UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Análisis de rendimiento de diagramas de secuencia mediante redes Petri

Luis Enrique Abarca Sánchez
Tutor: María Elena Gómez Martínez
Ponente: Juan De Lara Jaramillo

Junio 2021

ANÁLISIS DE RENDIMIENTO DE DIAGRAMAS DE SECUENCIA MEDIANTE REDES DE PETRI

AUTOR: Luis Enrique Abarca Sánchez
TUTOR: María Elena Gómez Martínez

MISO
Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2021

Resumen

Este Trabajo Fin de Grado implementa un mecanismo para analizar rendimiento en fases tempranas en el desarrollo software mediante técnicas de Ingeniería de Prestaciones Software (en inglés, SPE, Software Performance Engineering). Este tipo de desarrollo software se centra en el análisis de comportamiento dinámico de los sistemas software para obtener datos de su rendimiento. Esto se hace muy difícil con el uso generalizado del estándar UML (Unified Modelling Language) propuesto por OMG (Object Management Group).

UML trata de estandarizar los diagramas del desarrollo software. Estos diagramas permiten un nivel de abstracción que proporciona una herramienta perfecta para el diseño de software. Sin embargo, estos diagramas son semi formales. Los diagramas semiformales son aquellos que nos permiten representar el software de forma más semejante al código, pero conserva un nivel de abstracción suficiente como para ser entendidos con facilidad tanto por personal cualificado como no cualificado. Estos diagramas por tanto están pensados para facilitar la comprensión y no tanto como para usarlos de forma activa en un análisis de rendimiento. Por esa razón, se han propuesto perfiles UML, como MARTE, que permiten añadir a estos diagramas información que posibilitan el análisis de rendimiento.

En el caso de MARTE, proporciona unas estructuras de datos sobre el funcionamiento útiles para los análisis de prestaciones. A pesar de ello, la limitación que tenemos con UML y un perfil como MARTE unidos es que, por mucho que permitan la representación de parámetros para la mejor visualización y representación del diseño, no permiten hacer un análisis automático.

Por ello, este TFG trata de cómo utilizar los diagramas UML, tan comúnmente utilizados hoy en día, para poder hacer un análisis de rendimiento que siente las bases de un diseño más eficiente desde una fase temprana del desarrollo. Para lograr esto, se harán uso de las propiedades de las SPN (Stochastic Petri Nets).

Este tipo particular de grafos dirigidos permite representar sistemas temporizados ya que son un caso particular de las cadenas de Márkov. Esta característica hace ideales a las SPN para representar sistemas software, ya sean distribuidos, paralelos o concurrentes. Esta formalización, por tanto, aparte de permitirnos representar un sistema software, nos permite además analizar su comportamiento dinámico.

Por ello, para hacer un análisis de rendimiento de un diagrama UML se necesita transformarlo a una red de Petri. Este TFG trata uno de los diagramas dinámicos más utilizados que es el diagrama de secuencia. Estos diagramas sirven para representar las comunicaciones entre los distintos módulos del sistema a lo largo del tiempo. Por tanto, es muy útil el análisis de un diagrama de secuencia para conocer el comportamiento dinámico de un sistema.

Por último, una herramienta muy extendida para el desarrollo software es Eclipse, por ello, en este TFG se muestra el desarrollo de un *plugin* para Eclipse que analiza el rendimiento de los diagramas de secuencia por medio de una transformación a redes Petri.

Palabras clave

SPE, SPN, UML, OMG, Red de Petri Estocástica, Diagrama de Secuencia, MARTE, Análisis de Rendimiento.

Abstract

This Bachelor Thesis implements a mechanism to analyze performance in the early stages of software development by means of Software Performance Engineering (SPE) technics. This type of software development focuses on the dynamic behavior analysis of the software systems to obtain performance data. This is very difficult due to the generalized use of the UML (Unified Modelling Language) standard proposed by the OMG (Object Management Group).

UML tries to standardize the software development diagrams. These diagrams enable a level of abstraction that provides a perfect tool for software design. Nevertheless, these diagrams are semi-formal. The semi-formal diagrams are those that enable us to represent software in more similar way to the code, although it preserves enough abstraction level so as to be understood by qualified personal and not qualified personal. Therefore, these diagrams are intended to facilitate the understanding rather than to be used in performance analysis. For this reason, profiles of UML, like MARTE have been proposed, that permit to add to these diagrams information that enable us to do performance analysis.

In the case of MARTE, it provides of data structures about the functioning that are useful to do performance analysis. In spite of that, the limitation that we have with UML and a profile such as MARTE together, is that although they allow the representation of the parameters for a better representation and visualization of the design, they do not allow to do an automatic analysis.

Therefore, this Bachelor Thesis is about how use the UML diagrams, so commonly used nowadays, in order to do performance analysis that lays the foundations of a more efficient design since an early stage of development. To achieve this, it takes advantage of SPN (Stochastic Petri Nets).

This particular type of directed graphs allow to represent temporized systems because they are a particular case of Markov chain. This characteristic makes the SPN ideal for representing software systems either distributed, parallel or concurrent. Therefore, this formalization, apart from allowing to represent a software system, also allows to analyze its dynamic behavior. Hence, to do a performance analysis of an UML diagram, it is only needed to be transformed to a Petri Net.

This Bachelor Thesis is about one of the most used dynamic diagrams, the sequence diagram. These diagrams are used to represent the communications between the different modules of the system over time. Therefore, the analysis of a sequence diagram is very useful to know the dynamic behavior of a system.

Lastly, a very extended tool for software development is Eclipse, thus in this Bachelor Thesis is shown the development of an Eclipse *plugin* that analyze the performance of the sequence diagrams by means of a transformation to Petri Nets.

Keywords

SPE, SPN, UML, OMG, Stochastic Petri Net, Sequence Diagram, MARTE, Performance Analysis.

Agradecimientos

Me gustaría agradecer a mi tutora María Elena Gómez Martínez toda la ayuda incesante desde el principio hasta el final y su incansable dedicación a la informática y a la enseñanza. Me gustaría agradecer a mi familia por mantenerme y aguantarme en todo momento. Por último, se lo agradezco a mi pareja, que por encima de todo siempre me ha apoyado en los momentos difíciles y siempre ha confiado en mí y en mis capacidades.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 *Motivación*

El desarrollo software ha cambiado mucho a lo largo de los años. Desde sus inicios, con un diseño apenas existente a las nuevas técnicas de diseño orientadas a modelos, con unas fases de diseño que cubren la mayoría del desarrollo.

Estas nuevas técnicas de desarrollo nos han permitido crear un software mucho más rápido, complejo y consistente. Le tenemos que dar las gracias en gran parte a los diagramas y sobre todo a UML (Unified Modelling Language) que logró estandarizarlos. Estos diagramas sirven para abstraer la complejidad del software permitiendo diseñar software a un nivel más alto. Esto ocasiona por un lado el punto positivo que es la facilidad y agilidad de diseño, pero, por otro lado, la abstracción de los diagramas con respecto al código aleja un poco la vista del funcionamiento interno, el cual se realiza en una fase posterior de desarrollo, por lo tanto, es importante prestar atención en la parte de diseño al desempeño que tendrá el software para evitar futuros problemas de rendimiento. Por ello, son necesarias herramientas que nos permitan tener en cuenta este tipo de factores como es el rendimiento, ya que un fallo de rendimiento descubierto en la parte de desarrollo o en la de pruebas, supone un coste mucho mayor que si se encontrase en la etapa de diseño.

Con este TFG se trata de proporcionar una herramienta para ayudar en esta causa ya que se considera vital un buen rendimiento en el desarrollo de software actualmente. También, se ha querido integrar en Eclipse para facilitar su instalación y su uso, debido a que se integra en una herramienta muy ampliamente extendida en el mundo del desarrollo software.

En resumidas cuentas, la motivación de la realización de este proyecto es a la de aportar a la comunidad de desarrollo de software una herramienta actualizada e integrada en Eclipse para ayudar en el desarrollo de un software de alto rendimiento para proyectos basados en modelos.

1.2 *Objetivos*

El objetivo principal del proyecto es crear un *plugin* para Eclipse que contenga la funcionalidad para poder analizar el rendimiento de un diseño de software por medio de los diagramas de secuencia UML descritas en EMF mediante transformaciones a redes de Petri. Los objetivos específicos para lograrlo son:

- Crear un módulo que transforme un diagrama de secuencia de EMF en una red de Petri en PNML con ATL.
- Crear un módulo que transforme una red de Petri en PNML a una red de Petri en PNPRO con Acceleo.
- Hacer uso de GreatSPN para poder visualizar y analizar la red de Petri generada.

1.3 *Estructura del documento*

La memoria de este TFG consta de ocho capítulos:

- En el capítulo 1 se explican las motivaciones y objetivos del proyecto, así como la organización del documento.

- En el capítulo 2 se explican los conceptos necesarios para entender el proyecto. En esos conceptos necesarios se incluyen: unas nociones básicas de UML profundizando en los diagramas de secuencia y se comenta un perfil UML llamado MARTE. Se hace una introducción a las redes de Petri y en concreto a las redes de Petri Estocásticas. Además, se

explica que es la ingeniería orientada a modelos. Por último, se explica la herramienta de Eclipse EMF.

- En el capítulo 3 se exponen los trabajos previos más relevantes del ámbito que se trata en este TFG. Primero se comentan otros sistemas dinámicos de representación diferentes a las redes de Petri. Después se expondrán los trabajos previos acerca de las transformaciones UML a redes de Petri. Por último, se explican las herramientas previas para análisis de rendimiento, tanto en Eclipse como fuera de Eclipse.

- En el capítulo 4 se comentan los aspectos del proyecto que tienen que ver con la fase de análisis de un proceso software. Primero se describe brevemente el software a desarrollar, después, se explica la metodología de trabajo en Cascada, y se explica porque se utiliza, luego se muestra una lista de requisitos tanto funcionales como no funcionales. Por último, se comentan las entradas y salidas del programa y los aspectos fundamentales de estas.

- En el capítulo 5 se expone la información referente a la etapa de diseño en un desarrollo software. Primero se explican las herramientas utilizadas en el desarrollo, tanto dentro de Eclipse como una herramienta externa. Luego, se muestra el diseño de las transformaciones que se han desarrollado en esta etapa del proceso software. Por último, se muestran los diagramas creados para el diseño del software.

- En el capítulo 6 se expone la etapa de desarrollo en un proceso software, esto incluye toda la información acerca de cómo se ha implementado el software y toda su funcionalidad. Primero se muestra la arquitectura de ficheros que tiene el proyecto. A continuación, se comenta por separado los aspectos fundamentales de la implementación de la transformación de ATL. Por último, se comenta la transformación de Acceleo.

- En el capítulo 7 se muestran las pruebas realizadas para comprobar tanto el correcto funcionamiento como que se han cumplido los requisitos correctamente. Por último, se muestra la creación del plugin de Eclipse y su integración con dicho entorno de trabajo.

- En el capítulo 8 se explican las conclusiones del TFG junto con lo que se ha aprendido y las opiniones de las herramientas utilizadas y del trabajo realizado. Para terminar, se comentan los aspectos que deja abiertos este TFG para poder realizar un trabajo futuro sobre él.

2 Conceptos previos

A continuación, se describen brevemente una serie de conocimientos previos necesarios para el desarrollo del proyecto. Primero hablaremos de UML ya que es el sistema de representación utilizado para los diagramas de secuencia en el proyecto. A continuación, se abordarán las partes principales de los diagramas y su utilidad.

Después, se describirá un perfil UML llamado MARTE, que se encargará de otorgarle la información necesaria para hacer análisis de rendimiento.

Luego se explican las redes de Petri, este es el modelo de análisis elegido para este proyecto. Se explica el funcionamiento principal y un resumen de sus características. También se habla más específicamente de las Redes de Petri Estocásticas, las cuales son las que usaremos concretamente, ya que nos permiten modelar el tiempo.

Por último, se hará una introducción a la ingeniería orientada a modelos. Este es un paradigma de desarrollo en el que se centra el desarrollo software en modelos. Esto se explica para saber qué es y para qué sirve, ya que el *plugin* a desarrollar sirve para beneficiar a este tipo de desarrollo. Y para ayudar en este tipo de desarrollo, existen herramientas de modelado como EMF de Eclipse, la cual también se usará en este proyecto.

2.1 *UML*

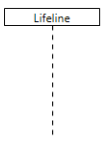
UML (*Unified Modelling Language*) o lenguaje unificado de modelado [1], es un lenguaje creado para estandarizar el modelado visual de proyectos software. Una de las características principales de UML es la orientación a objetos y esto es debido a dos motivos principales, por un lado, tenemos la incipiente expansión del paradigma orientado a objetos en la década de los 90, por otro lado, tenemos la cercanía de este concepto con el mundo real, ya que, en un lenguaje orientado a objetos, se representan objetos reales los cuales por medio de unos algoritmos se relacionan entre ellos.

El estándar UML consta de trece diagramas que se pueden dividir principalmente en dos tipos, los diagramas estructurales y los diagramas de comportamiento. Como su nombre indica, los diagramas estructurales sirven para representar aspectos estáticos o estructurales de un sistema, por otro lado, los diagramas de comportamiento representan los aspectos dinámicos y de uso de un sistema. Estos dos tipos de diagramas se complementan entre sí. Por ejemplo, un diagrama de secuencia suele complementarse con un diagrama de caso de uso y con un diagrama de clase. El primero es el que muestra el caso de uso especifico y el diagrama de clases es el que representa las diferentes clases que forman el sistema. La gran diferencia entre un diagrama de casos de uso y un diagrama de secuencia es que el diagrama de secuencia representa la temporalidad de los mensajes visualmente a lo largo del eje vertical.

2.1.1 Diagramas de secuencia

Los diagramas de secuencia son unos de los diagramas más utilizados dentro del estándar UML, estos diagramas sirven para representar el intercambio de mensajes entre los diferentes componentes un sistema.

Los componentes principales de un diagrama de secuencia son los siguientes [2]:

Lifeline	Representa cada uno de los objetos o clases que interactúan en el diagrama de secuencia.	

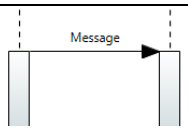
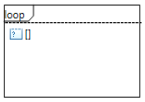
Message	Representa el envío de un mensaje de un objeto o clase a otro. Hay distintos tipos de mensajes que se representan con diferentes tipos de flechas.	
ExecutionSpecification	Representa una acción atómica ejecutada por el *Lifeline* en el que se encuentra.	
CombinedFragment	Se representa con un cuadro con diferentes etiquetas en la esquina izquierda superior que denota la funcionalidad que queremos representar. Pueden ser anidados y pueden definirse varios operadores en el interior por medio de una línea discontinua.	
InteractionUse	Se representa con un cuadro parecido al de *CombinedFragment* pero con la etiqueta *Ref.* Su uso es hacer referencia a otro diagrama.	

Tabla 2.1: Elementos principales de los diagramas de secuencia en UML. Fuente propia

Los *CombinedFragment* tienen varios tipos dependiendo de su funcionalidad, los más utilizados son las siguientes.

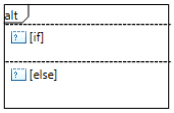
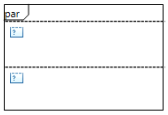
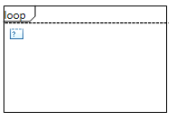
Alternative	Su etiqueta es *alt* y representa una cláusula *if*, se pueden añadir *elsif* y *else* añadiendo más *InteractionOperand* dentro del *CombinedFragment*.	
Parallel	Su etiqueta es *par* y representa unas partes de código a ejecutar en paralelo, cada zona paralela viene separada por una línea discontinua denotando cada *InteractionOperand*.	
Loop	Su etiqueta es *loop* y representa un bucle.	

Tabla 2.2: Tipos principales de *CombinedFragment*. Fuente propia

2.1.2 MARTE

Los diagramas de secuencia, como se ha contado en el apartado anterior, son útiles para la representación de las comunicaciones entre módulos a lo largo del tiempo. Por consiguiente, nos da una idea del comportamiento dinámico del sistema, el problema es que los diagramas de secuencia de UML no disponen de la información necesaria para realizar un análisis matemático para obtener datos de rendimiento, y aquí es donde entra MARTE.

MARTE es un perfil UML que nos permite, como los demás perfiles de los modelos de UML, ampliar el modelo manteniendo el original, con el objetivo de conseguir una funcionalidad no contemplada por UML, pero manteniendo la compatibilidad con otras funcionalidades de UML.

Este perfil en concreto nos permite añadir información sobre el rendimiento. Estos parámetros van desde parámetros generales como puede ser un tiempo de ejecución hasta algo más específico de los diagramas de secuencia como la probabilidad de ejecución de cada una de las opciones en un *Alternative*. El perfil MARTE es muy completo, por tanto, los elementos que se usarán en este proyecto se comentarán en la parte de diseño, cuando haya una idea de la funcionalidad a implementar.

2.2 ***Redes de Petri***

Una red de Petri es un grafo dirigido con dos tipos de nodos, esos dos tipos de nodos son **Lugares** y **Transiciones** que son unidos por A**rcos** [3]. La principal particularidad de estas redes con respecto a estos dos tipos de nodos es que no se pueden unir dos nodos del mismo tipo directamente. Por tanto, las uniones siempre serán de un lugar a una transición o viceversa. El último elemento de estos diagramas es el **Token**. Los tokens son las unidades que recorren la red de Petri. Se representan como puntos en los estados y que se mueven entre ellos a través de los arcos y las transiciones.

Como podemos observar en el ejemplo de la Figura 2.1, para que un token en un estado *P0* pueda llegar a un estado *P1*, tiene que pasar por la transición *T*.

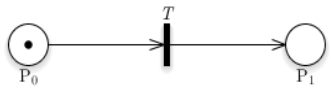


Figura 2.1: Ejemplo sencillo de red de Petri. Fuente propia.

Es importante saber, que como su nombre indica, un nodo de *Transición* es simplemente de transición, no se detiene en ningún momento la ejecución en esos nodos. Estos nodos sirven por tanto como control de paso o de flujo entre los diferentes estados. Este control se puede llevar a cabo por medio del número de arcos tanto de entrada como de salida.

Por un lado, tenemos la posibilidad de tener varios arcos de entrada a una transición. Está estructura sirve como sincronización entre dos estados, ya que es necesario que todos los estados origen de los arcos de entrada de una transición han de estar cubiertos para que esta transición pueda avanzar.

Podemos observar la Figura 2.2 en la que se ve como hay tres arcos de entrada a la transición *T*, el problema es que como no están cubiertos los tres estados, la transición no avanzará. En cambio, en la Figura 2.3 podemos ver que al estar cubiertos los estados de los tres arcos de entrada, la transición avanza.

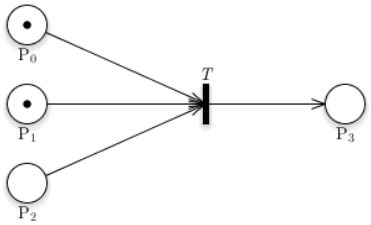


Figura 2.2: Transición no activada con varios arcos de entrada. Fuente propia.

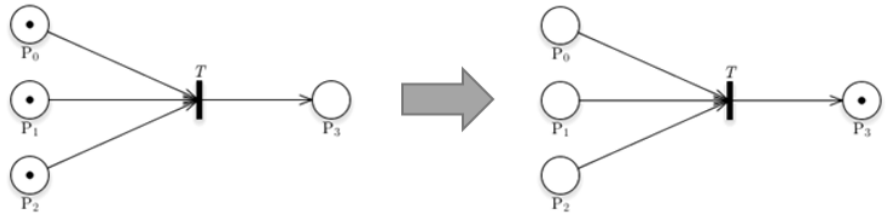


Figura 2.3: Transición activada con varios arcos de entrada. Fuente propia.

Por otro lado, tenemos la posibilidad de tener varios arcos de salida de una transición. Esta estructura sirve para indicar que el flujo de ejecución se divide y por tanto se crea un camino nuevo por cada arco de salida de la transición. Como podemos observar en la Figura 2.4 la ejecución proveniente de P0 se divide entre P1, P2 y P3.

Figura 2.4: Transición con varios arcos de salida. Fuente propia.

Gráficamente lo que observamos es que, al avanzar la transición, avanza en todos los arcos de salida, por tanto, se multiplican las ejecuciones por el número de arcos de salida, al igual que en el caso anterior de la sincronización se divide el número de ejecuciones entre el total de arcos de entrada. Por tanto, podemos decir que varias entradas necesitan ser sincronizadas y varias salidas creadas de forma sincronizada al salir de la transición.

2.2.1 Redes de Petri Estocásticas

Las redes de Petri Estocásticas son una especialización de las redes de Petri. Las redes de Petri Estocásticas tienen la característica de que cada transición sucede tras un retraso probabilístico determinado por una variable aleatoria. También tienen una equivalencia directa con las cadenas de Márkov debido a que cumplen la propiedad de Márkov.

La propiedad de Márkov se define como la fuerte dependencia que hay entre un suceso y el anterior [4]. Esto quiere decir que cada estado del proceso solo depende del anterior, por lo tanto, la probabilidad del salto siguiente solo depende de la variable aleatoria propia de ese estado. Esto permite modelizar las redes de Petri Estocásticas en este caso y a las cadenas de Márkov en general como un conjunto de variables aleatorias independientes. Gracias a esto, las redes de Petri son muy útiles al servirnos para el estudio de un sistema por medio de sencillos estudios probabilísticos. En nuestro caso, la transformación de un diagrama de secuencia de UML a una red de Petri Estocástica nos permite hacer cálculos de rendimiento y desempeño directamente, acción que los diagramas de secuencia de UML no contempla.

2.3 *Ingeniería orientada a modelos*

La ingeniería orientada a modelos o ingeniería dirigida por modelos (MDE del inglés *Model Driven Engineering*) es un paradigma del desarrollo de software que trata de enfocar el desarrollo de software en modelos que permiten representar los problemas que se han de abordar con el software independientemente del lenguaje de programación [5]. Con esto conseguimos elevar el nivel de abstracción en el diseño software, consiguiendo así por un lado desarraigar el diseño de software del lenguaje de programación y por otro lado agilizar el proceso de análisis y diseño [6]. Por su fácil

utilización y por su fácil comprensión por personal no especializado beneficia a la comunicación con el cliente y, por tanto, mejora también la calidad del software.

Esta forma de desarrollar software también mejora la escalabilidad del software ya que permite modularizar, lo que permite la ampliación más sencilla. Además, otro beneficio del diseño desarraigado del lenguaje a utilizar beneficia a la integración con otro software. En este caso usaremos EMF (*Eclipse Modeling Framework*) que es una de las implementaciones que se ofrecen en Eclipse de MDE.

2.4 ***Eclipse (EMF)***

EMF es un *framework* (entorno de trabajo) de modelado y generación de código para creación de herramientas y otras aplicaciones basadas en un modelo de datos estructurado. Desde la especificación de un modelo descrito en XMI, EMF proporciona herramientas y soporte en tiempo de ejecución para producir un conjunto de clases java para el modelo, junto con un conjunto de adaptadores que permiten la visualización y edición de los modelos basada en comandos y un editor básico [21].

3 Estado del arte

En este apartado se exponen los trabajos previos más reseñables en el ámbito que incumbe a este proyecto. Primero se comentan otros sistemas dinámicos de representación a parte de las redes de Petri. A continuación, se comenta trabajos previos en la transformación de diagramas UML a redes de Petri. Por último, se comentan las herramientas que se consideran más relevantes en el ámbito del análisis de rendimiento.

3.1 *Otros sistemas dinámicos de representación*

A parte de las redes de Petri existen otros sistemas dinámicos que sirven para representar acciones en el tiempo, las más importantes son las siguientes.

Las cadenas de Márkov o procesos de Márkov son un modelo estocástico formado por estados y en cada momento del tiempo la probabilidad de un estado solo depende del anterior. Por tanto, la probabilidad de que el estado $X_{t+1} = i$ solo depende de X_t [11]. Esta característica permite hacer tanto estudios probabilísticos como simulaciones para el estudio del comportamiento de un sistema.

Layered Queueing Networks (LQN) son un modelo estocástico de representación que esencialmente está compuesto por nodos los cuales contienen colas de espera [10]. La diferencia entre una Layered Queueing Network y una Queueing Network es que las LQN permiten que un nodo contenga otra red la cual represente el sistema interno de un componente en concreto, permitiendo así representar un sistema más complejo pudiendo estudiar tanto los componentes por separado como en conjunto. Estas redes se basan en la teoría de colas, la cual es una disciplina dentro de la teoría de la probabilidad. Esto permite el estudio tanto de tiempos de espera como de los puntos de saturación por medio de modelos probabilísticos.

Un modelo también bastante extendido para usos como el de modelado de procesos es el de PEPA (Performance Evaluation Process Algebra). “Este lenguaje ha sido desarrollado para investigar cómo las características compositivas del álgebra de procesos pueden impactar en la práctica de modelado de rendimiento” [22]. La ventaja de este modelo es que se basa en un proceso estocástico que permite modelarse como una cadena de Márkov de tiempo continuo, lo que permite usarlo para obtener datos de rendimiento. A la misma vez, mantiene gran parte de las características del algebra de procesos. Estas dos características, lo hacen una herramienta muy interesante, ágil y completa para el SPE.

3.2 *Transformación de UML a redes de Petri*

Hay bastantes estudios acerca de la transformación de UML a redes de Petri, la mayoría de esos estudios explican desde un principio las limitaciones de UML, hablando de su informalidad, como de la inviabilidad de hacer estudios de rendimiento tan solo con UML. A continuación, enumero los que me han parecido más interesantes.

Uno de los estudios consultados [7] habla de la limitación de UML en el análisis dinámico del software, expresa el cómo UML en la gran mayoría de veces se usa simplemente con carácter documental y no se usa en la parte de análisis. Por ello, se ve necesario la transformación de los diagramas UML a redes de Petri, con el objetivo de rentabilizar la creación de estos diagramas y favorece al modelo de desarrollo MDE.

Por otro lado, otro de los estudios consultados que resultan interesantes [8] habla de cómo a día de hoy ha cobrado mucho interés en el desarrollo de software lo que es llamada SPE (Software Performance Engineering) que trata de tener en cuenta el rendimiento del software como parte principal del desarrollo desde una fase muy temprana del desarrollo, lo cual, debido a las limitaciones intrínsecas de UML esto no lo hace posible, por tanto la transformación a un sistema que sí que lo permita es fundamental.

Hay un estudio interesante [9], acerca de la transformación de modelos de secuencia en redes de Petri que es justamente la funcionalidad que tratamos en este trabajo. La diferencia con respecto a este trabajo es que las redes de Petri utilizadas en [9] son las CPN (Coloured Petri Nets) a diferencia de este trabajo en el que usamos SPN (Stochastic Petri Nets). Se ha decidido usar SPN ya que permite representar el tiempo de forma orgánica y así hacer un análisis de rendimiento más exhaustivo.

3.3 *Herramientas previas para análisis de rendimiento*

En esta sección se hace una lista de las herramientas que han sido creadas previamente enfocadas al análisis de rendimiento. Primero las que se han realizado dentro de Eclipse, como lo que se desarrolla en este proyecto, y luego, una herramienta fuera de Eclipse.

3.3.1 Basadas en Eclipse

En Eclipse existen diversas herramientas para análisis de rendimiento, pero la mayor parte de ellas están enfocadas en el estudio de rendimiento sobre un código de software, como son Eclipse TPTP (Test Performance Tools Platform) [12] o Trace Compass [13]. Lo que limita a estas a aplicaciones, es que, al usarse sobre código ya implementado, hace que el diseño orientado a rendimiento no sea posible causando por tanto que un problema de rendimiento cueste mucho más de reparar de haber sido localizado en el diseño.

Por otro lado, una herramienta muy utilizada en Eclipse para el análisis de rendimiento en fase de diseño es el *plugin* de PEPA [14]. Esta, es una herramienta integrada en el *framework* de Eclipse, que usa el lenguaje de PEPA, el que se explica en el apartado sobre sistemas dinámicos de representación, para hacer un modelo de alto nivel del diseño del software, permitiéndolo hacer desde un momento muy precoz del proceso software, esto permite hacer un diseño orientado al rendimiento y poder solucionar esos problemas desde un principio sin coste extra. Una de las propiedades más versátiles del *plugin* de PEPA y por lo que lo hacen tan popular es por el uso tanto de CTMC (Continuous-Time Markov Chain) para la representación y simulación, como de ODEs (Ordinary Differential Equations) lo que facilita el estudio matemático del rendimiento.

3.3.2 Fuera de Eclipse

Un ejemplo de herramientas de análisis de rendimiento fuera de Eclipse es ArgoSPE [15]. Esta herramienta nace del SPE (Software Performance Engineering) que propone métodos de evaluación de rendimiento en las fases más tempranas de desarrollo. Por otro lado, teniendo en cuenta que UML es un estándar para la ingeniería software, se ha optado por proveer a UML de parámetros para el análisis de software.

Basados en las premisas anteriores, se desarrolló esta herramienta con el fin de sacar el máximo provecho al SPE. ArgoSPE es una herramienta que sirve para hacer análisis de rendimiento desde diagramas UML transformándolos en SPN (Stochastic Petri Nets) permitiendo así aprovechar las ventajas de la abstracción de UML y de la capacidad de análisis de las redes de Petri. Las diferencias y principales desventajas frente a lo que se pretende desarrollar en este TFG es que no se hizo uso de MDE, por lo que la estructura de programa es mejorable y, además, la integración en proyectos resulta más difícil y menos eficiente debido a que no hace uso de Eclipse, sino que es una herramienta externa. Con respecto a la practicidad, es menos práctico ya que no dispone de MARTE por lo que se echa en falta todas las características de un perfil estándar como MARTE.

4 Análisis

En este apartado se comentan los aspectos del proyecto que tienen que ver con la fase de análisis de un proceso software. Primero se hace una breve descripción del software a desarrollar, a continuación, se explica la metodología de trabajo a utilizar, luego se hace una lista de requisitos tanto funcionales como no funcionales. Por último, se comentan las entradas y salidas del programa y los aspectos fundamentales de las mismas.

4.1 *Descripción*

El objetivo de este proyecto crear un *plugin* para Eclipse que permita la transformación de las funcionalidades principales de los diagramas de secuencia a una formalización de redes Petri, con el objetivo de poder hacer análisis de rendimiento en la fase de diseño del proceso software. Los diagramas de secuencia serán los creados con EMF de Eclipse y las redes Petri deberán poder ser abiertas ya analizadas con programas de análisis como es GreatSPN en este caso.

En la Figura 4.1 podemos observar que solo existe un caso de uso para el usuario que es el análisis de prestaciones de un diagrama de secuencia, el cual incluye la transformación de modelo de diagrama de secuencia en EMF a red de Petri en EMF, transformación de formato de red de Petri en EMF a red de Petri en PNPRO y por último una llamada a GreatSPN.

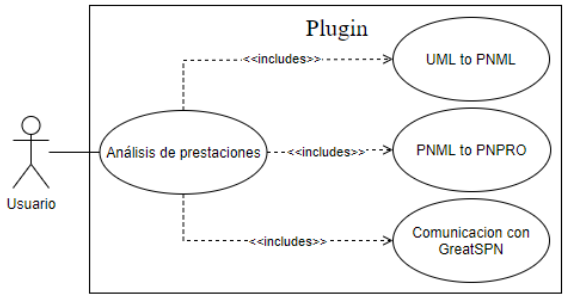


Figura 4.1: Diagrama de casos de uso del *plugin* a desarrollar. Fuente propia.

4.2 *Metodología de trabajo (Cascada Clásico)*

En este proyecto se ha optado usar como metodología de trabajo el modelo en cascada clásico. Este es un modelo primitivo que viene de otras disciplinas y que se adopta en los primeros años del desarrollo software [16].

Este modelo dispone de varias fases de desarrollo como se puede observar en la Figura 4.2, cada fase empieza cuando acaba por completo la anterior, esta disposición le confiere unas características propias que como veremos más adelante lo hacen ideal para este proyecto.

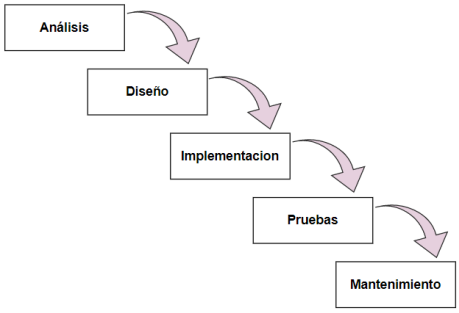


Figura 4.2: Diagrama de fases del método de trabajo de cascada clásico.

4.2.1 Ventajas, desventajas y conclusión

Las ventajas principales de esta metodología son la estabilidad y consistencia. Por un lado, el sistema creado a fases cerradas una tras otra genera que el proceso sea más consistente no permitiendo dar pasos atrás, lo que resultaría en una gran pérdida de tiempo y a tener que rehacer las cosas. Por otro lado, tener claros los requisitos desde un principio genera una fase de diseño solida que permite pulir errores y tener una idea más precisa de lo que será el producto final, por tanto, permite construir el software en la parte de implementación mucho más fácil, lo que genera un programa muy estable y fiable.

Las desventajas, principalmente vienen dadas por la complejidad del software o porque no está claro el producto a desarrollar. Si el cliente no tiene claro exactamente de lo que quiere genera que se le vayan ocurriendo requisitos según va avanzando el desarrollo del software, por tanto, no es adecuado en un sistema con fases definidas y lineal como este. Por otro lado, una alta complejidad del software origina que haya veces que no sea posible diseñar el software a la primera lo que ocasionaría en ese modelo que nos encontraríamos en la fase de implementación con errores de diseño, lo cual retrasaría mucho el desarrollo y descendería la calidad del software producido.

En conclusión, esta metodología está bien por su agilidad y estabilidad gracias a un sistema por fases ordenado y riguroso, pero con limitaciones en cuestión de complejidad y de claridad en ideas del cliente. En este caso, los requisitos están claros desde el principio y la complejidad del software no llega a ser abrumadora como para necesitar de más iteraciones para su diseño, por tanto, el modelo de cascada es el modelo indicado.

4.3 *Catálogo de Requisitos*

A continuación, se enumeran los requisitos formalizados catalogados en funcionales y no funcionales.

4.3.1 Funcionales

RF1. El usuario podrá hacer análisis de prestaciones de un diagrama de secuencia anotado con MARTE.

RF1.1. Se soportarán los siguientes elementos del diagrama de secuencia:

1. Envío y recepción de mensajes de los diagramas de secuencia.
2. Elemento *Reference* de los diagramas de secuencia.
3. Elemento *Parallel* de los diagramas de secuencia.
4. Elemento *Loop* de los diagramas de secuencia.
5. Elemento *Alternative* de los diagramas de secuencia.
6. La anidación de los *CombinedFragment*.

RF1.2. Se soportarán los siguientes elementos de MARTE:

1. <<GaWorkloadEvent>>{pattern}
2. <<GaStep>>{hostDemmand}
3. <<GaStep>>{prob}
4. <<GaStep>>{rep}

RF1.3. El análisis se realizará mediante transformación a redes de Petri y análisis de estas.

4.3.2 No funcionales

- De soporte

RNF1. El software desarrollado será un *plugin* de Eclipse.

RNF1.1. El *plugin* tendrá unas ventanas de interacción en las que se introducirán los datos y se obtendrán los resultados del análisis.

RNF2. Se usará ATL para realizar una transformación de UML a PNML.

RNF3. Se usará Acceleo para realizar una transformación de PNML a PNPRO.

RNF4. Se usará GreatSPN para realizar el análisis de redes de Petri.

- De documentación
  RNF5. Toda la documentación se facilitará en Castellano.

4.4 *Entradas del programa*

A continuación, se hace una breve explicación de los detalles más importantes de la estructura de los metamodelos de entrada. Solo se describirá aquello que será utilizado en la implementación.

4.4.1 UML

Hemos visto antes en la sección 2.1.1 los componentes de UML de los diagramas de secuencia, pero ahora en la Figura 4.3 podemos observar cómo se estructuran esos datos para poder diseñar una implementación para la transformación.

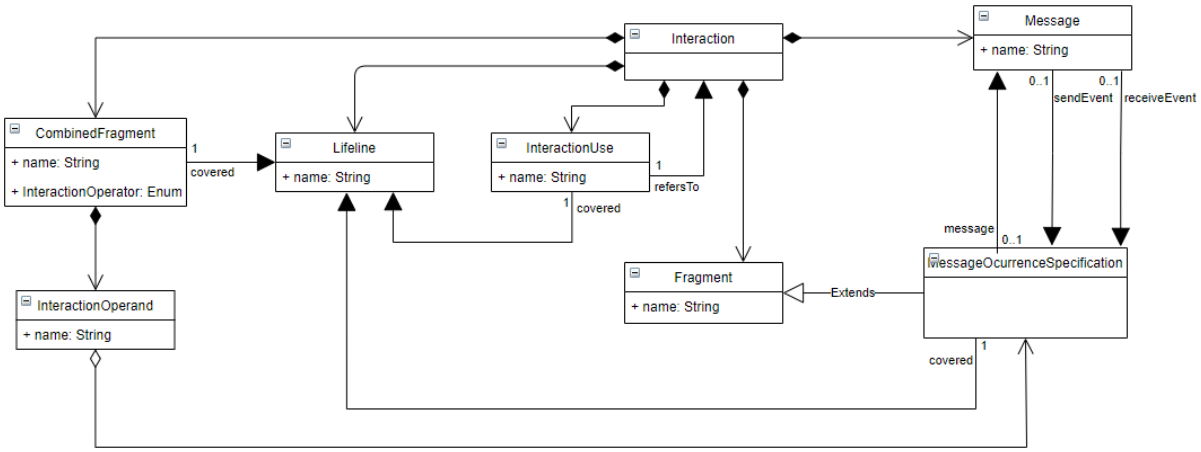


Figura 4.3: Diagrama de clases básico de los diagramas de secuencia de UML en EMF. Realizado a partir de un extracto de [2].

Empezamos con el elemento primitivo del diagrama de secuencia que es el *Interaction*, este elemento representa a cada diagrama de secuencia particular. Por lo tanto, podríamos tener más de un diagrama dentro de un mismo archivo, esto podría deberse a un diagrama referenciado desde otro, y cada uno tendría su propio *Interaction*. Hay ciertas cosas que se pueden entender mejor con el diagrama de clases que con los propios componentes del diagrama de secuencia como vimos al principio.

Primero, observamos la existencia de un elemento que no conocíamos que es *MessageOcurrenceSpecification*, este elemento es el que une cada mensaje con su emisor y receptor, este actúa de evento tanto de envío como de recepción en cada caso, esto permite acceder desde un mensaje a los dos actores que están involucrados en su intercambio. Segundo, podemos ver que los componentes *MessageOcurrenceSpecification*, *InteractionUse* y *CombinedFragment* contienen una referencia al *Lifeline* o los *Lifelines* a los que pertenecen con un campo *covered*, importante saberlo para poder realizar la transformación con coherencia. Tercero, observamos la existencia de un objeto llamado *InteractionOperand* el cual se contiene dentro de *CombinedFragment* y dependiendo del tipo de *CombinedFragment* que estemos hablando el numero variará, teniendo cada uno de ellos una funcionalidad en particular. También es destacable que no se representa el *ExecutionSpecification* el cual era mostrado al principio como elemento importante en el diagrama de secuencia, pero en lo que nos concierne para la transformación del modelo no tiene ningún papel importante. Por último, es reseñable destacar el *InteractionUse*, el cual fundamentalmente tiene un nombre y una referencia a una *Interaction* y una a un *Lifeline*. Esta es la razón que se contempla para que haya más de un *Interaction* dentro de un *Model*.

4.4.2 MARTE

Como se ha comentado anteriormente, MARTE es una extensión de UML que permite añadir información que UML no contempla, como datos de análisis de rendimiento. La notación de MARTE engloba un gran abanico de información, por tanto, en este proyecto, se van a usar los elementos que se presentan en la siguiente Tabla 4.1.

Estereotipo	Etiqueta	Elemento del diagrama	Explicación
GaWorkload-Event	pattern	Lifeline	Población cerrada que ejecuta el escenario
GaStep	hostDemmand	MessageOcurrenceSpecification	Demanda de CPU en el *host* que ejecuta una acción tras el envío o recepción de un mensaje
	rep	Interaction Operand	Numero de repeticiones de un l*oop*.
	prob	Interaction Operand	Probabilidad de ejecución de un *InteractionOperand* de tipo *alt*.

Tabla 4.1: Elementos del perfil MARTE que se van a usar para el análisis de rendimiento. Extraído y traducido de [16].

4.5 *Salida del programa*

La salida de este programa ha sido definida como PNPRO ya que es un formato que permite representar las redes de Petri gráficamente, a continuación, se detalla el formato de esta salida.

4.5.1 PNPRO

PNPRO es un formato para redes de Petri basado en XML, el cual contempla su representación gráfica como también contempla todos los tipos de redes de Petri. En este caso particular usaremos redes de Petri estocásticas, por lo tanto, se detallan en el diagrama siguiente los componentes de este formato para las redes de Petri estocásticas.

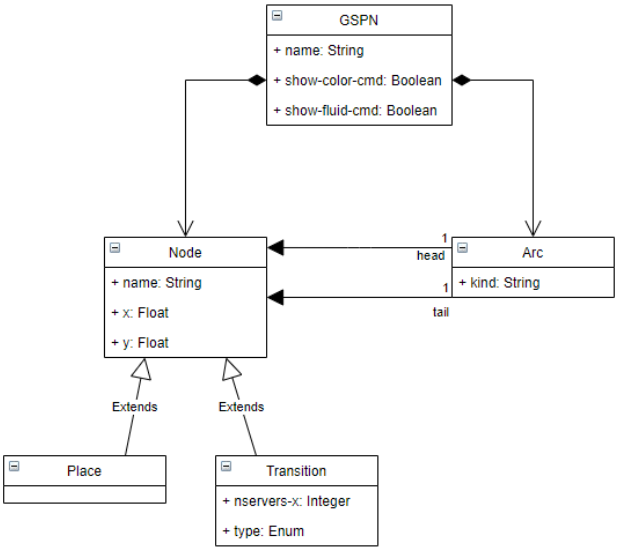


Figura 4.4: Diagrama de clases de los elementos importantes del formato PNPRO. Realizado con información extraída de [http://www.di.unito.it/~amparore/mc4cslta/editor.html].

La estructura de datos tiene a *GSPN* como elemento principal, este contiene elementos que se dividen entre nodos y arcos. Los nodos pueden ser lugares o transiciones, teniendo en su interior sus posiciones con coordenadas en x e y directamente. Es un formato más sencillo que el de PNML.

Se ha elegido este formato en particular porque es el que utiliza GreatSPN, por tanto, al hacer uso de dicha herramienta era necesario el uso de este formato.

5 Diseño

En este apartado se expone la información referente a la etapa de diseño en un desarrollo software. Primero se explican las herramientas que se utilizan para el desarrollo, tanto dentro de Eclipse como la herramienta externa. Por último, se muestra el diseño de las transformaciones que se han desarrollado en esta etapa.

5.1 *Herramientas a utilizar*

Las herramientas para este proyecto vienen definidas en los requisitos, por tanto, no ha habido ninguna comparación entre diferentes herramientas encontradas, a continuación, se hace una breve explicación de cada herramienta o lenguaje a utilizar.

5.1.1 Basadas en Eclipse

En este apartado se hace una breve descripción de las herramientas que están disponibles para Eclipse que van a ser utilizadas y la utilidad para este proyecto en particular.

5.1.1.1 Papyrus

Eclipse Papyrus es una herramienta de código abierto para la ingeniería basada en modelos. Esta herramienta tiene soporte para diseño gráfico para UML2 definido por la especificación de OMG. Esta herramienta es la que se utilizara para crear, mostrar y editar los diagramas de secuencia que servirán de entrada al *plugin* a desarrollar. Esta herramienta hace mucho más fácil y cómodo la creación de diagramas con carácter funcional, por otro lado, nos permite introducir la funcionalidad extra del perfil MARTE a UML, por tanto, se ha visto necesario utilizar una herramienta de estas características para facilitar la entrada del programa dentro de unos estándares muy utilizados y de calidad [18].

5.1.1.2 ATL

ATL (ATL Transformation Language) es un lenguaje y una herramienta de transformación de modelos. Es un lenguaje M2M (Model to Model) el cual trata de un número de reglas que sirven para definir como navegar y enlazar los elementos de un modelo para la creación e inicialización de elementos de otro modelo [19].

Se ha elegido usar este lenguaje por su idoneidad, facilidad de uso y buen rendimiento. Este lenguaje se usará para hacer la transformación conceptual de un diagrama de secuencia a una red de Petri en PNML (Petri Net Markup Language), pero esta red de Petri no será el resultado final, pero permitirá poder ser manejado por EMF, por tanto, será necesaria una transformación más desde ese formato PNML al final que se ha descrito en los requisitos funcionales. Para esta transformación final se usará Acceleo, del cual se habla a continuación.

5.1.1.3 Acceleo

Acceleo es un lenguaje y una herramienta M2T (Model to Text) que permite producir cualquier tipo de código desde cualquier tipo de dato disponible en formato EMF [20]. Esta tecnología por tanto es la que necesitamos para pasar de una red de Petri en formato PNML a una red de Petri en otro formato (PNPRO en este caso), ya que el cambio es solo de formato ya que los cambios entre los modelos son superficiales.

5.1.1.4 EMF

EMF se utilizará para crear los modelos de entrada y para el manejo de estos junto con el modelo intermedio y sus metamodelos dentro de Eclipse.

5.1.2 Herramientas externas

Se ha usado una herramienta externa a Eclipse para este proyecto y esa herramienta es GreatSPN, la cual se comenta brevemente su función a continuación.

5.1.2.1 GreatSPN

GreatSPN es una herramienta para el modelado, validación y análisis de rendimiento de GSPN o redes de Petri estocásticas generalizadas y sus variantes con color. Esta herramienta nos permite tanto modelar, como visualizar, como también analizar y simular las redes de Petri [17]. Esta herramienta se utiliza en el proyecto como referencia para visualizar las redes de Petri generadas y hacer el análisis sobre estas.

5.2 *Transformaciones Diagrama de Secuencia con MARTE a redes de Petri*

Antes de desarrollar el programa se ha hecho un análisis acerca del funcionamiento de los diagramas de secuencia y de las redes Petri y se ha formalizado la transformación de cada elemento de los diagramas de secuencia en su correspondiente en redes de Petri.

5.2.1 Elementos de diagrama de secuencia

Para cada requisito funcional de transformación se ha hecho una estructura genérica que seguirá el programa para transformar esos elementos. En algunos casos se muestran los ejemplos de dos y tres *Lifeline* con el objetivo de ver que sirven para cualquier número de *Lifeline* involucrados. Se han planteado las transformaciones obviando los *CombinedFragment* o *Reference*, por tanto, el punto de partida de las funcionalidades de los *CombinedFragment* o *Reference* se basan en el esquema de envío de mensaje básico y después se le hacen modificaciones para cumplir con cada funcionalidad, por ello las transformaciones de esos elementos se muestran en dos pasos.

- Envío y recepción de mensajes. Se han hecho dos ejemplos, uno de envío y recepción básico y otro que incluye una respuesta. (RF1.1.1)

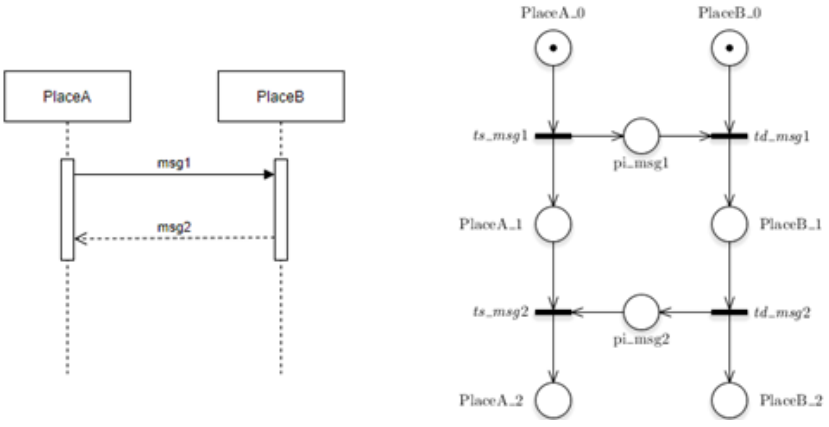


Figura 5.1: Transformación de envío y sincronización de diagrama de secuencia a red de Petri. Fuente propia.

Como podemos observar en el ejemplo de la Figura 5.1, se muestra un ejemplo de envío de un mensaje con respuesta y su transformación a red de Petri. Como vemos, cada *Lifeline* tiene sus propios *Places* y cuando un mensaje se envía se hace un envío con un toquen que se sincroniza con el de la llegada y un token que queda a la espera de más funcionalidad del *Lifeline* de envío, cuando la respuesta llega se vuelve a sincronizar. Cada recurso tiene su propio token que es el que representa ese recurso, por tanto, cuando hay un envío de mensaje, se genera un token que simula el mensaje y otro token se queda a la espera en el *Lifeline* de envío, por otro lado, en el *Lifeline* de recepción, se fusionan en token de mensaje y el del *Lifeline* de recepción, por tanto, se representa un mensaje cuando se envía, pero el resto del tiempo solo existen los tokens de los recursos.

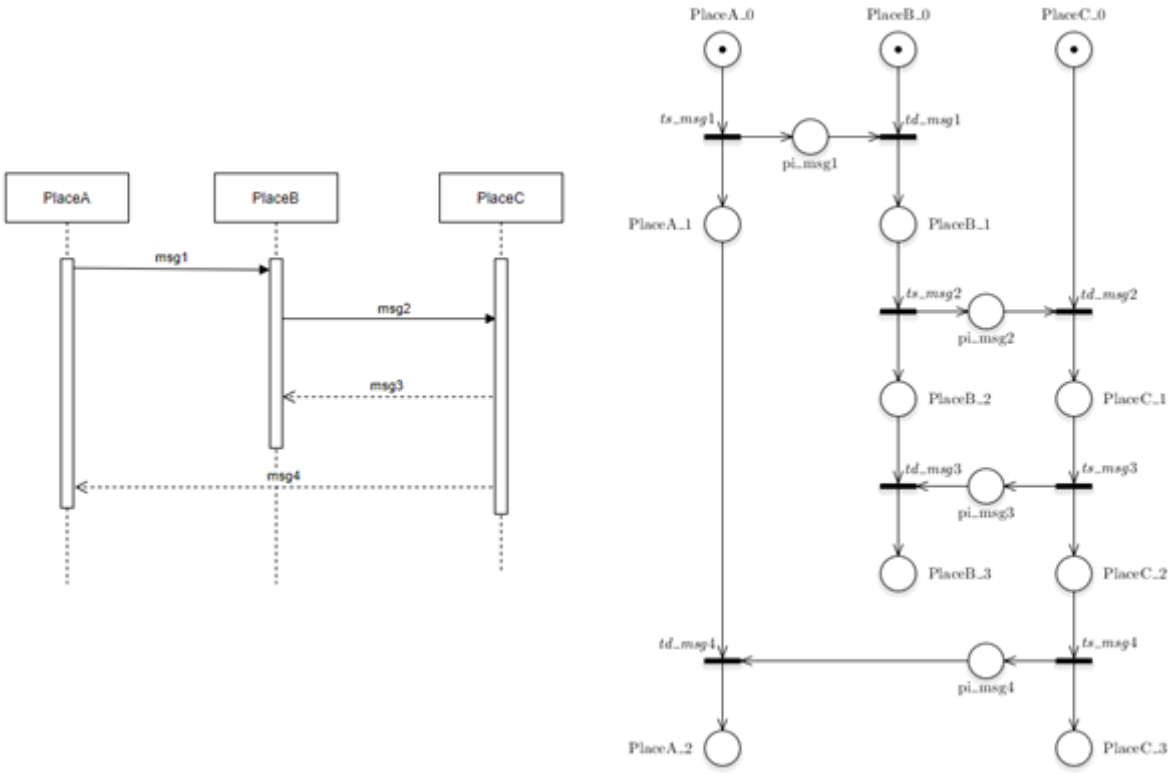


Figura 5.2: Transformación de intercambio de mensajes de diagrama de secuencia con 3 *Lifeline* a red de Petri. Fuente propia.

En la Figura 5.2, podemos observar un ejemplo más complejo, con tres *Lifeline*. Con esto vemos que la estructura es la misma.

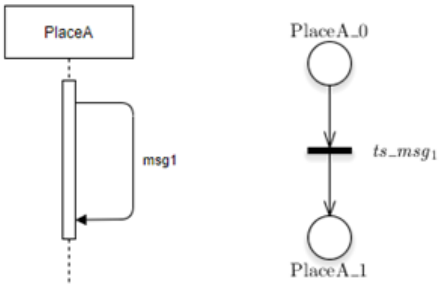


Figura 5.3: Transformación de envío y recepción de mensaje por un mismo *Lifeline*. Fuente propia.

Para el caso particular de envío de mensaje de un *Lifeline* a sí mismo, la estructura de envío de mensaje es más sencilla, reduciéndose a una transición a un nuevo estado perteneciente al mismo *Lifeline*, como observamos en la Figura 5.3.

- Funcionalidad de referencia a otro diagrama de secuencia. (RF1.1.2)

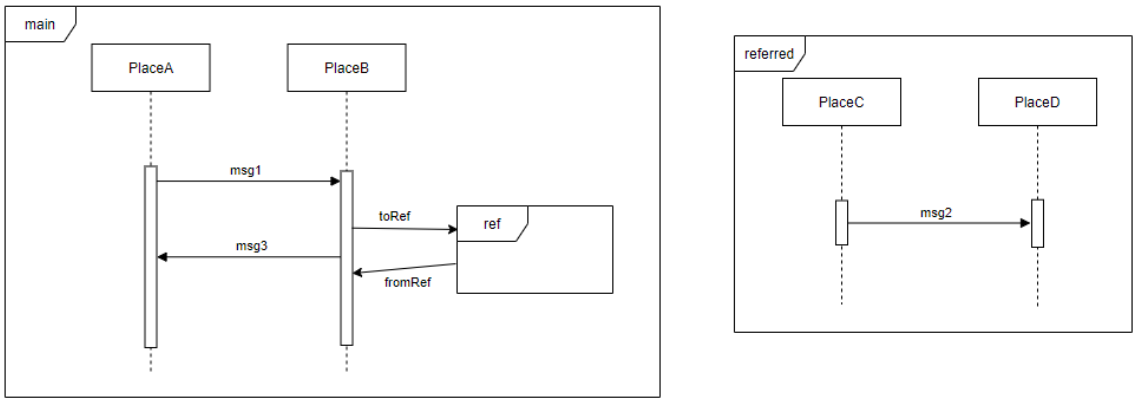


Figura 5.4: Diagrama de secuencia con *InteractionUse* de *Reference*. Fuente propia.

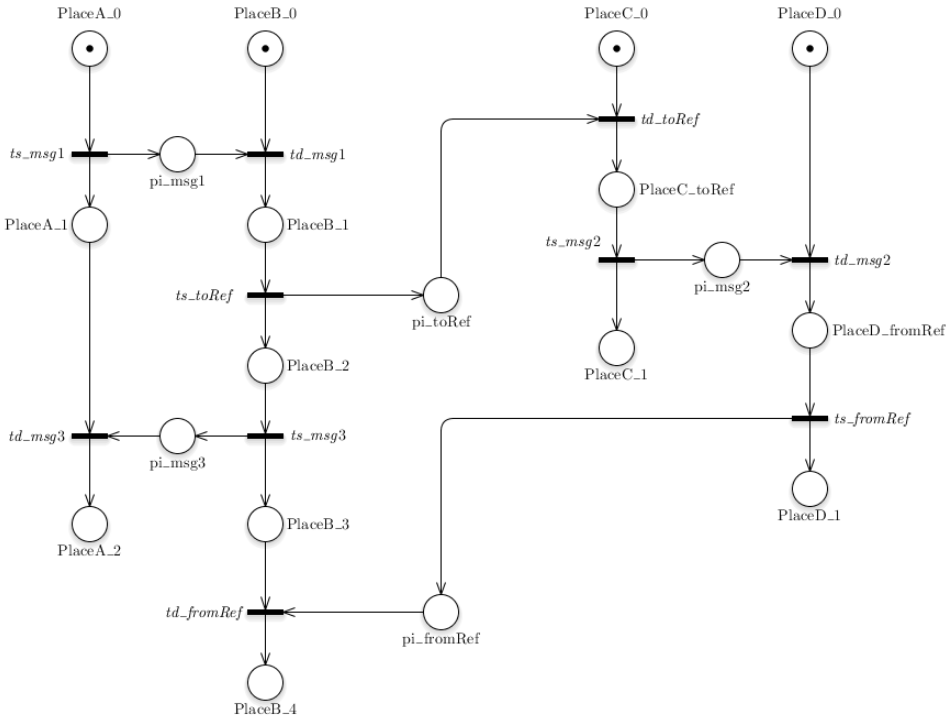


Figura 5.5: Transformación del diagrama de Figura 5.4. Fuente propia.

Como vemos, el *Reference* se referencia por medio de mensajes. Por tanto, la estructura es igual que la de un envío de mensaje con la diferencia de que el mensaje que recibe el *Reference* lo recibe al principio del *Lifeline* que hace la primera interacción interna en el diagrama referenciado. El mensaje de vuelta lo envía el *Lifeline* que hace la última interacción después de la misma.

- Funcionalidad de ejecución de instrucciones en paralelo. (RF1.1.3)

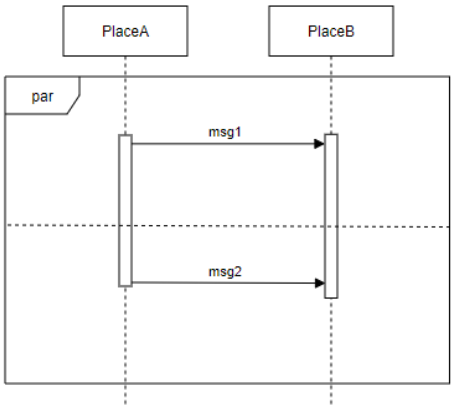


Figura 5.6: Diagrama de secuencia con *CombinedFragment* de tipo *Parallel*. Fuente propia.

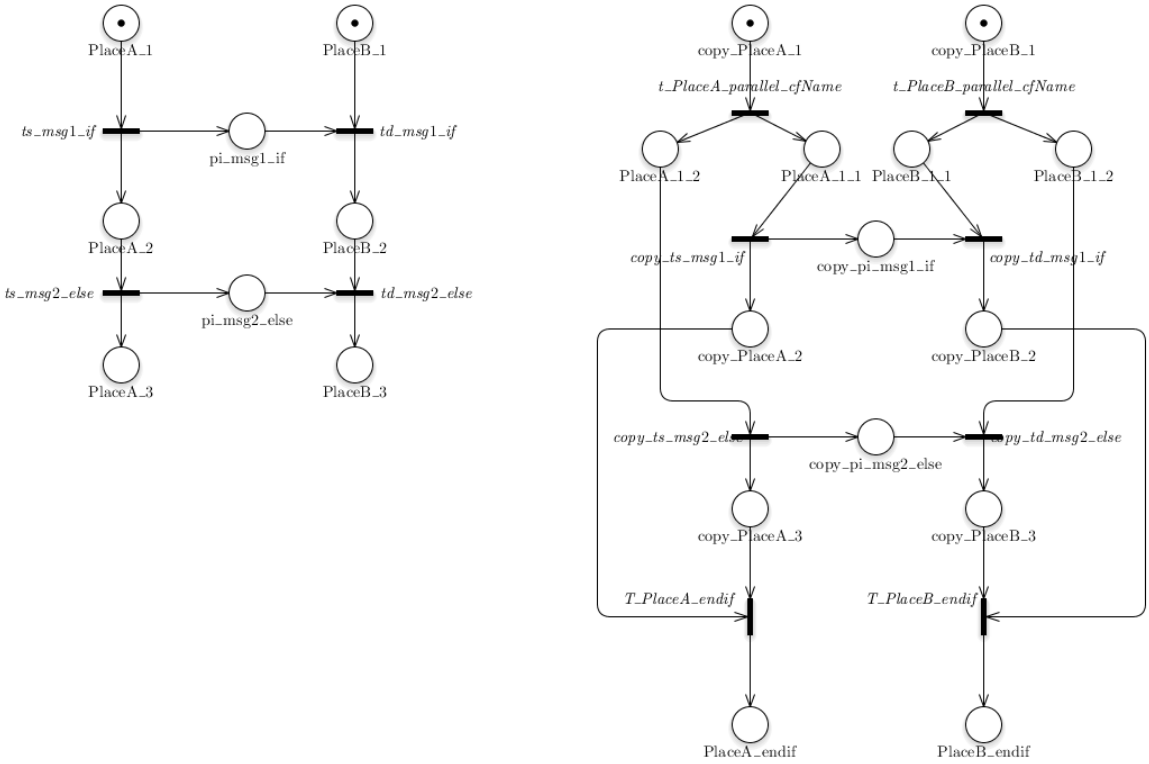


Figura 5.7: Transformación en dos pasos de diagrama de secuencia de Figura 5.6 a red de Petri. Fuente propia.

Para lograr una ejecución en paralelo, hay que multiplicar las líneas de ejecución por el número de zonas en paralelo que haya, o lo que es lo mismo, hay que generar ese número de tokens, con el objetivo de que se puedan ejecutar todas las líneas de ejecución. Para lograr esto, utilizamos la característica que vimos en la Figura 2.4, en esta vemos que para un token de entrada a una transición hay tantos tokens de salida como arcos de salida tenga esa transición. Por tanto, en el inicio del *CombinedFragment*, se generan una transición de inicio común y un *Place* de inicio para cada zona paralela y se conectan, al final de la zona paralela se hace lo contrario, como se explica en la Figura 5.7.

- Funcionalidad de ejecución de instrucciones en bucle, sabiendo que el número de repeticiones del bucle es el indicado por la notación de MARTE que se explicará en el siguiente apartado, para este ejemplo suponemos 5 repeticiones. (RF1.1.4)

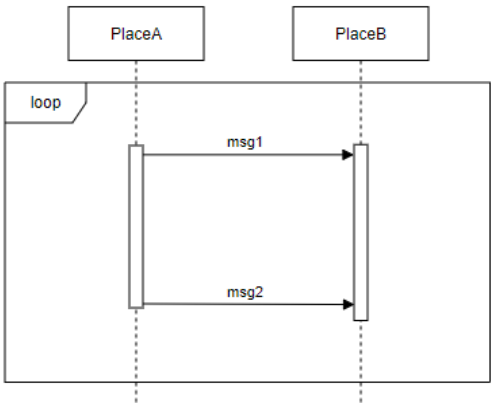


Figura 5.8: Diagrama de secuencia con *CombinedFragment* de tipo l*oop*. Fuente propia.

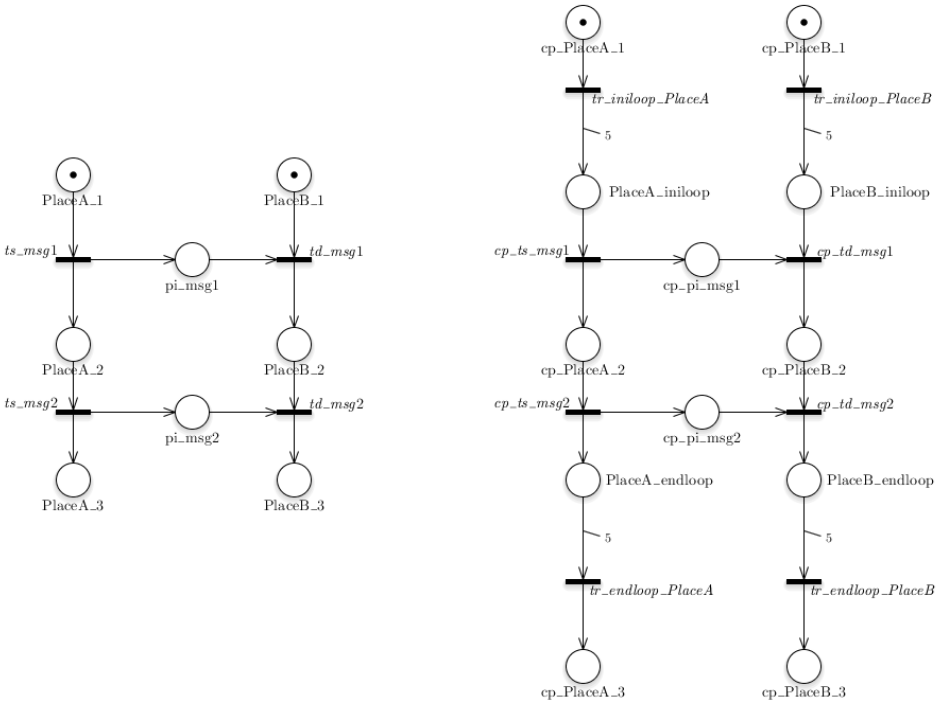


Figura 5.9: Transformación en dos pasos de diagrama de secuencia en Figura 5.8 a red de Petri. Fuente propia.

La generación de bucles es a través de la multiplicidad de los arcos de entrada y salida. Esta característica de los arcos es simular que hay un numero de arcos definido por esa multiplicidad con el mismo origen y destino. Con esto logramos que generar el número de tokens que queramos, o lo que es equivalente, que una línea de ejecución se realice tantas veces como queramos, que es exactamente lo que es un bucle. Por tanto, al inicio del bucle se crea un arco con origen en una transición con multiplicidad igual a la del número de iteraciones del bucle, lo que nos genera ese mismo número en tokens en el *Place* de inicio del bucle. A la salida del bucle se genera un arco con la misma multiplicidad que al inicio, pero con origen en un *Place*, esto genera una reducción a un token único para seguir la ejecución y sincroniza los tokens a la salida del bucle.

Funcionalidad de *if-then-else*, en las que hay varias líneas de ejecución y solo se ejecuta una de ellas en cada iteración, sabiendo que la probabilidad de cada rama es la indicada por la notación de MARTE, el ejemplo es sin tomar una probabilidad definida para cada rama, por tanto, todas las ramas tienen la misma probabilidad. (RF1.1.5)

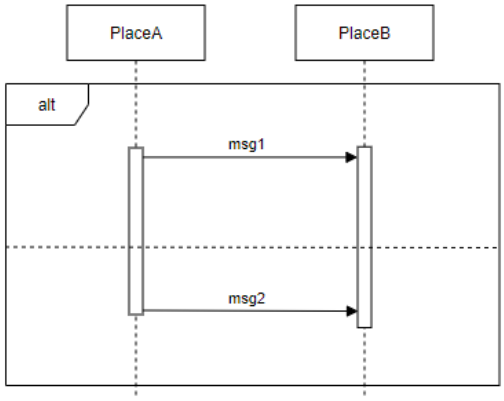


Figura 5.10: Diagrama de secuencia con *CombinedFragment* de tipo *Alternative*. Fuente propia.

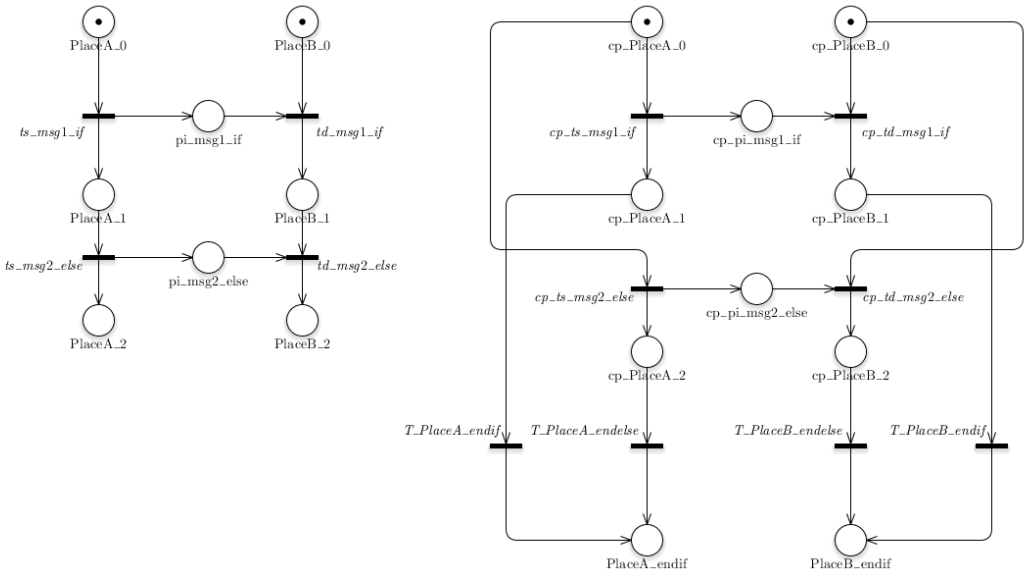


Figura 5.11: Transformación en dos pasos de diagrama de secuencia en Figura 5.10 a red de Petri. Fuente propia.

Al contrario que en *Parallel*, lo que se pretende lograr en esta estructura, es que al inicio del *CombinedFragment* se tenga que elegir una línea de ejecución. Por tanto, en vez de sacar varios arcos de una transición, se sacan los arcos desde el *Place* de entrada, por tanto, en cada ejecución, se ejecutará una de las líneas. Al final, cada línea de ejecución tiene su propia transición para finalizar los arcos en un *Place* común para todas las opciones del *CombinedFragment*.

Funcionalidad de anidación. Esta función no se ha formalizado ya que no se veía necesario, en caso de anidación, el primer nodo del fragmento anidado iría conectado al último nodo del fragmento inmediatamente interior y el ultimo nodo del fragmento anidado se conectaría al primero nodo del fragmento siguiente, por ello se ha tratado a cada funcionalidad como atómica, para poder trabajar con ello como una unidad fácilmente trazable y editable. (RF1.1.6)

5.2.2 Elementos de MARTE

La transformación de los elementos de MARTE se integra dentro las transformaciones tanto la de ATL como la de Acceleo. La razón de ello es que como podemos observar en el diagrama de la Figura 6.2, PNML no dispone de la estructura necesaria para almacenar y trabajar con parámetros de las redes de Petri que se requieren para hacer el análisis. Por tanto, en una primera transformación en ATL, los parámetros obtenidos de MARTE se almacenarán como *ToolInfo* en su objeto correspondiente para posteriormente en Acceleo añadirse al diagrama de GreatSPN en sus campos correspondientes. En esta fase de diseño se tendrán en cuenta las transformaciones desde MARTE a PNPRO sin importar lo intermedio. Siendo el intermedio una tarea superflua que se resuelve en la parte de programación. A

continuación, se listan los elementos de MARTE que vimos en la sección de conceptos previos de MARTE y se explican sus transformaciones.

- <<GaWorkloadEvent>>{pattern}: este elemento perteneciente a un *Lifeline*, quiere decir la población que ejecutarán el sistema, o lo que es equivalente en la red de Petri, indicará el número de tokens iniciales que contendrá el primer *Place* del *Lifeline* en cuestión.

- <<GaStep>>{hostDemand}: este elemento se ha decidido que se coloque en los eventos de recepción y envío de mensajes ya que es una forma intuitiva para el usuario y fácil de programar que permite dar sentido a que el trabajo que realiza un host en particular, lo realiza cuando le llega un mensaje o después de enviar un mensaje. Este elemento se traduce en las redes de Petri como una transición de tipo *exp* (exponencial) con un *rate* definido por el valor de este elemento de MARTE.

- <<GaStep>>{rep}: esta transformación, es la más intuitiva sabiendo el cómo se genera transforma un bucle en redes de Petri. Como vemos en la Figura 5.9, el bucle se representa añadiendo un arco de inicio y otro de final con una multiplicidad definida por el número de repeticiones del bucle, por tanto, se obtiene este número de repeticiones de este campo de MARTE y se le añade al primer y último arco del bucle en un campo llamada *multiplicity*.

- <<GaStep>>{prob}: esta es una probabilidad estimada de las veces que se va a ejecutar en el tiempo una opción en un *CombinedFragment* de tipo *alt*. Por tanto, cada *InteractionOperand* que tenga una probabilidad definida se obtendrá de este campo. La transformación se realiza añadiéndole un campo llamado *weight*, con el valor de este elemento de MARTE, a la primera transición que forma parte de dicho *InteractionOperand*. En la estructura de la Figura 5.11 las transiciones de cada *InteractionOperand* que designarían su probabilidad serían *cp_ts_msg1_if* y *cp_ts_msg2_else*, para el primer y segundo *InteractionOperand* respectivamente.

5.3 ***Diagramas***

En este apartado se muestran los diagramas realizados en el diseño de este proyecto, los dos diagramas considerados fundamentales en la etapa de diseño son, el diagrama de clases y el diagrama de secuencia. El primero porque nos permite visualizar los componentes de los que se compondrá nuestro programa y la estructura de estos. Por otro lado, el diagrama de secuencia nos muestra la secuencialidad de un caso de uso concreto permitiéndonos ver como interactúan y en qué orden los componentes del programa, en este caso es más provechoso todavía ya que por las características de este programa solo dispone de un caso de uso, por lo tanto, con un único diagrama de secuencia se puede representar la funcionalidad integra del programa a desarrollar.

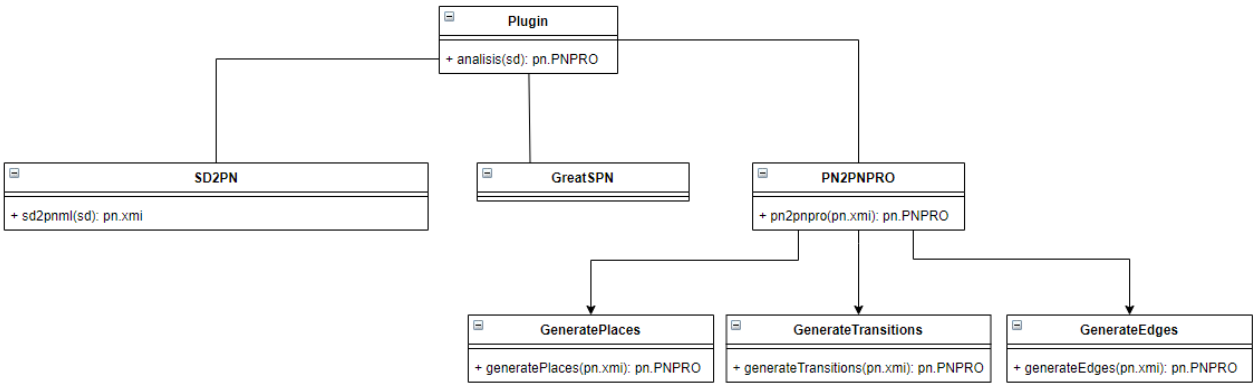


Figura 5.12: Diagrama de clases del *Plugin* a desarrollar. Fuente propia.

Como vemos, el *Plugin* principalmente está compuesto de dos módulos internos y uno externo. El módulo externo con el que trabaja el *plugin* es GreatSPN que es el que nos permitirá representar y analizar las redes de Petri una vez creadas. Por otro lado, los dos módulos internos son los encargados

de cada uno de los dos pasos de la transformación del diagrama de secuencia a redes de Petri, siendo SD2PN el que realiza la primera transformación y PN2PNPRO el que realiza la segunda. Las funciones enlistadas en SD2PN son las reglas de las que dispone para transformar cada uno de los elementos de los diagramas de secuencia que constan en los requisitos, estas reglas serán explicadas en profundidad en el apartado de desarrollo. El módulo de PN2PNPRO como vemos le heredan tres módulos más, cada uno se ocupa de la generación de uno de los elementos principales de las redes de Petri (Lugares, Transiciones y Arcos).

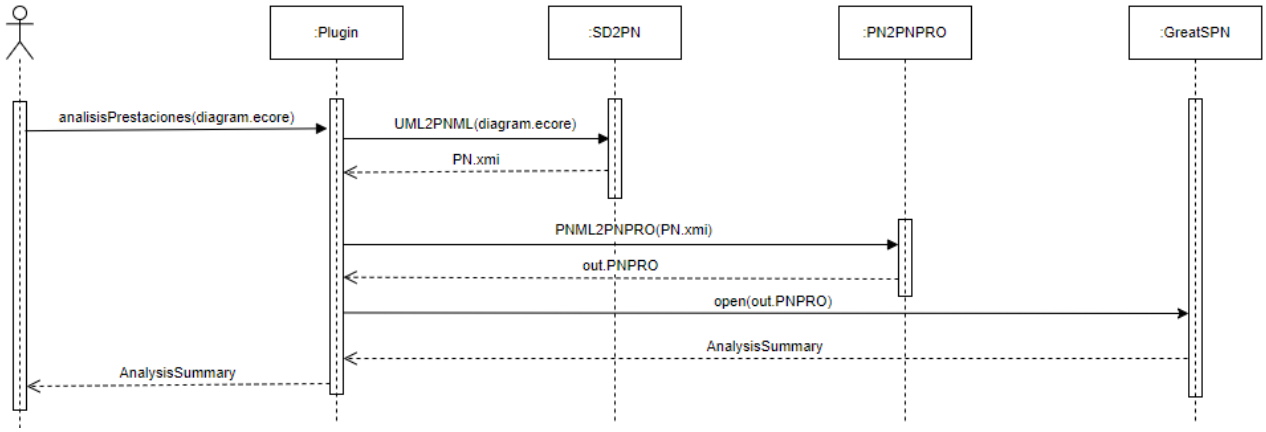


Figura 5.13: Diagrama de secuencia del *Plugin* a desarrollar. Fuente propia.

Como podemos observar en el diagrama de secuencia, el usuario interactúa con el *plugin*, el cual sirve de intermediario entre el resto de los módulos. Como vemos en el diagrama, el orden de las funciones es primero desde el *diagrama.ecore* a *PN.xmi*, después de *PN.xmi* a *out.PNPRO*. Por último, se envía a GreatSPN para la representación y posterior análisis, y para finalizar, el *plugin* nos devuelve un mensaje con un resumen del análisis realizado a parte de todos los archivos generados en el análisis en la carpeta de destino elegida.

6 Desarrollo

En este apartado se expone la fase de desarrollo en un proceso software, esto incluye toda la información referente a como se ha implementado el software y toda su funcionalidad. Primero se comenta la arquitectura de ficheros que tiene el proyecto. A continuación, se comenta por separado los aspectos fundamentales de la implementación de la transformación de ATL, y, por último, la de Acceleo.

6.1 *Arquitectura de ficheros*

La arquitectura de ficheros que se ha creado del *plugin* se divide por funcionalidades. A continuación, vemos los módulos de los que se compone.

- **umlAnalisis*Plugin***: éste es el módulo principal y donde se integra la funcionalidad del *plugin*, desde la ampliación de la interfaz de Eclipse para la integración en su barra de herramientas hasta las llamadas a todos los demás módulos y la representación de la salida.
- **uml2pnml**: este módulo es el que contiene la funcionalidad de la primera transformación en ATL, así como las clases java necesarias para poder ser accedida desde el módulo principal. Esas clases java han sido creadas por la herramienta de *ATL plugin* para crear *plugin* de ATL para permitir ejecutar la transformación de forma programática.
- **pnml2pnpro**: este es el módulo que contiene la funcionalidad de la segunda transformación en Acceleo y sus clases java para la comunicación con el módulo principal.
- **pnml**: este módulo contiene el metamodelo de PNML necesario para las dos transformaciones, ya que es el metamodelo del modelo intermedio.
- **MiniGreatSPN**: este es el módulo que contiene la una versión portable de GreatSPN con la que se hace un primer paso del análisis. En este primer paso, se generan unos archivos ".net" y ".def" que son los que usará GreatSPN para hacer el análisis.
- **creditCard**: este último módulo, es un módulo externo y no es necesario, pero ha sido utilizado un módulo de prueba con varios modelos de ejemplo. Este módulo será creado por el usuario y es donde tendrá los diagramas a analizar, así como se supone que almacenará las salidas.

6.2 *ATL*

En este apartado se explica los aspectos más importantes de la programación del módulo de ATL encargado de la primera transformación desde un diagrama de secuencia en EMF a una red de Petri en EMF.

6.2.1 Arquitectura de ficheros

Los elementos de este módulo han sido ordenados por tipo de elemento para una mejor comprensión. En la Figura 6.1 se representa como están ordenados los componentes.

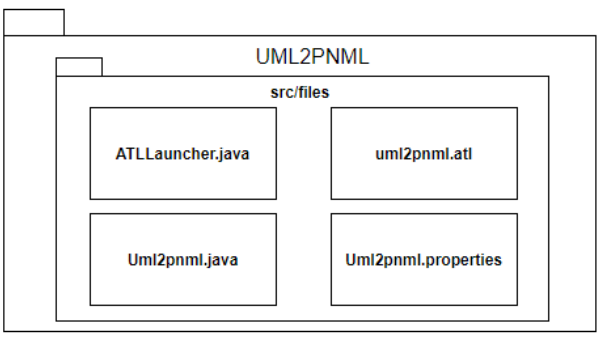


Figura 6.1: Arquitectura de los componentes internos del módulo UML2PNML. Fuente propia.

- **ATLLauncher.java**: este archivo es el que contiene la funcionalidad de lanzamiento de la transformación llamando a *uml2pnml.atl*. Este archivo es autogenerado por la herramienta *ATL plugin*.

- **uml2pnml.atl**: este archivo es el que contiene el código ATL con las reglas de las transformaciones.

- **Uml2pnml.java**: este archivo es el responsable de dar cohesión a este módulo. Por un lado, recoge las propiedades del *Uml2pnml.properties*, luego llama al *ATLLauncher.java* para lanzar la transformación, por último, provee de unas funciones que son las que se usarán para realizar la transformación desde el módulo principal del *plugin UMLAnalisisPlugin*.

- **Uml2pnml**: es un archivo de propiedades que contiene los datos necesarios para la la ejecución de la transformación. Principalmente, aquí se establecen las rutas a los metamodelos tanto de entrada como de salida.

6.2.2 Transformaciones de los elementos

En este apartado se muestran las reglas principales utilizadas en la transformación, los aspectos más importantes y como se ha afrontado cada transformación.

6.2.2.1 UML

- SD2PNML: esta regla tiene un *Interaction* de entrada y es por ello por lo que es la regla principal, esta regla comprueba que no haya una página creada y si no la hay la crea con el nombre de la *Interaction*. Esto se ha hecho así ya que en la red de Petri no vamos a tener redes secundarias con referencias entre ellas como si puede suceder en los diagramas de secuencia, por tanto, todos los *Interaction* serán incluidos en una misma Page de la red de Petri.

- Interaction2Page: esta regla es a la que llama SD2PNML para crear una página en el caso de que aún no haya sido creada, en ese caso se crea no solo la página sino la *PetriNetDoc* y la *PetriNet*.

- Lifeline2Place: esta regla recibe de entrada un *Lifeline* y crea un *Place*. Esta regla por tanto crea los lugares iniciales que servirán de base para el resto. Para los lugares se ha elegido usar unos nombres con una notación de la forma <*NombreLifeline*>_<*Número*>. *Número* sirve para indicar el orden de los lugares que pertenecen a un mismo *Lifeline*. Por tanto, los lugares creados en esta regla son de la forma <*NombreLifeline*>_0.

- MessageEventRule: esta regla recibe un *MessageOccurenceSpecification* como entrada y su funcionalidad es simplemente comprobar, por un lado, si es el evento de envío y no de recepción. Esto se hace así porque EMF ordena los mensajes por los eventos, por tanto, si queremos tener los envíos ordenados correctamente debemos recorrer sus eventos, esto ocasiona que recorramos dos eventos por mensaje, teniendo en cuenta que haya envío y recepción, por tanto, solo recorremos los de envío con el objetivo de solo crear el envío de mensaje una vez en la red de Petri. Por otro lado, si el mensaje es mandado de un *Lifeline* a otro o si es un mensaje a sí mismo, si es enviado a otro *Lifeline* se llama a la regla *MessageRule* y si es a sí mismo se llama a *Message2MyselfRule*.

- MessageRule: esta regla recibe de entrada un *Message* y crea la estructura de envío de mensaje que vimos en la sección de diseño. Tanto el lugar de partida como el de llegada se obtiene de una estructura de datos que se ha llamado *allPlaces*, que contiene los últimos lugares creados referentes a cada *Lifeline*. Por ello, se obtiene de los dos eventos correspondientes de envío y recepción los *Lifeline* de origen y destino respectivamente y se utilizan sus nombres para buscar en la estructura de *allPlaces* los lugares para hacer la estructura.

- Message2MyselfRule: esta regla recibe un *Message* con la peculiaridad de que sus eventos de envío y recepción apuntan los dos al mismo *Lifeline*, por tanto, al ser la estructura a la que hay que transformar diferente a la del envío a otro *Lifeline*, era necesario crear una regla diferente. Esta regla crea la estructura que podemos ver en la Figura 5.3.

- Message2Reference: esta regla recibe mensaje con la particularidad de que su recepción es por un *Gate* contenido dentro de un *InteractionUse*. Esta regla genera la estructura de un envío de mensaje normal pero solo la parte de envío hasta el *Place* intermedio.

- CombinedFragmentRule: esta regla es la regla principal para los *CombinedFrament*, esta regla recorre los eventos de envío de mensajes que están en su interior y obtiene datos necesarios para hacer la segunda parte de la transformación para los *CombinedFragment*. Esos datos son básicamente para saber la situación del *CombinedFragment* dentro del diagrama, por tanto, los datos que se obtienen son los primeros y últimos arcos, lugares y transiciones, necesarios para cada tipo de *CombinedFragment*. Después según el tipo llama a una función u otra. Para cada uno de los primeros elementos que sea necesario.

- AltCreateStructuresRule: esta regla recibe de entrada dos *Place* y un *CombinedFragment*, esta regla lo que hace es recibir el primer y último lugar dentro un *InteractionOperand* y en cada llamada lo que hace es crear la estructura para el tipo *Alternative* que se muestra en la Figura 5.11 para un *Lifeline* en concreto.

- AltConnectStructuresRule: esta regla el arco de inicio, el lugar de fin, un *CombinedFragment*, y un número. Este número lleva la cuenta de los *InteractionOperand*, con el objetivo de no repetir nombre en las estructuras generadas.

- StructuresCreateParallel: esta regla recibe como la anterior dos *Place* que son el primer y último lugar dentro de cada *InteractionOperand* dentro del *CombinedFragment*. Para cada llamada de esta regla crea la estructura necesaria para el tipo *Parallel* que se muestra en la Figura 5.7. Esta regla se llama dentro de un bucle anidado en otro. El primero bucle recorre los *Lifeline* y el segundo recorre los *InteractionOperand* del *CombinedFragment,* pero debido a la cardinalidad de las conexiones del primer lugar con la primera transición es distinta a la que une la primera transición con cada uno de los primeros lugares de cada camino paralelo, hace que sea necesaria una función a parte para conectar el primer lugar con la primera transición. Ocurre exactamente lo mismo con la última transición y el último lugar.

- StrcuturesConnectParallel: esta regla es la que genera las conexiones que faltaban por hacer en la regla *StructuresCreateParallel*. Esta llamada tiene un nivel menos de anidación de bucle, por lo que cumple con la cardinalidad del bucle que recorre los *Lifeline*.

- LoopRule: esta regla recibe como entrada el primer y último lugar del *CombinedFragment*, el primer y último arco del *CombinedFragment* y el número de iteraciones del bucle. Esta función genera en cada llamada la parte de la estructura de la Figura 5.9 de cada *Lifeline*.

- InteractionUseRule: esta regla es la regla principal de los *InteractionUse*, para nuestro caso, solo sirve para el tipo *Reference*, por tanto, esta regla hace una recolección de los datos necesarios para hacer la estructura de *Reference* en la Figura 5.5, para ello, lo importante es hallar los *Place* intermedios a los que conectar el *Interaction* que hace referencia el *InteractionUse*. Después se llama a *ReferenceRule* para generar las estructuras.

- ReferenceRule: esta regla recibe de entrada el lugar intermedio al que se conectará el diagrama referenciado, y un *InteractionUse*. El *InteractionUse* sirve para obtener el nombre de los lugares que fueron creados para poder unir el diagrama referenciado con el principal.

- TheEnd: esta regla llama a *MakeFinal* para conectar los últimos lugares de cada Lifeline con los primeros. Esto sirve para el análisis de rendimiento.

- MakeFinal: esta es una regla que se llama desde TheEnd. Esta regla recibe dos *Places* como argumento, uno de ellos es el primero de un *Lifeline* y el otro es el último del mismo *Lifeline*.

6.2.2.2 MARTE

La integración de MARTE se ha realizado después de desarrollar las transformaciones tanto de ATL como Acceleo del diagrama de UML sin el perfil MARTE. Por tanto, la integración de esta funcionalidad se ha llevado a cabo retocando las funciones que eran necesarias. A continuación, se listan las funciones que han sido retocadas y la funcionalidad extra.

- **NewToolInfo**: es una regla nueva creada que recibe un *PnObject* al que se le añadirá el nuevo *ToolInfo* creado, y dos *String* los cuales almacenan el nombre del parámetro de MARTE y el valor numérico. Estos valores se almacenan por tanto en un *ToolInfo* y se le añade al objeto que sea necesario. Es una función genérica que se usará en el resto de las funciones que se van a comentar.
- **Lifeline2Place**: esta regla de creación de los lugares iniciales es la que se encarga de comprobar si hay un parámetro definido de *GaWorkloadEvent-pattern* y en el caso de que lo hubiese, se almacenaría en el *Place* en cuestión.
- **MessageEventRule**: esta regla es la que se encarga de llamar a las diferentes reglas de envío de mensajes. En cuestión de MARTE se ocupa de añadir a la transición de destino de los mensajes enviados el parámetro de *GaStep-hostDemmand* llamando a la función de *newToolInfo*.
- **MessageRule**, **Message2Myself** y **Message2Reference**: estas reglas son las de los tres tipos diferentes de envío de mensajes que se han definido. Son las reglas que se ocupan de definir la nueva *ToolInfo* en las transiciones de origen del mensaje con la información del parámetro *GaStep-hostDemmand* para los casos en los que hubiese sido definido en el envío y no en la recepción.
- **CombinedFragmentRule**: esta regla que es la responsable de la gestión de los *CombinedFragment*. En esta regla se comprueba si es de tipo *loop* y de serlo así se obtiene el número de repeticiones para enviársela a la regla de *LoopRule*.
- **LoopRule**: esta regla les incorpora a los arcos de entrada y salida una multiplicidad igual a lo obtenido del parámetro *GaStep-rep* que se pasó como argumento desde la *CombinedFragmentRule*.
- **AltConnectStructuresRule**: en esta regla se añade el parámetro de *GaStep-prob* en las primeras transiciones de cada *InteractionOperand* en el caso de haberlo, haciendo uso de la regla *newToolInfo*.

6.3 *Acceleo*

Este apartado se dedica a explicar los aspectos más importantes acerca del módulo de Acceleo, el cual es el encargado de la segunda transformación, desde una red de Petri en EMF a una red de Petri en PNPRO.

6.3.1 Arquitectura de ficheros

En este módulo (PNML2PNPRO) hay dos elementos fundamentales que son *src* y *output*. Es importante saber que este módulo coge su entrada desde la salida del módulo de ATL, por tanto, solo existe en este módulo una carpeta con la salida.

La carpeta output es donde se almacena la salida del módulo tras la transformación a formato PNPRO y por tanto la salida del *plugin*. Por otro lado, en la carpeta *src* tenemos principalmente *pnmlAcceleo.files* y *pnmlAcceleo.main*. En *pnmlAcceleo.main* se encuentra *generate.mtl* que es el código Acceleo principal, en el que se genera la plantilla donde se introducen el resto de los elementos, también se encuentra *Generate.java* que es el código Java encargado de hacer la generación de lo que se haya mandado desde *generate.mtl*. Por último, en *pnmlAcceleo.files* tenemos tres archivos que son *generateEdges.mtl*, *generatePlaces.mtl* y *generateTransitions.mtl*. Cada uno de ellos se ocupa de generar los elementos que indica su nombre. Los detalles de estos se explican en el siguiente apartado.

6.3.2 Transformaciones de los elementos

En este apartado se explica la funcionalidad principal de los diferentes elementos de este módulo, así como las limitaciones que han propiciado el diseño elegido.

La red de Petri que disponemos como entrada es la que se ve en la Figura 6.2, en cambio la salida debe ser como en la Figura 4.4. Como vemos hay diferencias esencialmente basadas en la complejidad de una y otra pero que en concepto son similares.

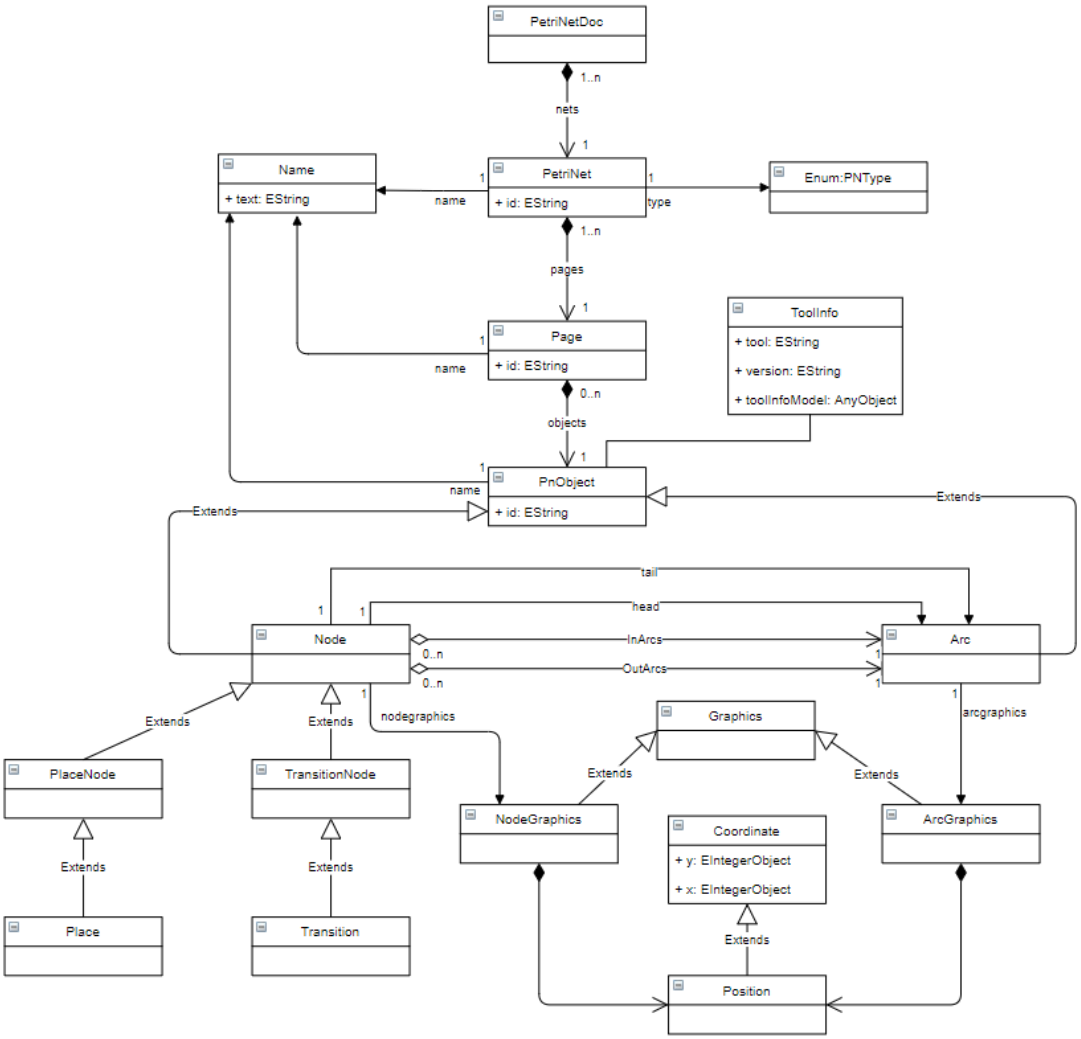


Figura 6.2: Diagrama de clases del metamodelo *pnmlcoredmodel.ecore*. Fuente propia.

Primero, los *Places* son generados desde *generatePlaces.mtl* por una función llamada también *generatePlaces(PetriNet)*. Esta función recorre todos los *Places* del modelo fuente y crea los *Places* del modelo destino por medio del id y por una posición que ha sido fijada la x y la y va aumentando, con esto conseguimos una representación sencilla que permite visualizar todos los *Places*.

Segundo, las *Transitions* son generadas desde *generateTransitions.mtl*, por una función llamada también *generateTransitions(PetriNet)*. Esta función recorre las *Transitions* de una forma similar a la función *generatePlaces* recorre los *Places* y los va representando también en una línea vertical que queda a la derecha de los *Places*. La diferencia aquí es que las *Transition* en el metamodelo de entrada son todas fundamentalmente iguales, por tanto, aquí al tener un *type* y un *weight*, podemos darles sentido a los datos recogidos de MARTE y almacenados en *toolspecifics* de forma temporal. Los *weight* y *type* son creados como se comenta en el apartado siguiente.

Tercero, los *Edges* son generados desde *generateEdges.mtl* por una función *generateEdges(PetriNet)*. Esta función recorre los *Arcs* como lo hacen las otras dos funciones con sus elementos y a continuación genera los *Edges* colocando en *head* y *tail*, los *source.id* y target.id en sus

lugares respectivamente. El campo *mult* es el campo que sirve para almacenar la multiplicidad del arco, este sirve para las transformaciones de los bucles como se explica en el apartado 5.2.2.

6.3.3 MARTE

Las funciones de Acceleo son las mismas, para la incorporación de los parámetros de MARTE se han retocado unas cosas que se cuentan a continuación.

- generateEdges: en esta función, el único parámetro posible que hubiese que añadir es la multiplicidad en los casos de *loop*. Por eso, se comprueba que el arco de origen tenga un *ToolInfo* o no, y en caso de tenerlo se le añade un campo extra al arco llamado *mult* con el valor almacenado en el campo *tool* de *ToolInfo*.

- generatePlaces: en esta función, el único parámetro posible es *GaWorkLoadEvent-pattern* que en las redes de Petri se traduce a un número de tokens inicial, por tanto, en el caso de tener un *ToolInfo* en el *Place*, se le añade un campo extra llamado *marking* con el valor almacenado.

- generateTransitions: en esta función habrá que comprobar aparte de si hay algún *ToolInfo*, también habrá que comprobar el tipo de parámetro que almacenan ya que pueden ser de dos tipos. Si es de tipo *rep*, se añade un campo de tipo *weight* con el valor correspondiente. Por otro lado, si es de tipo *hostDemmand*, se añade un campo de tipo *delay* con el valor correspondiente.

7 Integración, pruebas y resultados

Tras terminar la fase de desarrollo, se han diseñado y realizado unas pruebas para comprobar el correcto funcionamiento del software y de la compleción de los requisitos. Por último, se explica la creación del *Plugin* de Eclipse que es la forma en la que se integran estos módulos con el entorno de trabajo Eclipse.

7.1 *Pruebas*

Para probar el correcto funcionamiento del *Plugin* se han desarrollado seis casos de prueba, pero cinco de ellos están en el Anexo C. En este apartado se muestra el Ejemplo5(MARTE). Esta prueba es la más completa ya que cubre gran parte de los requisitos funcionales. Este ejemplo prueba la funcionalidad de envío de mensajes y su sincronización (RF1.1), la funcionalidad de *loop* (RF1.1.4), la funcionalidad de *alt* (RF1.1.5), la anidación de *CombinedFragment* (RF1.1.6) y por último incorpora todos los parámetros de MARTE requeridos (RF1.2).

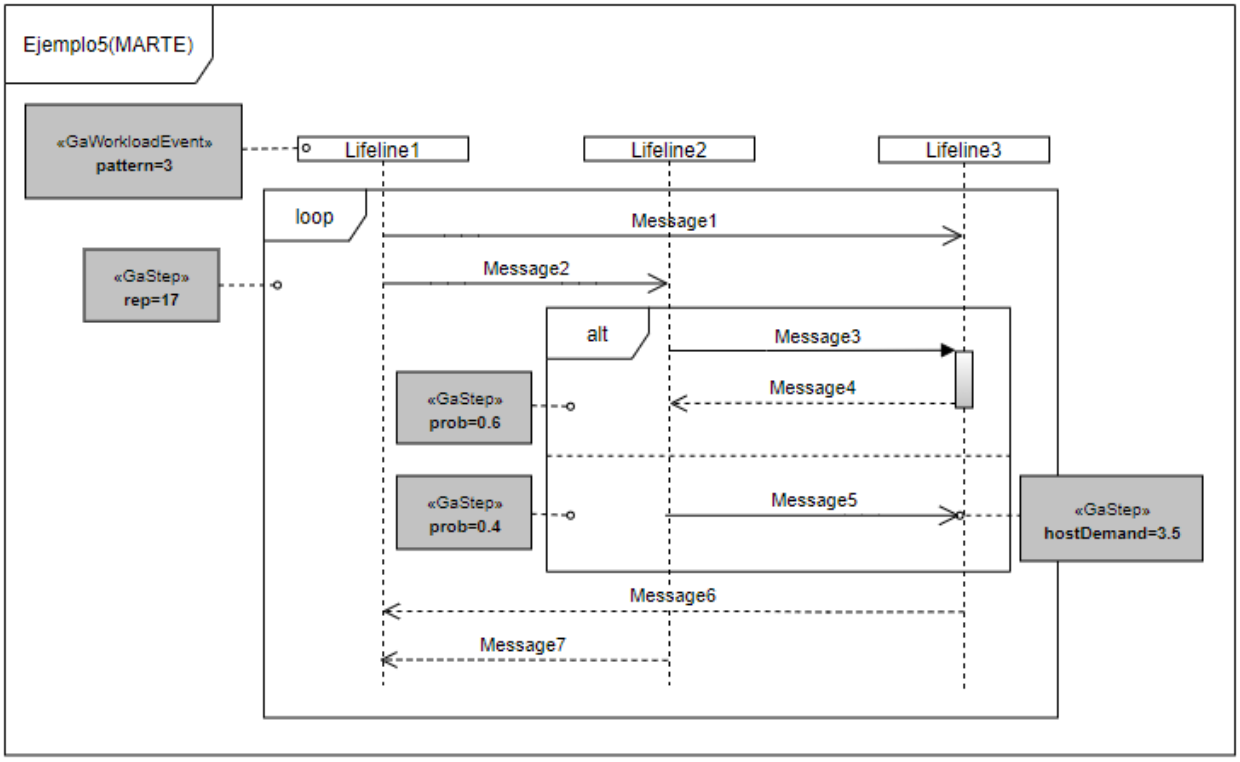


Figura 7.1. Ejemplo5(MARTE). Mismo diagrama que el ejemplo 5 con parámetros de MARTE añadidos. Fuente propia.

7.2 *Resultados*

A continuación, la red de Petri resultante del diagrama del Ejemplo5(MARTE) lo podemos observar en la Figura 7.2.

Primero, el *workloadEvent* se transforma en un número de tokens inicial en el *Place* inicial del *Lifeline* correspondiente. Por otro lado, los *gaStep*, se han transformado en parámetros de transiciones o arcos. Primero, las iteraciones del *loop* del parámetro *rep* de *gaStep*, se transforma en una multiplicidad en los arcos de entrada y en la salida. Segundo, el parámetro *prob* se transforma en un *weight* de las primeras transiciones de cada opción del *Alternative*. Realmente los únicos *weight* necesarios serían en las transiciones del primer mensaje enviado de cada opción, pero se ha realizado en todos los *Lifeline* por sencillez y porque no afecta al funcionamiento. Por último, el *hostDemmand* de *gaStep*, se ha decidido hacer una nueva transición tras la transición de envío o recepción, la cual será de tipo exponencial y con un *rate* de valor igual al valor del parámetro *hostDemmand*. Esta

transición adicional se ha realizado porque en el caso de que sea la primera transición de un *Alternative* con *prob*, ocasionaría un problema ya que una transición no puede tener un *weight* y un *rate* a la vez.

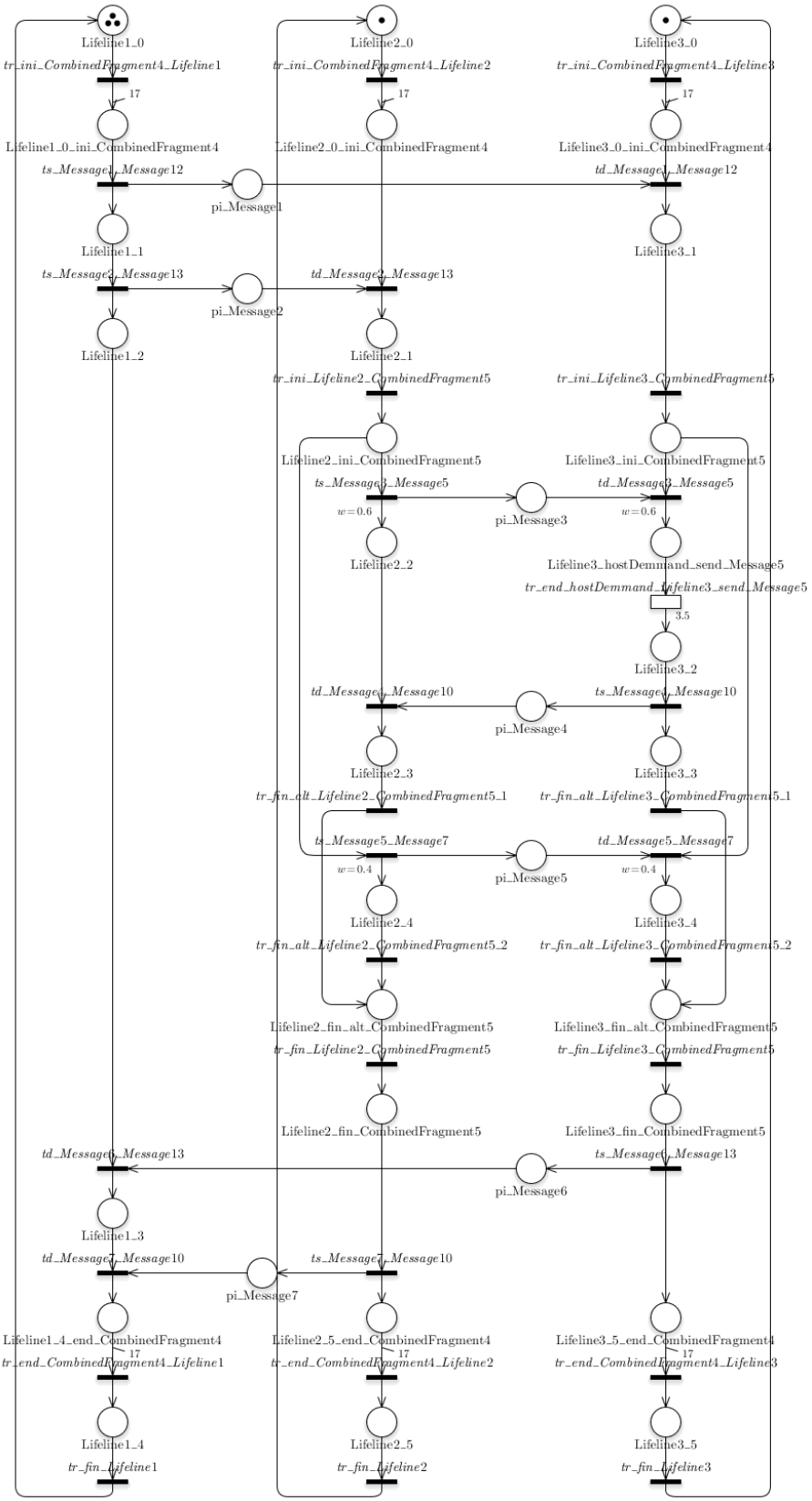


Figura 7.2: Red de Petri del *Ejemplo5*(MARTE). Fuente propia.

Después del análisis de esta red de Petri, se ha realizado un análisis usando el *WNRG solver* de GreatSPN. La resumen del análisis que devuelve GreatSPN y que muestra el *plugin* es el que se muestra en la Figura 7.3.

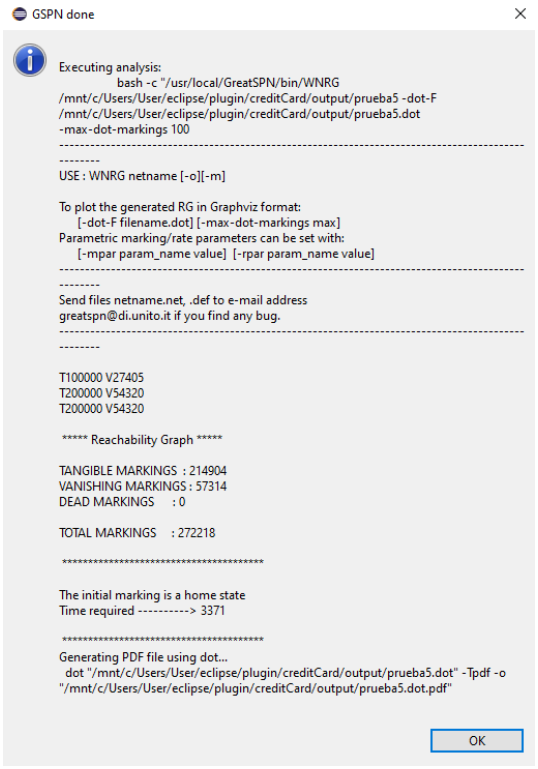


Figura 7.3: Resumen de análisis *Ejemplo5(MARTE)*. Fuente propia.

7.1 ***Plugin Eclipse***

Por último, para integrar todo con Eclipse, se ha desarrollado un *Plugin* que contenga toda la funcionalidad necesaria, la transformación de ATL, la transformación de Acceleo y por último el análisis de la red de Petri resultante por medio de GreatSPN. Este *plugin* consta de un elemento en la barra de herramientas superior de Eclipse, desde donde se podrá realizar la funcionalidad requerida. Como vemos en la Figura 7.4, el elemento está denotado como *UML Analysis*, tras pulsar en él, se desplegará la opción que vemos en la misma figura llamada *Analyze*, también se puede llamar a través del comando *ctrl+6* como sale especificado.

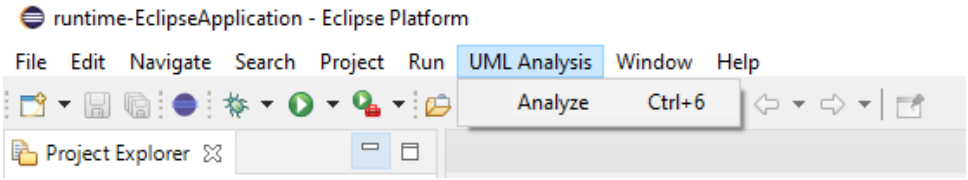


Figura 7.4: Barra de herramientas de Eclipse con la pestaña de Análisis del *plugin* desarrollado. Fuente propia

Tras llamar a *Analyze*, se ha creado una ventana de interacción en la que salen tres campos a rellenar como se ve en la Figura 7.5.

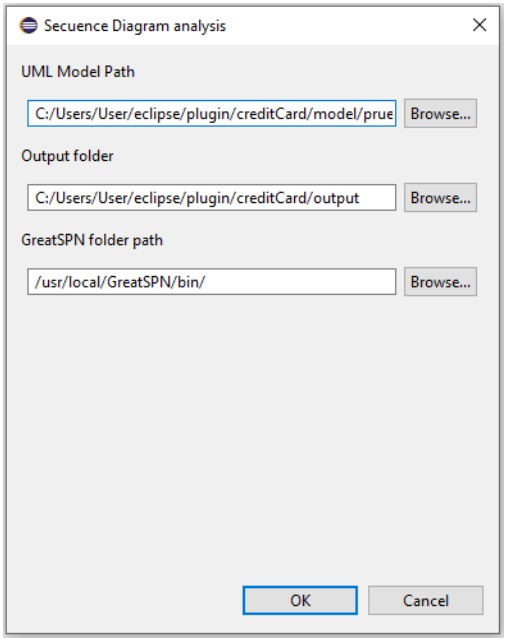


Figura 7.5: Ventana de interacción del *plugin* para la introducción de datos para el análisis de un diagrama UML.

Primero, se nos presenta una casilla en la que introduciremos la ruta absoluta del fichero de entrada *.uml*, que habrá sido desarrollado con Papyrus/MARTE como está previsto. El segundo campo es la ruta absoluta de la carpeta de salida tanto para los archivos intermedios como para los resultados finales. El último campo es la ruta absoluta en el WSL para la carpeta del analizador WNRG (por defecto */usr/local/GreatSPN/bin/*) que es el que usa este *plugin*. En este último campo, de no querer hacer el análisis, de solo querer la transformación a red de Petri, hay que poner en este campo *No* o dejar el campo en blanco. De esta forma en la carpeta de salida solo se nos producirá la transformación a red de Petri y no todos los datos referentes al análisis.

De no realizar el análisis saldrá una ventana que nos informará de si la transformación se ha realizado de forma satisfactoria y de donde se ha almacenado la transformación. Si se realiza el análisis, la ventana que nos sale al final se parecida a la que se muestra en la Figura 7.6. En esta ventana se muestra la salida que nos devuelve el analizador sobre el análisis realizado.

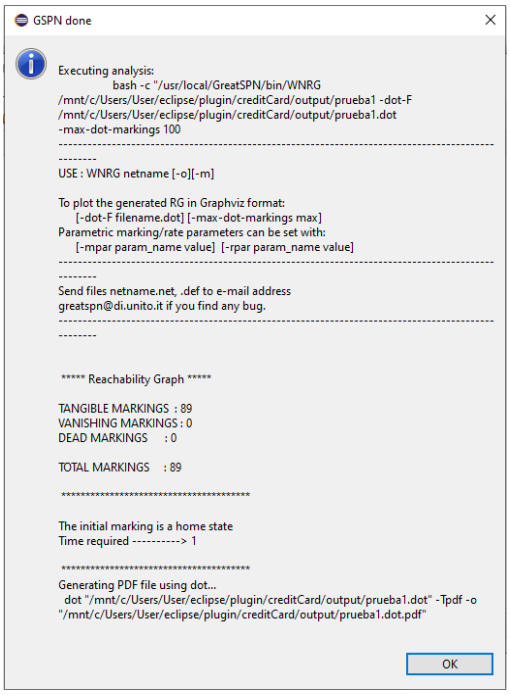


Figura 7.6: Ventana del *plugin* con la información del análisis realizado. Fuente propia.

8 Conclusiones y trabajo futuro

En este trabajo se ha realizado una herramienta para beneficiar al diseño de software más eficiente. Herramientas de este tipo hacen que el diseño y creación de software con más rendimiento pase a ser algo sencillo y fácil de incorporar. Esto permite llevar a segundo plano el rendimiento sin olvidarnos de él, lo que permite focalizarse en otros aspectos del diseño más característicos de cada software.

La transformación con ATL resulta bastante útil por el nivel de abstracción en el manejo de las estructuras de datos. Se echa en falta la posibilidad de hacer una *lazy rule* con la posibilidad de devolver algo a la regla que la llama. Por otro lado, el *debugging* no es nada elegante ni sencillo de realizar. Hablando de Acceleo, es un lenguaje muy limitado en cuestión programática. Sirve para obtener datos e imprimirlos sin más, apenas se permiten realizar modificaciones en la forma en la que se representan los datos. Para este caso, habría sido mejor usar otra tecnología, sobre todo teniendo en cuenta la posible mejora en la representación gráfica. Hablando de herramientas externas, GreatSPN es una herramienta que funciona muy bien, pero resulta difícil hacerlo funcionar en el sistema operativo de Windows. Por último, la herramienta de Eclipse para hacer *plugins*, hace sencillo hacerlos, pero es bastante difícil incorporar funcionalidad de otras tecnologías como Acceleo y ATL. Para concluir con las tecnologías usadas, la documentación de Acceleo y de GreatSPN sobre todo, resulta ser muy escasa, lo que hace su incorporación a un proyecto un serio problema.

Esta herramienta por el momento tiene un marco muy cerrado y posee una funcionalidad muy limitada comparada con lo extenso que es el estándar UML. Por otro lado, algo que está abierto a una mejora es la representación gráfica de la red de Petri, ya que no se ha tomado como prioridad en este proyecto, sino que se ha centrado en la funcionalidad. Por ello, una mejora posible sería realizar un algoritmo para la disposición de los elementos de la red de Petri de acuerdo con las características estructurales de los diagramas de secuencia.

Hablando de la funcionalidad, como trabajo futuro, se podría fácilmente ampliar el *plugin* con otros diagramas de comportamiento dinámico o se podría ampliar para más funcionalidades de los diagramas de secuencia. Hay otras funcionalidades que se han considerado no esenciales de los diagramas de secuencia que se podrían llegar a ser útiles según la ocasión. Con respecto a otros diagramas, este proyecto se ha centrado como se dijo al principio en el diagrama de secuencia por su uso tan extendido y por su utilidad, al representar de forma bastante fiel las interacciones dentro de un sistema software, lo que permite un análisis bastante preciso, pero hay otros diagramas que según el sistema a desarrollar pueden ser útiles, por tanto, es una posibilidad interesante como ampliación.

En el manual de programador adjunto en el Anexo B, se describe de forma breve y sencilla como se podría ampliar y así ir generando una herramienta más y más útil que nos acabe beneficiando a todos.

Con este proyecto he aprendido a manejarme con Eclipse, uno de los entornos de desarrollo más usados en el mundo en la industria del desarrollo software. Acerca de la experiencia con Eclipse me ha parecido una herramienta muy completa, extensa y fácil de usar para la mayoría de las cosas que se pueda hacer, pero me parece que hay cosas que hacer desde cero resultan muy difícil. Por otro lado, he aprendido a hacer un plugin, lo que creo que es muy útil ya que he aprendido a trabajar con varios proyectos a la vez, para acabar teniendo un único producto final. Por otro lado, me ha parecido muy interesante el lenguaje ATL ya que es un lenguaje de reglas que no suele tener un uso tan extendido como otros lenguajes más generales, pero que tiene un potencial enorme para tareas concretas. En resumen, me ha parecido un proyecto que me ha ayudado a ser mejor informático y has sido muy gratificante elaborar una herramienta útil para la comunidad del desarrollo software.

9 Referencias

[1] Martin Fowler, Kendall Scott: UML distilled - a brief guide to the Standard Object Modeling Language (2. ed.). notThenot Addison-Wesley object technology series, Addison-Wesley-Longman 2000, ISBN 978-0-201-65783-8, pp. XV-2.

[2] Object Management Group, «OMG Unified Modeling Language ™ (OMG UML),» Marzo 2015. [En línea]. Available: https://www.omg.org/spec/UML/2.5/PDF. [Último acceso: 4 Marzo 2021].

[3] L. D. Murillo, «Redes de Petri: Modelado e implementación de algoritmos para autómatas programables,» *Tecnología en Marcha,* vol. 21, nº 4, pp. 102-125, 2008.

[4] Chung K.L. (1960) Fundamental defintions. In: Markov Chains with Stationary Transition Probabilities. Die Grundlehren der Mathematischen Wissenschaften (In Einzeldarstellungen mit Besonderer Berücksichtigung der Anwendungsgebiete), vol 104. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-49686-8_1

[5] V. García Díaz, E. R. Núñez Valdez, J. Pascual Espada, C. P. García Bustelo, J. M. Cueva Lovelle, y C. E. Montenegro Marín, «Introducción breve a la ingeniería dirigida por modelos», Tecnura, vol. 18, n.º 40, pp. 127–142, abr. 2014.

[6] J. M. Cueva Lovelle y B. C. Pelayo García-Bustelo, «MDE: Ingeniería dirigida por modelos. Otra forma de construir software,» Noviembre 2008. [En línea]. Available: http://di002.edv.uniovi.es/~cueva/asignaturas/masters/2008/MDE_udistrital.pdf. [Último acceso: 1 Marzo 2021].

[7] L. Baresi y M. Pezzè, «Improving UML with Petri nets,» *Electronic Notes in Theoretical Computer Science ,* vol. 44, nº 4, p. 13, 2001.

[8] S. Distefano, M. Scarpa and A. Puliafito, «From UML to Petri Nets: The PCM-Based Methodology,» in IEEE Transactions on Software Engineering, vol. 37, no. 1, pp. 65-79, Jan.-Feb. 2011, doi: 10.1109/TSE.2010.10.

[9] J. A. Custódio Soares y J. Pascoal Faria, «Automatic Model Transformation from UML Sequence Diagrams to Coloured Petri Nets,» 2018. [En línea]. Available: https://www.scitepress.org/papers/2018/67318/67318.pdf. [Último acceso: 08 Marzo 2021].

[10] Woodside, M., 2013. Tutorial Introduction to Layered Modeling of Software Performance. [En línea] Sce.carleton.ca. Available at: <http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf> [Último acceso 8 Marzo 2021].

[11] Universidad de Granada, «Capitulo 10 Cadenas de Markov,» [En línea]. Available: https://www.ugr.es/~bioestad/_private/cpfund10.pdf. [Último acceso: 12 Marzo 2021].

[12] Eclipse, «Test and Performance Tools Platform,» [En línea]. Available: https://projects.eclipse.org/projects/tptp.platform. [Último acceso: 19 Marzo 2021].

[13] Eclipse Trace Compass, «Trace Compass,» Eclipse, 2017. [En línea]. Available: https://www.eclipse.org/tracecompass/. [Último acceso: 15 Marzo 2021].

[14] S. T. Gilmore, A. Duguid y M. Tribastone, «The PEPA eclipse plug-in,» *ACM SIGMETRICS Performance Evaluation Review,* vol. 36, nº 4, pp. 28-33, Marzo 2009.

[15] E. Gómez-Martinez y J. Merseguer, «ArgoSPE: Model-Based Software Performance Engineering,» de *Petri Nets and Other Models of Concurrency*, Springer, 2006, pp. 401-410.

[16] Cervantes Ojeda, J., Gómez Fuentes, María del Carmen, «Taxonomía de los modelos y metodologías de desarrollo de software más utilizados,» Universidades [en linea]. 2012, (52), 37-47[fecha de Consulta 14 de Junio de 2021]. ISSN: 0041-8935. Disponible en: https://www.redalyc.org/articulo.oa?id=37326902005.

[17] G. Chiola, G. Francheschinis, R. Gaeta y M. Ribaudo, «GreatSPN: graphical editor and analyzer for timed and stochastic Petri nets,» *Performance Evaluation,* vol. 24, pp. 47-68, 1995.

[18] Eclipse, «Eclipse Papyrus Modeling Environment,» 2019. [En línea]. Available: https://www.eclipse.org/papyrus/. [Último acceso: 18 Marzo 2021].

[19] Eclipse, «ATL - a model transformation technology,» [En línea]. Available: https://www.eclipse.org/atl/. [Último acceso: 18 Marzo 2021].

[20] Eclipse, «What is Acceleo?,» 2020. [En línea]. Available: https://www.eclipse.org/acceleo/overview.html. [Último acceso: 18 Marzo 2021].

[21] Eclipse, «Eclipse Modeling Framework (EMF),» [En línea]. Available: https://www.eclipse.org/modeling/emf/. [Último acceso: 19 Marzo 2021].

[22] J. Hillston, "Performance Evaluation Process Algebra," in A Compositional Approach to Performance Modelling, Cambridge: Cambridge University Press, 1996, pp. 17–44.

10 Glosario

ATL	ATL Transformation Language
EMF	Eclipse Modeling Framework
GSPN	Generalized Stochastic Petri Net
MDE	Model Driven Engineering
PEPA	Performance Evaluation Process Algebra
PN2PNPRO	Petri Net To PNPRO
PNML	Petri Net Model Language
PNML2PNPRO	PNML To PNPRO
SD2PN	Sequence Diagram To Petri Net
SPE	Software Performance Engineering
SPN	Stochastic Petri Net
TFG	Trabajo Fin de Grado
UML	Unified Modeling Language
UML2PNML	UML To PNML
WSL	Windows Subsystem for Linux

Anexos

A Manual de instalación y usuario

Manual de instalación

Este manual sirve para conocer como instalar el *plugin* en Eclipse.

1. El proyecto ha sido desarrollado y está preparado para funcionar en la versión de *Eclipse Modelling Tools 2021-03*. Este entorno de trabajo se puede descargar desde su página oficial [https://www.eclipse.org/downloads/].

2. Para la ejecución del *solver* o analizador, es necesario tener un sistema operativo Windows 10. También será necesaria la instalación de WSL (Windows Subsystem for Linux) con la distribución de Ubuntu 20.04 LTS. En este entorno Linux habrá que instalar GreatSPN siguiendo las instrucciones de instalación de [https://github.com/greatspn/SOURCES]. Esto solo será necesario de querer realizar el análisis. De solo querer realizar la transformación a redes Petri, este paso no es necesario.

3. Para la instalación del *plugin*, en el entorno virtual de Eclipse se ha de acceder en la barra de herramientas principal a *Help -> Install New Software -> Add -> Archive* y entonces seleccionar el *plugin* que está comprimido en formato zip. Cuando se haya instalado, será necesario reiniciar Eclipse para que se incorporen los cambios.

Manual de usuario

Este manual sirve para saber cómo utilizar el *plugin*. Se supone de antemano, que el usuario ha creado un diagrama de secuencia de UML, presuntamente con Papyrus. Este *plugin* necesita el archivo *<nombre del archivo>.uml* para funcionar.

A El primer paso es pulsar en la opción de la barra de herramientas UML Analysis -> Analyze, como se muestra en la Figura I.

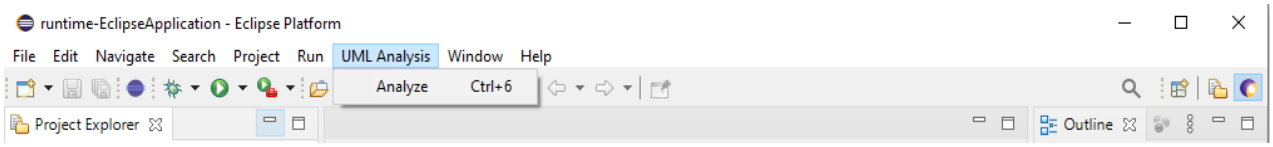


Figura I: Opciones para el analizador de diagramas de secuencia de la barra de herramientas de Eclipse. Fuente propia.

B Tras el paso anterior se abrirá una ventana como la que se muestra en la Figura II.

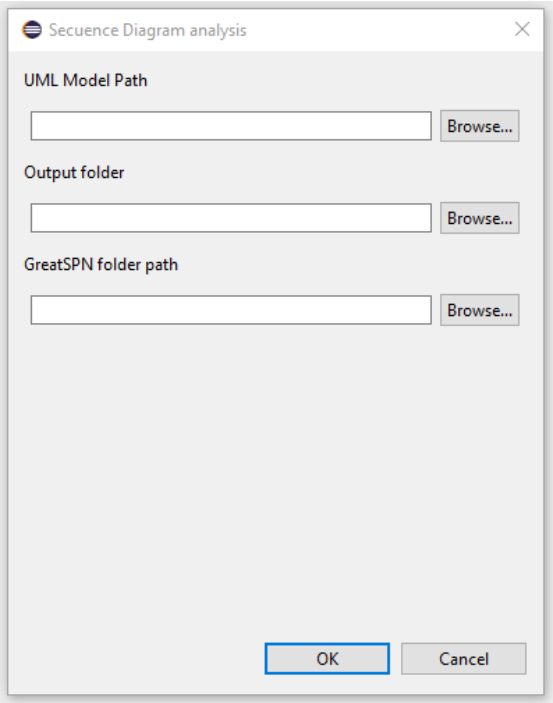


Figura II: Ventana de interacción del *pluginUMLAnalysis* en Eclipse. Fuente propia.

A continuación se deberá introducir la ruta absoluta al archivo *.uml* que contiene el diagrama que se desea analizar en el campo "UML Model Path". También se deberá introducir la ruta absoluta a la carpeta destino donde se almacenará la salida del *plugin*, en el campo "Output folder". Por último, en el campo de "GreatSPN folder path" si se desea realizar el analis de las redes de Petri generadas, deberá introducir la ruta de instalación de GreatSPN en su carpeta *bin*, la cual contiene los *solver* por lo tanto, la ruta a introducir sería por defecto sería */usr/local/GreatSPN/bin/*. Mientras, si no quiere realizar dicho análisis ya que solo desea la transformación del diagrama de secuencia a red de Petri, deberá indicar en este campo el texto "No" o simplemente dejarlo en blanco.

B Manual del programador

Este *plugin* está formado por cinco proyectos, a continuación, se mostrarán las dependencias y funcionalidades de cada uno de ellos.

pnml

- Dependencias:
  - org.eclipse.core.runtime
  - org.eclipse.emf.core

- Funcionalidades: este proyecto dispone del metamodelo *pnmlcoremodel.ecore* el cual es el metamodelo de las redes de Petri y un código autogenerado para su manejo por clases java. Este proyecto no prevé cambios a no ser que se necesitase actualizar a una nueva actualización del metamodelo.

uml2pnml*plugin*

- Dependencias:
  - pnml
  - org.eclipse.core.runtime
  - org.eclipse.uml2.uml
  - org.eclipse.m2m.atl.engine.emfvm
  - org.eclipse.m2m.atl.emftvm

- Funcionalidades: este proyecto contiene la funcionalidad de la primera transformación en ATL. El archivo en el que se escriben las reglas de transformación es *src/uml2pnml/files/uml2pnml.atl*, el código Java autogenerado para el manejo y llamada de las transformaciones ATL es *src/uml2pnmlplugin/files/Uml2pnml.java*. Por último, de querer cambiar las rutas a los metamodelos, se deberá modificar el fichero *src/uml2pnmlplugin/files/Uml2pnml.properties*.

pnml2pnpro

- Dependencias:
  - org.eclipse.core.runtime
  - org.eclipse.emf.core
  - org.eclipse.emf.xmi
  - org.eclipse.ocl
  - org.eclipse.ocl.core
  - org.eclipse.acceleo.common
  - org.eclipse.acceleo.model
  - org.eclipse.acceleo.profiler
  - org.eclipse.celleo.engine
  - com.google.guava
  - pnml

- Funcionalidades: este proyecto contiene la funcionalidad de la segunda transformación en Acceleo. El archivo principal en el que se describe la transformación es *src/pnml2pnpro/main/generate.mtl*, este hace uso de otros archivos que están en *src/pnml2pnpro/files*, los cuales se ocupan cada uno de un elemento de la red de Petri, devidiendose en *generateArcs.mtl*, *generatePlaces.mtl*, y *generateTransitions.mtl*. Por otro lado, el código autogenerado de java para la llamada a esta transformación de Acceleo es *src/pnml2pnpro/main/Generate.java*.

MiniGreatSPN

- Dependencias:
  - org.eclipse.emf.core
  - org.antlr.runtime (4.3.0)

- Funcionalidades: este proyecto contiene un portable de GreatSPN que se usa para crear los archivos .net y .def requeridos por GreatSPN para la llamada al analizador. Este proyecto no prevé ninguna modificación.

UmlAnalysis*Plugin*

- Dependencias:
  - javax.inject
  - org.eclipse.osgi
  - org.eclipse.e4.ui.model.workbench
  - org.eclipse.e4.ui.di
  - org.eclipse.e4.ui.services
  - org.eclipse.e4.core.di.annotations
  - org.eclipse.core.runtime
  - org.eclipse.core.resources
  - org.eclipse.ui
  - pnml
  - uml2pnml*plugin*
  - MiniGreatSPN

- Funcionalidades: este proyecto es el que integra la interfaz de usuario con Eclipse y desde la que se llaman a todas las funcionalidades. De querer modificar el cuadro de dialogo de introducción de datos, se deberá modificar el archivo *src/dialogs/AnalysisDialog.java*. de quererse modificar las llamadas que se realizan, se deberá modificar el *handler src/UmlAnalysisPlugin/handlers/AnalysisHandler.java* aquí se puede añadir funcionalidad o cambiar la funcionalidad como por ejemplo el uso de otro analizador de redes Petri. De querer cambiar el menú de barras de tareas o el comando, se deberá modificar el archivo *plugin.xml* en la carpeta raíz del proyecto.

C Otras pruebas y resultados

Se han ideado cinco ejemplos diferentes con el objetivo de comprobar el correcto funcionamiento del *plugin* desarrollado y el completo cumplimiento de los requisitos. Las tres pruebas desarrolladas junto con los resultados de sus transformaciones en redes de Petri se pueden observar en los diagramas siguientes, en las figuras III-XII disponibles en este anexo.

El *Ejemplo1*, que se puede observar en la Figura III, sirve para comprobar la implementación, por un lado, como todos los ejemplos, del requisito más básico que es el RF1.1.1 el del envio de mensajes, y además, vemos que gracias a la incorporación de un *CombinedFragment* de tipo *loop* cubre el requisito RF1.1.4.

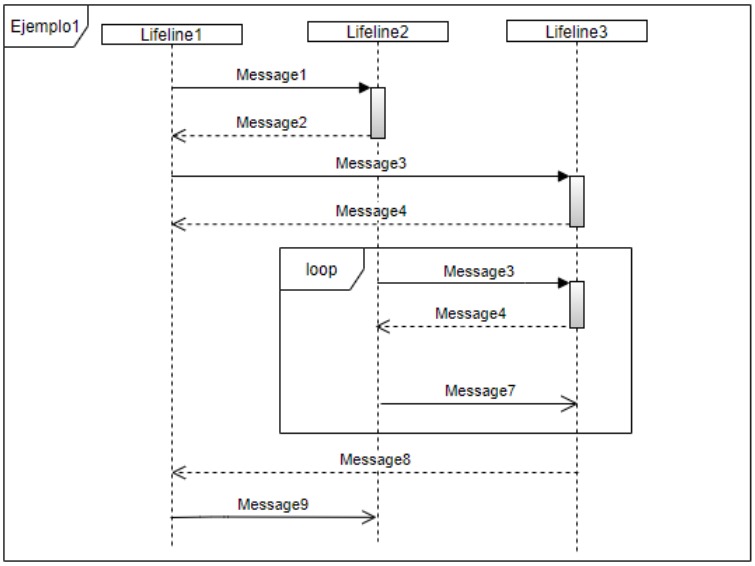


Figura III: *Ejemplo1*. Diagrama de secuencia con *CombinedFragment* tipo *loop*. Fuente propia.

El *Ejemplo2*, representado en la Figura IV, sirve para comprobar el requisito RF1.1.5 al incorporar un *CombinedFragment* de tipo *alt*.

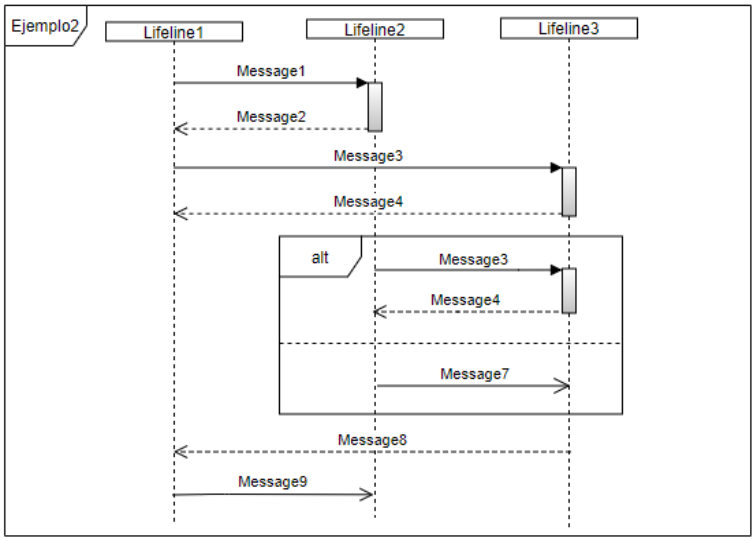


Figura IV: *Ejemplo2*. Diagrama de secuencia con *CombinedFragment* de tipo a*lt*. Fuente propia.

El *Ejemplo3*, representado en la Figura V, sirve para comprobar el requisito RF1.1.3 al incorporar un *CombinedFragment* de tipo *par*.

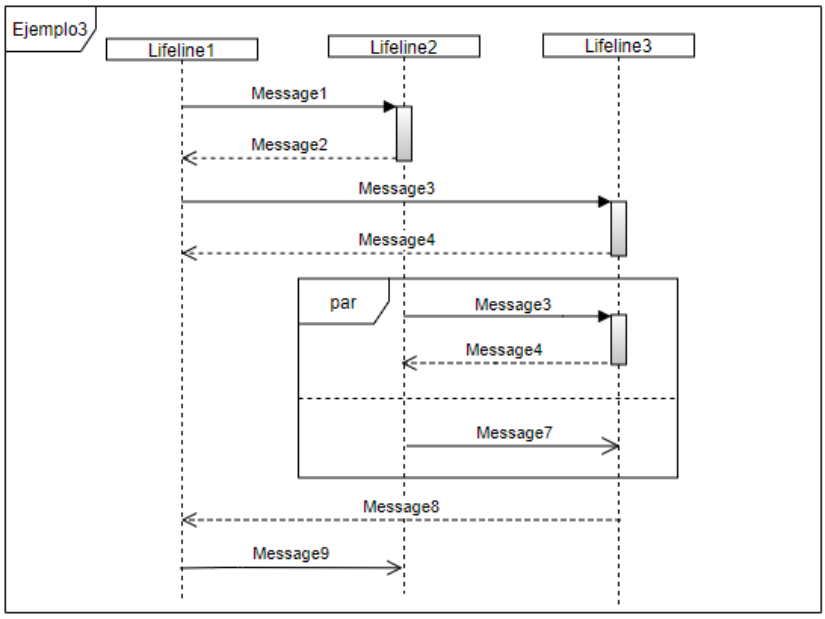


Figura V: *Ejemplo3*. Diagrama de secuencia con *CombinedFragment* de tipo *par*. Fuente propia.

El *Ejemplo4*, representado en la Figura VI, sirve para comprobar el requisito RF1.1.2 al incorporar una referencia a un diagrama de secuencia por medio de un *InteractionUse*.

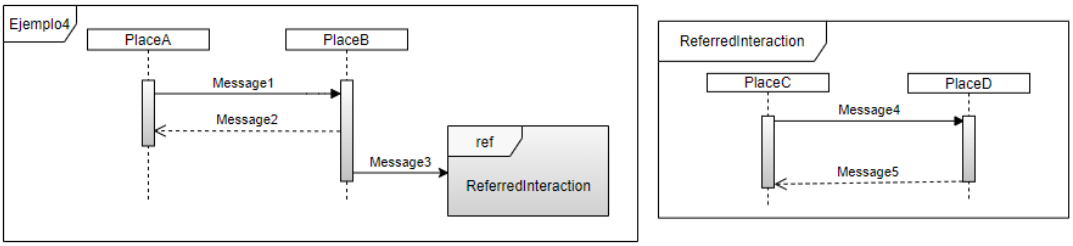


Figura VI: *Ejemplo4*. Diagrama de secuencia con referencia a otro diagrama de secuencia por medio de un *InteractionUse*. Fuente propia.

El *Ejemplo5*, representado en la Figura VII, sirve para comprobar el requisito RF1.1.6 al incorporar la anidación de dos *CombinedFragment*.

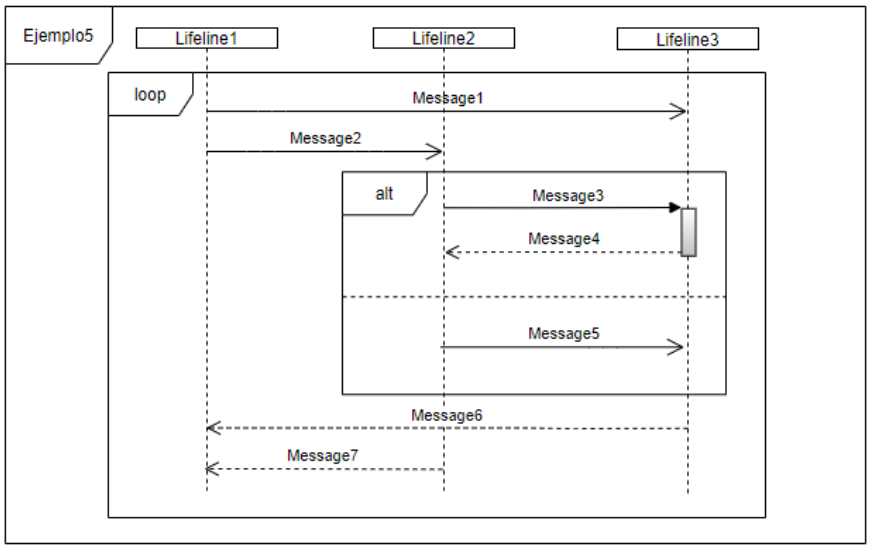


Figura VII: *Ejemplo5*. Diagrama de secuencia con dos *CombinedFragment* anidados. Fuente propia.

Tras ejecutar el *plugin* para realizar las transformaciones, los resultados en redes de Petri son las siguientes.

En el ejemplo de la Figura VIII, se ve la transformación de todos los envíos de mensaje como en los demás ejemplos, lo particular es la transformación del *loop*, en este ejemplo, no se añadió un numero de repeticiones por medio de MARTE, por tanto, no tendría la funcionalidad buscada de representación de un bucle, ya que como vemos, tanto el arco de entrada como de salida del *CombinedFragment* tienen una multiplicidad de 1. Este ejemplo, por tanto, cumple con el requisito RF1.1.1 y estructuralmente el RF1.1.4, el cual se completa añadiéndole MARTE.

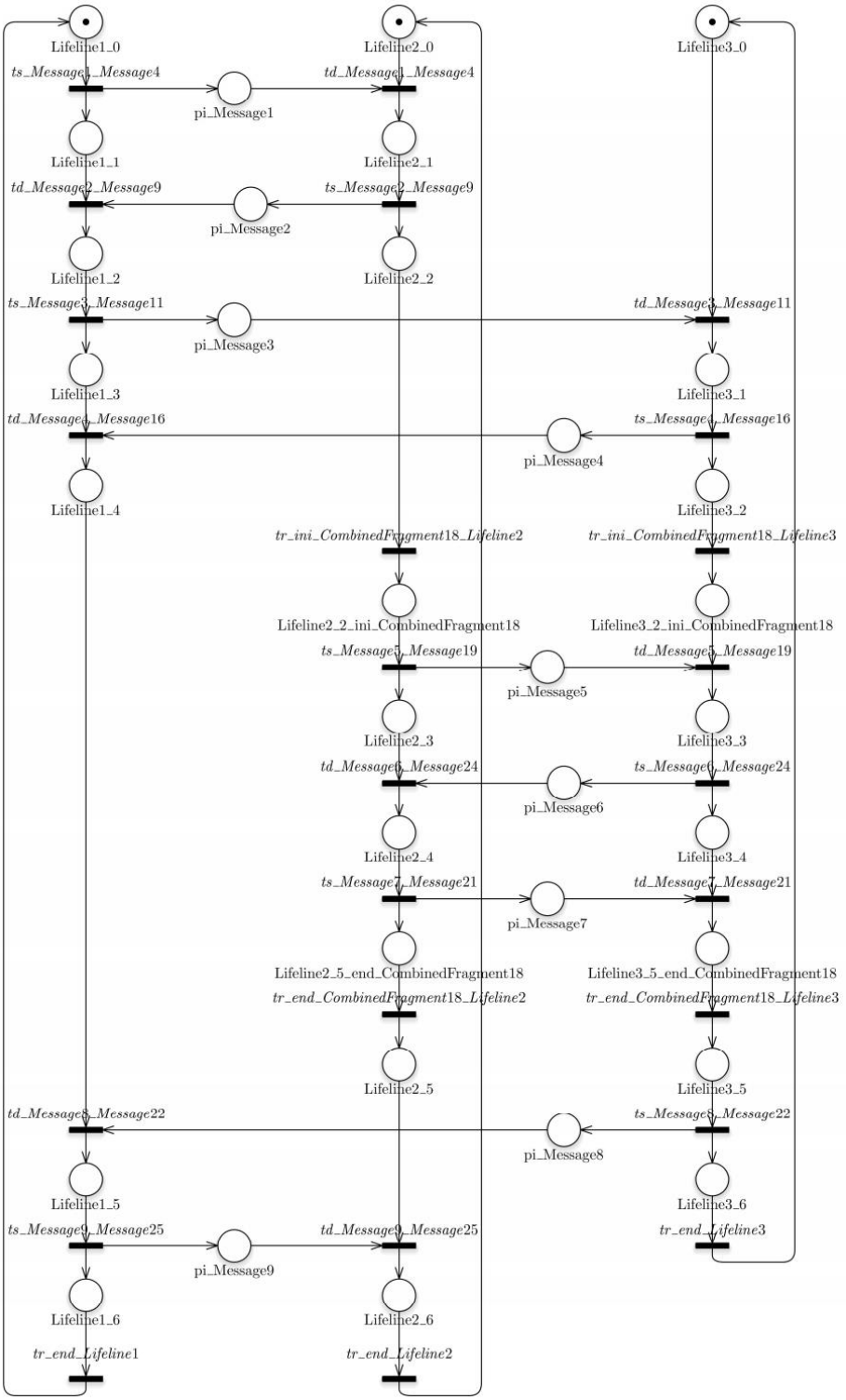


Figura VIII: Red de Petri del *Ejemplo1*. Fuente propia.

Como vemos, lo particular de este ejemplo es el *Alt*, que se realiza siguiendo la estructura de la Figura 5.11. Esta prueba por tanto demuestra la funcionalidad del requisito RF1.1.5.

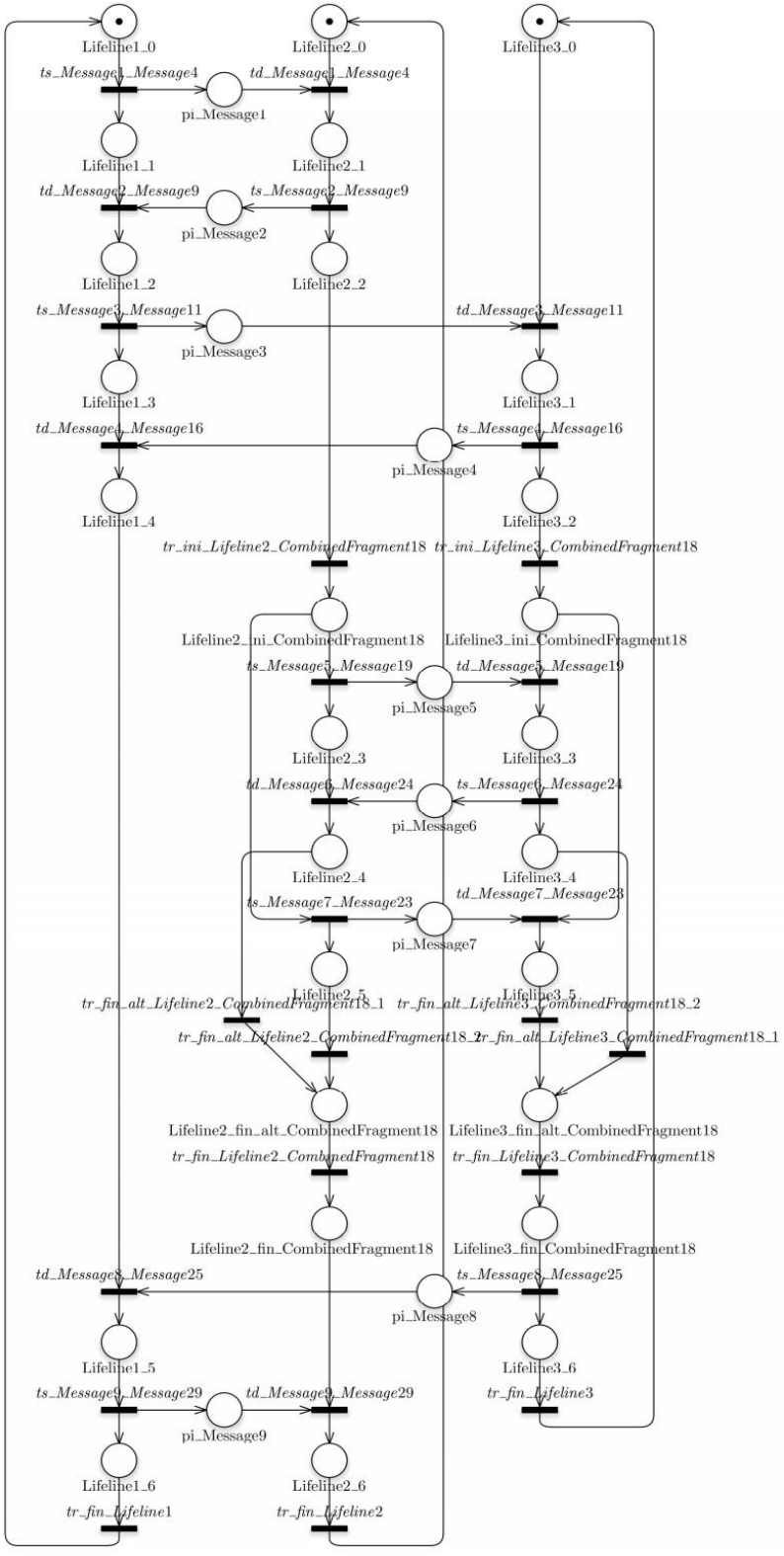


Figura IX: Red de Petri del *Ejemplo2*. Fuente propia.

Este ejemplo de la Figura X, la particularidad es el *Parallel*, se realiza siguiendo la estructura de la Figura 5.7. Este ejemplo, por tanto, confirma el funcionamiento del requisito RF1.1.3.

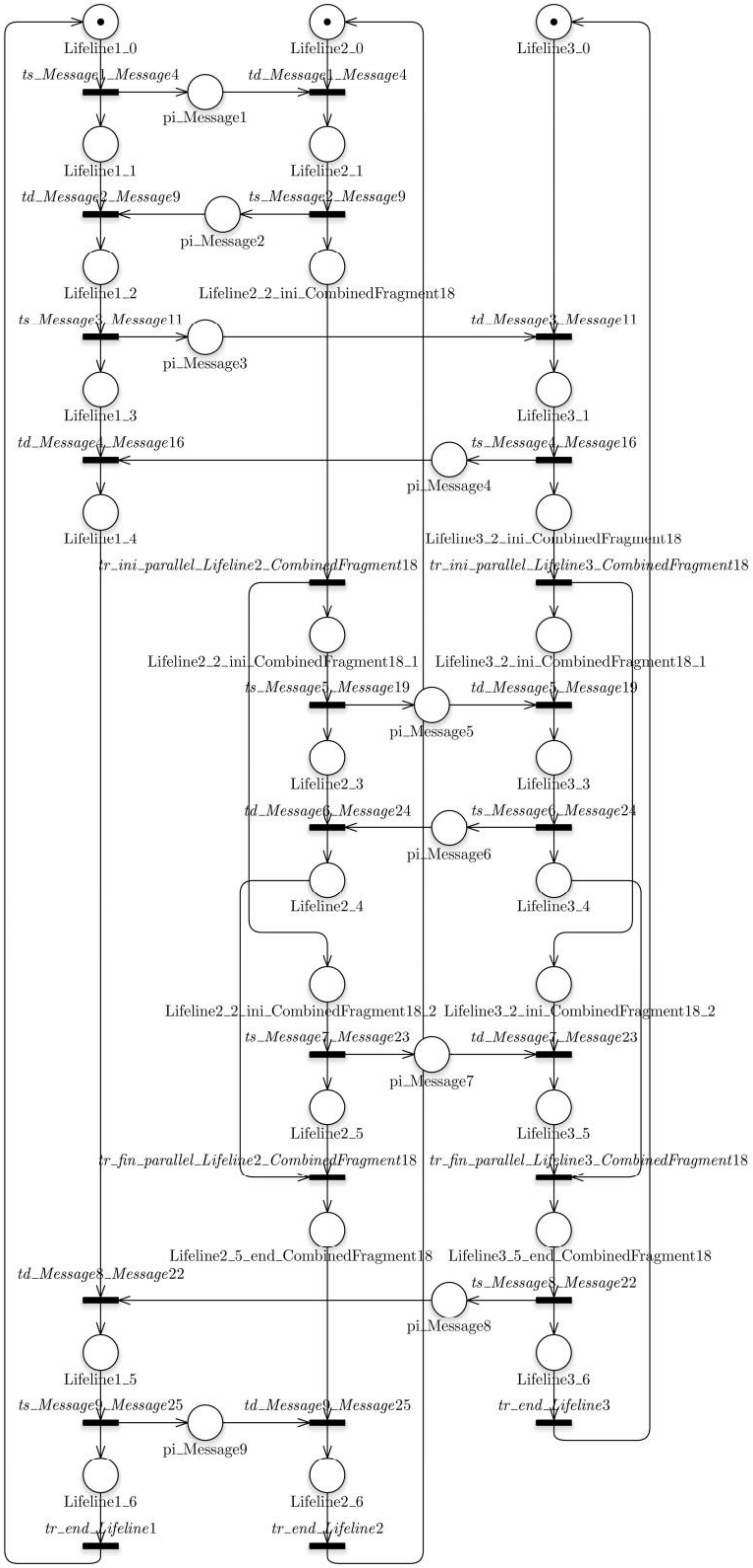


Figura X: Red de Petri del *Ejemplo3*. Fuente propia.

El ejemplo de la Figura XI como vemos es la transformación de un ejemplo básico de llamada a un *Interaction* referenciado. La transformación por tanto ha seguido la estructura de la Figura 5.5. Esta funcionalidad es la que se contempla en el requisito RF1.1.2 y por tanto queda confirmado su correcto funcionamiento.

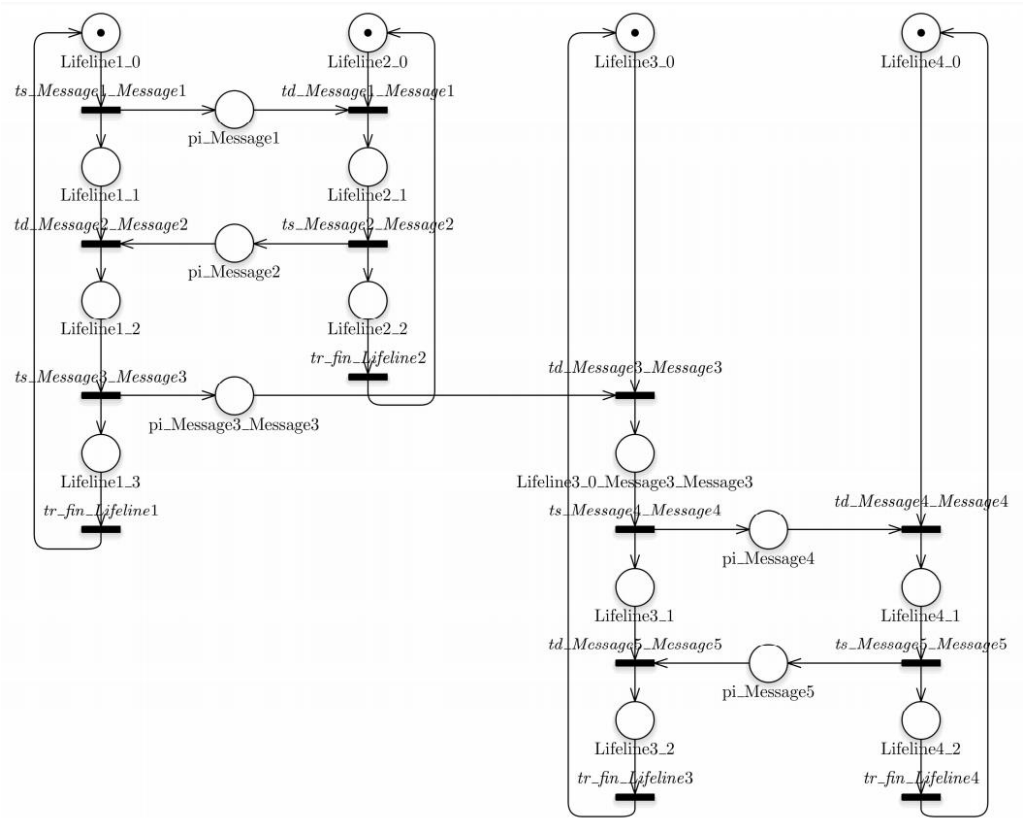


Figura XI: Red de Petri del *Ejemplo4*. Fuente propia.

En el ejemplo de la Figura XII, se muestra la transformación de un diagrama que contiene un CombinedFragment dentro de otro. Como vemos, su correcto funcionamiento nos confirma el requisito RF1.1.6.

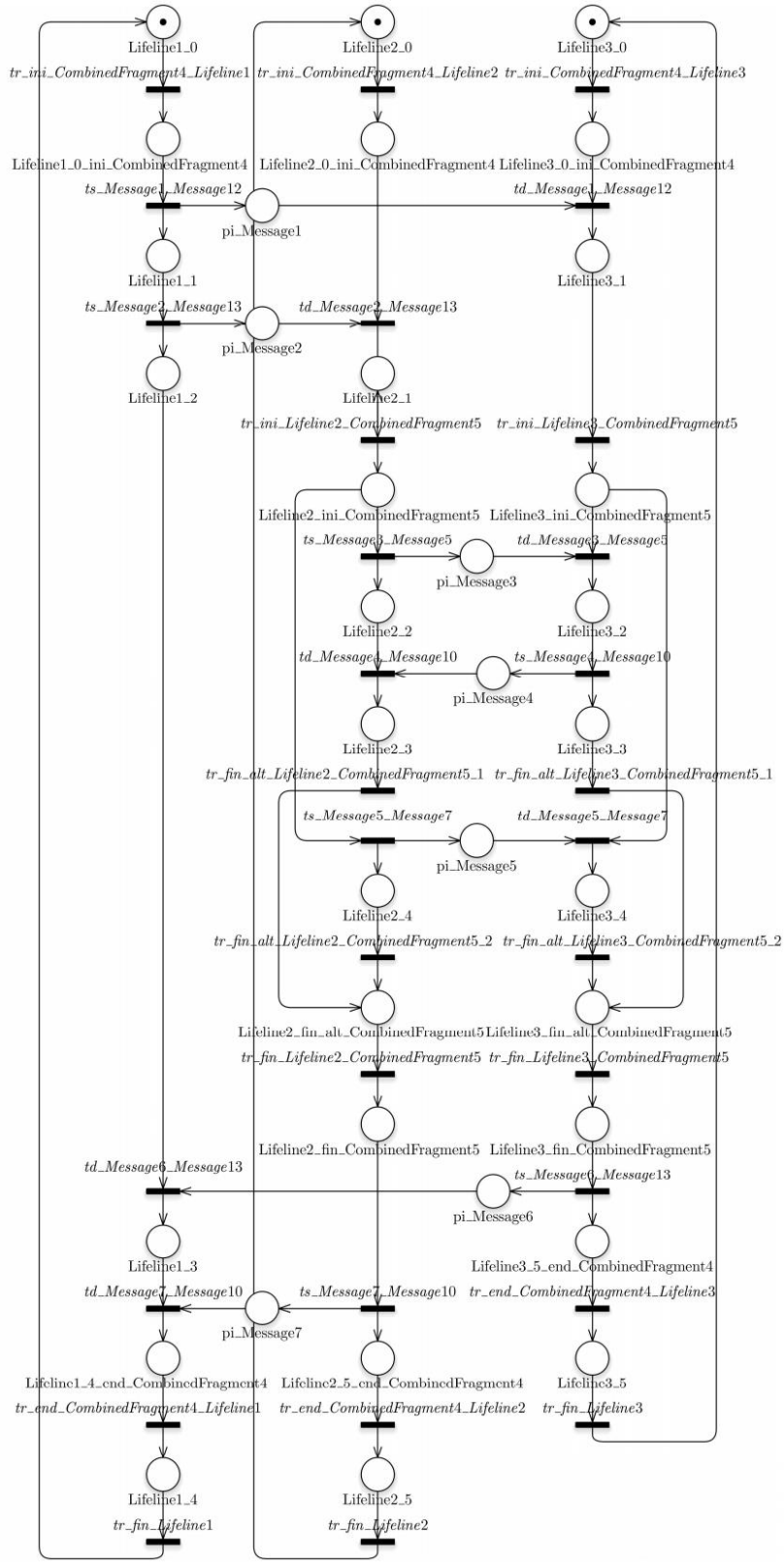


Figura XII: Red de Petri del *Ejemplo5*. Fuente propia.