicación. En el diagrama de clases (ver figura D.1) podremos observar qué asociaciones tiene cada una de las clases.

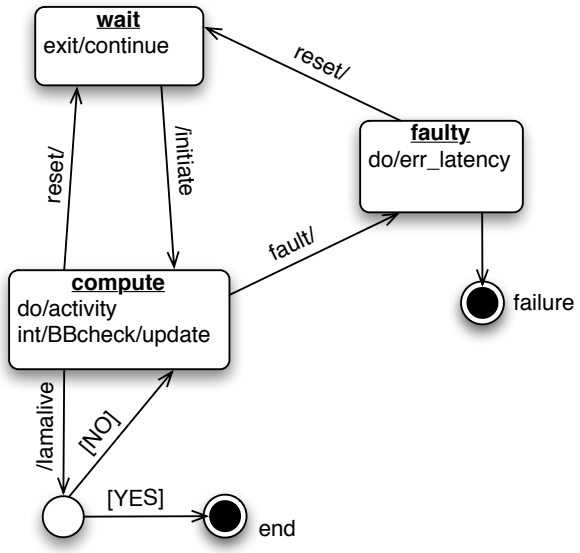

Figura D.2: Diagrama de estados de APP.

La actuación de los objetos de la clase APP vendrá definida por el diagrama D.2, una de estas instancias inicializará el WT con la ayuda de la acción *initiate*, los parámetros con los que configuraremos a WT serán pasados a través de la acción *continue*. Una vez inicializado el WT la aplicación empezará su labor ejecutando *activity*, durante este trabajo el BB podrá solicitar información de actualización a la APP enviándole un evento del tipo *BBcheck*. Después de completar su cometido el objeto enviará un *Iamalive* al WT para que reinicie su cuenta atrás para posteriormente decidir si termina o continua con su trabajo.

Durante la ejecución de la *do/activity* un evento *fault* puede ser recogido provocando que cambiemos de estado, pasando a encontrarnos en el estado *faulty*. De este estado sólo podremos salir si el BB ejecuta un *reset*, produciéndose una recuperación de la aplicación, o si la actividad *err_latency* finaliza provocando que el presente objeto finalice su vida.

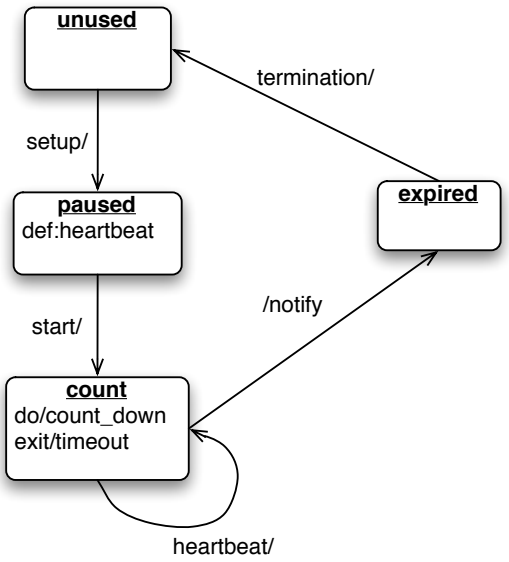

Figura D.3: Diagrama de estados de WT.

Esta máquina de estados modela el comportamiento de WT (figura D.3), el cual una vez inicializado, gracias al evento *setup*, es activado por la aplicación del usuario a través del evento *start*, en ese momento WT comienza la actividad *countdown*. Durante el proceso de cuenta atrás el WT puede recibir eventos *heartbeat* que inicializarán el contador.

Si WT no recibe ninguna señal de la aplicación de usuario se producirá un *timeout* y se enviará un mensaje notificando lo sucedido al BB. El WT podría recibir señales de que la aplicación de usuario se mantiene activa también en el estado *paused*, pero serán eventos diferidos.

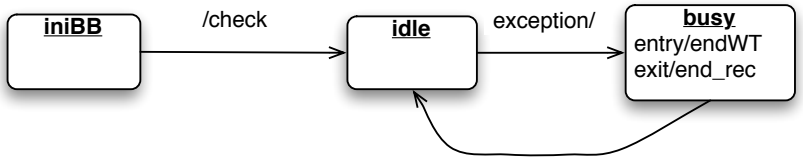

Figura D.4: Diagrama de estados de BB.

La clase BB, descrita en la figura D.4, se ocupará de emprender acciones de recuperación del sistema si en algún momento se produce una excepción, como resultado de la recepción de un evento *exception* generado por WT , en este caso BB ejecuta la acción *endWT* y también reinicia la aplicación con *end_rec*.

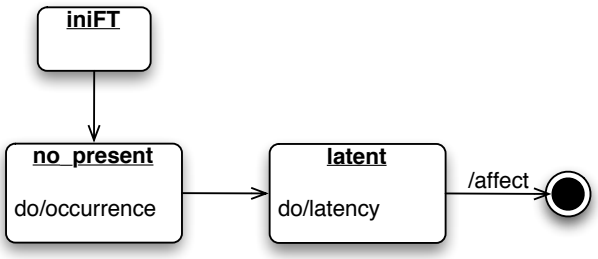

Figura D.5: Diagrama de estados de FT.

En el momento en el que ocurre un fallo de este tipo (FT) pasa a estar en el estado *latent* y empieza su período de latencia, en el instante en que éste termina ejecuta la acción *affect* dirigida a APP.

Después de haber descrito el comportamiento global de nuestro sistema gracias a las máquinas de estados de cada una de las clases modeladas, tenemos que representar un escenario en el cual colaboren dichas clases. Como es lógico esto será modelado gracias a un diagrama de colaboración.

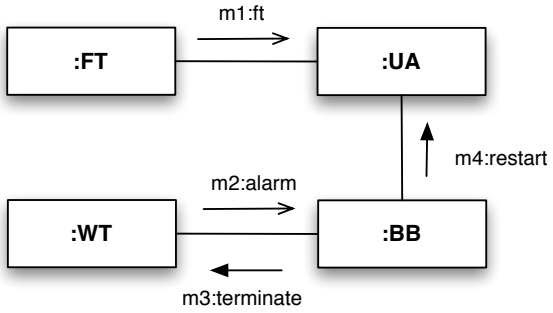

Figura D.6: Diagrama de Colaboración de una situación de fallo.

Podemos observar como en este escenario se produce un fallo representado por la señal *ft* en la APP mientras está trabajando. En cuanto el *timer* de la WT finaliza es generada una excepción recogida por el BB que obligará a terminar a WT enviándole una acción *terminate* y hará un *reset* de la aplicación de usuario, gracias a la operación *recovery*.

Lamentablemente como se ha comentado en algún capítulo anterior ArgoUML no implementa todos los elementos que son proporcionados por UML, por lo que tendremos que encontrar la manera de expresar esos elementos para que nuestro modelo sea correcto.

Este es el caso de los **eventos diferidos**, este tipo de eventos no pueden ser modelados directamente con ArgoUML por lo que nos veremos obligados a modificar el fichero XMI, a mano, que representa nuestro modelo, para hacer creer a nuestro módulo que en

realidad habíamos modelado un evento diferido. Tras abrir el fichero XMI tendremos que buscar el elemento que representa al estado al que queremos añadirle el evento, y dentro de él, le añadiremos el siguiente elemento:

```
<Behavioral_Elements.State_Machines.State.deferrableEvent>
```

Posteriormente tenemos que incluir dentro de este elemento referencias a todos los eventos que queramos que sean diferidos. Esto se hará con el subelemento siguiente:

```
<Behavioral_Elements.State_Machines.State.DeferrableEvent xmi.idref="xxx">
```

Siendo `xxx` el número que identifica a un evento dentro del fichero XMI.

Un proceso análogo tendremos que realizar para poder modelar los mensajes asíncronos del diagrama de colaboración, en este caso tendremos que encontrar la acción asociada al mensaje y modificarle el valor del atributo que la identifica como una acción síncrona.

En metamodelo de UML los eventos, las acciones, las operaciones y las señales están relacionadas a través de diversas asociaciones, por simplificar la implementación los desarrolladores de ArgoUML han eliminado gran parte de esta relación, lo que ha hecho que tengamos que utilizar los mismos nombres para relacionar los eventos que existen en una máquina de estados con las acciones realizadas por otros diagramas de estados diferentes. Debido a este motivo nosotros solamente utilizaremos la última columna de la tabla D.1 para modelar la interacción entre máquinas de estados.

Acción	**Operación/Señal**	**Evento**
initiate	init	setup
continue	cont	start
Iamalive	kick	heartbeat
notify	alarm	exception
check	control	BBcheck
endWT	terminate	termination
end_rec	restart	reset
affect	ft	fault

Cuadro D.1: Relación entre acciones, operaciones/señales y eventos.

Así por ejemplo en la máquina de estados de la aplicación de usuario tenía, en el estado *wait*, la acción de salida *continue* que en ArgoUML pasará a ser *WT.start*, como se puede apreciar las acciones se formarán con el nombre de la clase a la que pertenece el evento, seguida de un punto y el nombre del evento que queremos generar.

D.2. Anotación

Cuando modelamos un sistema software necesitaremos una serie de elementos para poder anotar las características que cuantifican las prestaciones de ciertas partes de nuestro sistema, para poder obtener resultados que nos ayuden a estudiar el rendimiento de nuestro diseño.

El conjunto de posibles anotaciones que podemos utilizar con ArgoSPE está descrito en la siguiente tabla, cabe destacar que dichas anotaciones se realizarán siguiendo el UML-SPT, que propone la utilización del lenguaje TVL:

Anotación	**Estereotipo**	**Tag**	**Elemento**	**Unidades**
Duración	PAstep	PArespTime	Estado*	ms, s, m, h
Probabilidad	PAstep	PAprob	Transición	-
Tamaño	PAstep	PAsize	Mensaje, Trigger o Effect	b, B, Kb, KB, Mb, MB
Velocidad	PAcommunication	PAspeed	Nodo	bps, Bps, Kbps, KBps, Mbps, MBps
Número inicial de objetos	PAclosedLoad	PApopulation	Clase	-
Estado inicial	PAinitialCondition	PAinitialState	Estado	$true, $false
Clases residentes	GRMcode	GRMmapping	Nodo	Identificadores

Cuadro D.2: Tabla con las anotaciones implementadas por ArgoSPE.

Podemos apreciar que en la tabla existen algunos detalles que no quedan suficientemente claros, por lo que vamos a proceder a realizar una breve explicación de cada una de las anotaciones, que figuran en el cuadro.

Es importante señalar antes de empezar con las aclaraciones de la tabla, que cualquiera de las anotaciones que está presente en la tabla debe ser representada por su estereotipo **y** por su valor etiquetado, de lo contrario nuestra herramienta no detectará adecuadamente la anotación a la cual queremos hacer referencia.

La primera anotación del cuadro D.2 es *duración*, ésta representa la prolongación en el tiempo de la *do/activity* que **debe aparecer**[2] en el estado en el cual se ha realizado la anotación. Las unidades que podemos utilizar para expresar el intervalo de tiempo son milisegundos, segundos, minutos y horas.

La *probabilidad* va asociada a las transiciones de salida de un pseudoestado de elección, una muestra de cómo anotar estas transiciones aparecería en el diagrama C.4.

El *tamaño* de un evento viene descrito por el par *PAstep-PAsize*, para poder representar adecuadamente el valor de *PAsize* tenemos que seguir el siguiente ejemplo:

```
PAsize=(8,'B')
```

Vemos que el valor asignado consta de dos partes, la cantidad y la unidad expresada entre comillas simples, esto también ocurre para el caso de la etiqueta *PArespTime*, aunque aquí tendremos otro tipo de medidas. La anotación del tamaño puede darse tanto en los diagramas de estados como en los diagramas de colaboración, aunque tenemos que tener presente que siempre que exista un diagrama de colaboración en el modelo y exista un evento anotado tanto en el diagrama de estados como en el de colaboración, la

[2]El asterisco representado junto a la palabra Estado indica que la anotación, aunque se realice en el estado, está referida a la actividad, por lo que el estado tendrá que modelar una do/activity, sino la anotación no tiene sentido.

anotación que será tenida en cuenta para la traducción a GSPN's será la del diagrama de colaboración.

La *velocidad* de transmisión de los nodos de comunicación modelados en los diagramas de despliegue, es importante para determinar los retrasos en los mensajes existentes intercambiados entre los diferentes objetos que forman parte de nuestra aplicación.

En los diagramas de despliegue también modelaremos los nodos en los que ubicaremos las clases de nuestro sistema, gracias a esto podremos hacer pruebas de rendimiento distribuyendo las clases de diferentes formas. La localización de cada clase vendrá determinada por el par *GRMcode-GRMmapping*, el valor asignado a *GRMmapping* será el identificador de una clase.

La anotación de las *clases residentes* es la **única**, en ArgoSPE, que se realiza por medio de comentarios asociados a los elementos que queremos anotar, como en el diagrama D.7.

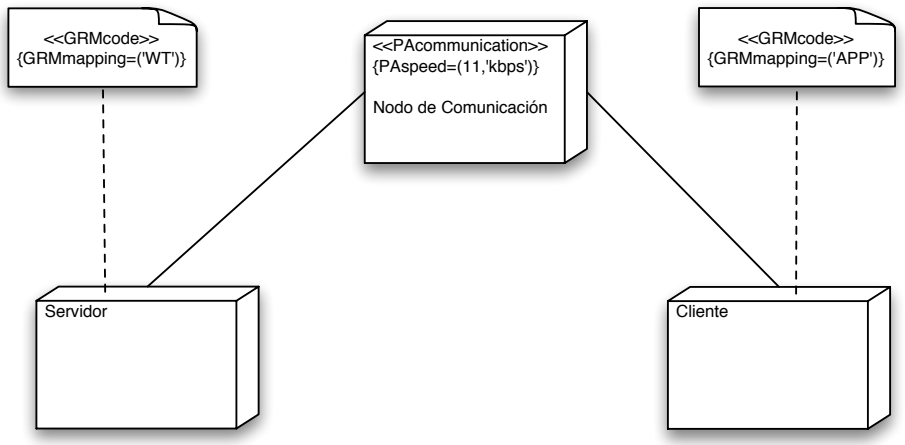

Figura D.7: Diagrama de Despliegue posible para WatchDog Timer.

El resto de elementos se anotarán utilizando el panel de propiedades del elemento que nos interesa y utilizando el botón para crear un estereotipo, representado en la figura.

Figura D.8: Botón para crear un estereotipo de un elemento en ArgoUML.

La anotación *número inicial de objetos* indicará el número de objetos vivos que tendremos al principio de la ejecución de nuestro sistema, de una clase determinada. Es obvio que el valor de la etiqueta *PApopulation* tendrá que ser un número entero.

Por último podremos indicar en cada máquina de estados cúal es el estado que inicia el comportamiento de una instancia, para esto utilizaremos el par *PAinitialCondition-PAinitialState*, el valor será `true` o `false`, precedidos del símbolo de dolar (`$`).

D.3. Interrogar el modelo

Dentro de nuestro módulo existen diversas consultas que podemos realizar sobre un modelo UML de las características de nuestro ejemplo, las que aparecen a continuación están implementadas en estos momentos en ArgoSPE.

- **Time in state**: nos indica el porcentaje de objetos en un cierto estado. Esto puede ser útil para detectar la saturación de un proceso software, el porcentaje que pasa un recurso sin estar ocupado, o cómo un agente comparte su ejecución entre diferentes tareas. El resultado de esta consulta será obtenido al dividir el número de objetos en el estado seleccionado, entre el número medio que pobla la clase. Si por ejemplo quisieramos calcular esta medida para el estado *count* del diagrama de estados de WT, el estado nos tendría que aparecer de la siguiente manera en ArgoUML:

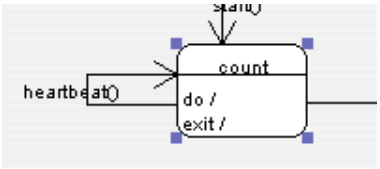

Figura D.9: Representación de un estado seleccionado.

- **Stay time**: mide el tiempo medio que los objetos de una clase específica invierten en cada uno de los estados. Podemos llegar a calcular el tiempo medio de ejecución de una acción compleja. Los cálculos son realizados aplicando la Ley de Little, por lo que es necesario dividir el número medio de objetos que están en media en el estado, entre el total de la tasa (throughput) de salida de ese estado. La consulta *Stay time* también necesita de la selección de un estado.

- **Response Time**: con esta consulta nos aparece el tiempo medio de respuesta de un escenario particular, es decir, la duración de una ejecución específica de nuestro sistema. El escenario será representado por un diagrama de colaboración, por lo tanto el resultado de esta consulta será el tiempo de respuesta del diagrama de colaboración.

  Antes de ejecutar la consulta deberá estar seleccionado en el panel del explorador de ArgoUML el diagrama de colaboración para el cual deseemos realizar los cálculos. La selección quedaría como se muestra en la figura:

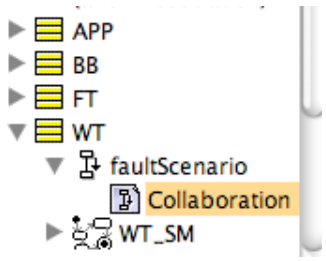

Figura D.10: Representación de la selección de un diagrama de colaboración.

- **Transmission speed**: es el retraso de la conexión de red entre dos elementos del modelo (nodos físicos, componentes o clases). Puede detectar cuellos de botella en los sistemas que estamos modelando. El cálculo de esta consulta se obtiene utilizando el algoritmo de Floyd, el cual encuentra el camino más corto entre dos vértices de un grafo. En nuestro caso, las distancias son interpretadas como velocidades y los vértices del grafo corresponden con los nodos del diagrama de despliegue.

  Para poder seleccionar dos de estos elementos (bien sean nodos o clases) tenemos que presionar el botón izquierdo del ratón mientras arrastramos el puntero, encerrando a los elementos en el rectángulo de selección que nos aparece. El resultado de la acción descrita aparece en la siguiente pantalla.

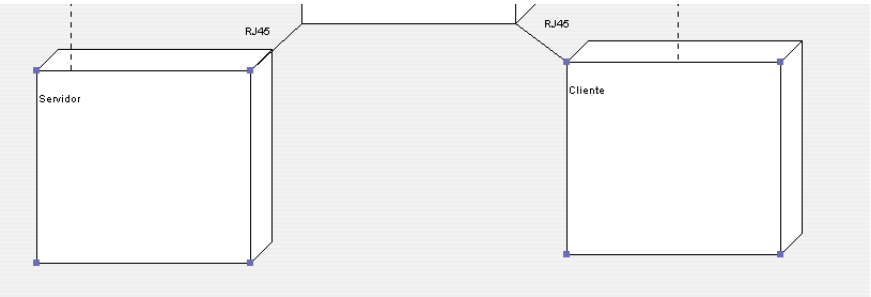

Figura D.11: Representación de dos nodos físicos seleccionados.

- **Message Delay**: esta consulta nos proporciona el retraso desde que un evento es llamado hasta que es recibido por la clase que lo estaba esperando. El emisor y el receptor tienen que residir en diferentes nodos físicos. La correcta ejecución de esta consulta implica la selección de un transición de una máquina de estados que posea un **disparador con un tamaño anotado**. En nuestro caso si desearamos calcular el retraso del evento *fault* de la máquina de estados de la aplicación de usuario, tendríamos que seleccionar la transición quedando algo como esto:

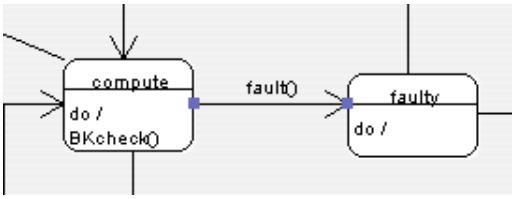

Figura D.12: Representación de una transición seleccionada con un disparador anotado.

Apéndice E

The GNU General Public License

Version 2, June 1991

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is

not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that

you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the

same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places

the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of Terms and Conditions

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Apéndice F

Glosario

Este capítulo es un compendio de algunos de los términos más relevantes que han aparecido durante el transcurso de nuestro proyecto, no todos aparecen explícitamente en este documento, pero sí forman parte de los documentos expuestos en las referencias bibliográficas y sin duda, ayuda a la mejor comprensión de ciertos aspectos.

Asociación es la relación semántica entre dos o más clasificadores que especifica conexiones entre sus instancias.

Enlace una conexión semántica entre una tupla de objetos; una instancia de una asociación.

Metamodelo Un modelo que define el lenguaje para expresar un modelo. Una instancia de un metametamodelo.

Metamodelo de UML define un número de elementos tales como Clases, Operación, Atributo, Asociación, etc. Estos elementos se llaman metaclases.

DTD un Document Type Definition define elementos de metadatos, su orden, estructura, reglas, y relaciones. Un DTD permite procesar automáticamente en una forma uniforme diferentes instancias de documentos del mismo tipo.

IDL Interface Definition Language. En CORBA la interfaz de un objeto es definida en IDL. Dicha definición especifica los métodos que el objeto está preparado para realizar, sus parámetros de entrada, su resultado y cualquier excepción que pueda generarse durante la ejecución. Toda la información necesaria para construir un cliente del objeto es proporcionada por la interfaz.

Modelo Una abstracción semánticamente consistente de un sistema.

Sistema Una colección de unidades conectadas entre sí, que están organizadas para llevar a cabo un propósito específico. Un sistema puede describirse mediante uno o más modelos, posiblemente desde puntos de vista distintos.

MOF Meta Object Facility es una especificación de la OMG para estandarizar la representación de repositorios de metadatos. Define como representar y manipular metadatos.

CORBA Common Object Request Broker Architecture es la especificación de la OMG para la interoperabilidad de objetos distribuidos.

CWM Common Warehouse Metamodel es un estándar para la representación de metadatos.

XMI XML Metadata Interchange. Es una especificación de la OMG basada en XML para el intercambio de metadatos . Permite el intercambio fácil de metadatos entre herramientas de modelado (basadas en UML) y repositorios de metadatos (basados en MOF) en ambientes heterogéneos distribuidos. XMI integra tres estándares XML, UML y MOF.

XML Extensible Markup Language es una especificación que define una forma estándar de agregar marcas a un documento es, en definitiva, un lenguaje para documentos que contienen información estructurada.

UML Unified Modeling Language es un lenguaje estándar definido por la OMG para análisis y diseño orientado a objetos.

Ciclo de vida (lifecycle) Período de tiempo que comienza con la concepción del producto de software y termina cuando el producto está disponible para su uso. Normalmente, el ciclo de vida del software incluye las fases de concepto, requisitos, diseño, implementación, prueba, instalación, verificación, validación, operación y mantenimiento, y, en ocasiones, retirada. Nota: Esta fases pueden superponerse o realizarse iterativamente.

Requisito (requirement) (1) Condición o facultad que necesita un usuario para resolver un problema. (2) Condición o facultad que debe poseer un sistema o un componente de un sistema para satisfacer una especificación, estándar, condición de contrato u otra formalidad impuesta documentalmente. (3) Documento que recoge (1) o (2).

OO (Orientación a Objetos) Enfoque para el desarrollo de sistemas de software que representa el dominio de aplicación de forma natural y directa 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. Las estrategias de orientación por objetos han desarrollado metodologías tanto para requisitos, como para análisis, diseño y programación.

Implementación (1) Proceso de transformación de un diseño en componentes de hardware, software o de ambos. (2) El resultado del proceso (1).

Diseño Proceso de definición de la arquitectura, componentes, interfaces y otras características de un sistema o de un componente.

Ingeniería del software (1) Aplicación de procesos sistemáticos y disciplinados para el desarrollo, operación y mantenimiento de software. (2) El estudio de la aplicación (1).

Red de Petri es una clase particular de grafo dirigido, junto con un estado inicial llamado marcado inicial M_0. 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 (lugar de entrada), o de una transición a un lugar (lugar de salida).

Bloqueo (*Deadlock*) Un conjunto de objetos está en *deadlock* si cada objeto del conjunto está con la obligacion de esperar la ocurrencia de una accion, cuyo cliente es otro objeto que también pertenece a dicho conjunto.

Alcanzabilidad Es una base fundamental para el estudio de las propiedades dinámicas de cualquier sistema. Una secuencia de disparos da lugar a una secuencia de marcados. Un marcado M_n se dice que es alcanzable desde un marcado M_0 si existe una secuencia de disparos que transforma M_0 en M_n. Al conjunto de todos los posibles marcados alcanzables desde M_0 en una red (N; M_0) se denota por R(N; M_0). El problema de la alcanzabilidad para las RdP es el problema de encontrar si ocurre $M_n \rightarrow$ R(N; M_0) para un marcado dado M_n en una red (N; M_0). Aunque el estudio de la alcanzabilidad es un problema decidible la complejidad de dicho cálculo es, en términos generales, muy elevada.

Acotamiento Una red (N; M_0) se dice que está k-acotada o simplemente acotada si el número de marcas en cada lugar no excede un número finito k para ningún marcado alcanzable desde M_0. Una red de Petri se dice que es segura si es 1-acotada.

Vivacidad Una RdP (N; M_0) se dice que está viva (o que M_0 es un marcado vivo para N) si sea cual sea el marcado alcanzado desde M_0, es posible disparar alguna transición desde él. La vivacidad es una propiedad ideal de muchos sistemas pero es impracticable su verificación. Por esta razón, se han definido diferentes niveles de vivacidad relajando esta condición. Este concepto está íntimamente relacionado con la ausencia completa de bloqueos en los sistemas operativos.

Persistencia Una RdP (N; M_0) es persistente si, para dos transiciones habilitadas cualquiera, el disparo de una de ellas no deshabilita a la otra. En una red persistente, una vez una transición está habilitada, se mantiene habilitada hasta que se dispara.

Bibliografía

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley Series in Parallel Computing - Chichester, 1995.

[2] The ArgoSPE project. `http://argospe.tigris.org`.

[3] The ArgoUML project. `http://argouml.tigris.org`.

[4] Borja Fernández Bacarizo. Evaluación del rendimiento del software: Traducción de UML (Máquinas de estados + Diagramas de actividades) a GSPN, September 2003.

[5] S. Bernardi, J. Campos, S. Donatelli, and J. Merseguer. GSPN Compositional Semantics for UML Statecharts and Sequence Diagrams. Research report, Departamento de Informática e Ingeniería de Sistemas, 2003. To be submmited to IEEE Transactions On Software Engineering.

[6] S. Bernardi, J. Campos, S. Donatelli, and J. Merseguer. Performance Evaluation of UML Models using GSPN. Research report, Departamento de Informática e Ingeniería de Sistemas, 2005. Submmited to IEEE Transactions On Software Engineering.

[7] Isaac Trigo Conde. Análisis automático de prestaciones de sistemas a partir de modelos en UML, mediante traducción a redes de Petri generalizadas, September 2004.

[8] DepAuDE EEC-IST project 2000-25434. http://www.depaude.org.

[9] The Dewey Decimal Classification, 1876. `http://www.oclc.org/dewey/`.

[10] Document Object Model. `http://www.w3.org/DOM/`.

[11] Document Type Definition. `http://www.xmlfiles.com/dtd/`.

[12] GNU's Not UNIX, 1984. `http://www.gnu.org`.

[13] E. Gómez-Martínez and J. Merseguer. A Software Performance Engineering Tool based on the UML-SPT. In *Proceedings of the 2nd International Conference on Quantitative Evaluation of Systems (QEST 2005)*, pages 247–248, Torino, Italy, September 19-22 2005. IEEE Computer Society.

[14] The GreatSPN tool. `http://www.di.unito.it/~greatspn`.

[15] J. Hillston and M. Ribaudo. Stochastic process algebras: a new approach to performance modeling. In K. Bagchi and G. Zobrist, editors, *Modeling and Simulation of Advanced Computer Systems*. Gordon Breach, 1998.

[16] J.P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic Petri nets: Application to software performance engineering. In J. Dujmovic, V. Almeida, and L. Doug, editors, *Proceedings of the Fourth International Workshop on Software and Performance (WOSP'04)*, pages 25–36, Redwood City, California, USA, January 2004. ACM. ISBN 1-58113-673-0.

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

[18] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A compositional semantics for UML state machines aimed at performance evaluation. In A. Giua and M. Silva, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer Society Press.

[19] Michael K. Molloy. *Fundamentals of Performance Modeling*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1988.

[20] NSUML, 2002. `http://nsuml.sourceforge.net/`.

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

[22] OMG. *UML Profile for Schedulabibity, Performance and Time Specification*, January 2005. `http://www.omg.org/technology/documents/formal/schedulability.htm`.

[23] Petri Nets Markup Language. `http://www.informatik.hu-berlin.de/top/pnml/about.html`.

[24] R.S. Pressman. *Ingeniería del Software. Un enfoque práctico.* Mc Graw Hill, 2002.

[25] Rational Rose. `http://www-306.ibm.com/software/awdtools/developer/rose/`.

[26] M. Silva. *Las Redes de Petri en la Automática y la Informática*. Editorial AC, Madrid, 1985. In Spanish.

[27] Stream-Based Model Interchange Format. `ftp://ftp.omg.org/pub/docs/ad/97-12-03.pdf`.

[28] Connie U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[29] Tigris open source community. `http://www.tigris.org`.

[30] The World Wide Web Consortium. `www.w3.org`.

[31] Extensible Markup Language. `http://www.w3.org/XML/`.