Graduado en Ingeniería Informática

Aplicación Web para el Análisis Unificado de Procesos BPMN

Web Application for the Unified Analysis of BPMN processes

Realizado por
Pablo Espinosa Tarrío

Tutorizado por
Francisco Javier Durán Muñoz

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, junio de 2022

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

Aplicación web para el análisis unificado de procesos BPMN

Web application for the unified analysis of BPMN processes

Realizado por
Pablo Espinosa Tarrío

Tutorizado por
Francisco Javier Durán Muñoz

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2022

Fecha defensa: julio de 2022

Abstract

A **business process** is a set of structured activities that have the goal of developing a certain product or software. **BPMN** is the *de facto* notation for graphically designing and modeling these processes. The importance of business processes within organizations raises the need of start conducting formal analyzes on them in order to be able to design them efficiently and error-free.

Therefore, the purpose of this work is to develop a **web application** that allows, in a simple and efficient way, the creation and visualization of BPMN diagrams and to execute formal analyzes on them in real time. The analysis will be focused on automatically verifying certain properties of interest, such as execution time, degree of parallelism, and resource usage of the processes, in addition to allowing the verification of linear temporal logic (LTL) propositions on them to reason about their behaviour.

An incremental iterative methodology has been followed for the development of the application, using **Node.js** to develop the application's web server, **Maude** to carry out the formal analyzes of the BPMN processes, and **Java** for reading the XML files that contain the information of each diagram and generate their respective representation in Maude format.

Keywords:

BPMN, Maude, Node.js, formal analysis, business processes.

Resumen

Un **proceso de negocio** es un conjunto de actividades estructuradas que tienen como objetivo desarrollar un determinado producto o software. **BPMN** es la notación de *facto* para diseñar y modelar gráficamente estos procesos. La importancia de los procesos de negocio dentro de las organizaciones hace que se comiencen a realizar análisis formales sobre ellos para poder llevarlos a cabo de manera eficaz y libre de errores.

Por ende, el objetivo de este trabajo es desarrollar una **aplicación web** que permita, de forma sencilla y eficiente, la creación y visualización de diagramas BPMN para poder ejecutar sobre ellos análisis formales en tiempo real. Dichos análisis, estarán focalizados en verificar automáticamente ciertas propiedades de interés, como el tiempo de ejecución, el grado de paralelismo y el uso de recursos de los procesos, además de permitir verificar proposiciones de lógica temporal lineal (LTL) sobre ellos para razonar acerca de su funcionamiento.

Para el desarrollo de la aplicación se ha seguido una metodología iterativa incremental, utilizando **Node.js** para desarrollar el servidor web de la aplicación, **Maude** para realizar los análisis formales sobre los procesos BPMN, y **Java** para leer los ficheros XML que contienen la información de cada diagrama y generar su respectiva representación en formato Maude.

Palabras clave:

BPMN, Maude, Node.js, análisis formales, procesos de negocio.

Índice

1. Introducción **9**

1.1. Motivación . . . 9

1.2. Objetivos . . . 10

1.3. Metodología . . . 12

1.4. Tecnologías usadas . . . 13

1.4.1. Servidor web . . . 14

1.4.2. Cliente web . . . 16

1.4.3. Parser . . . 17

1.5. Estructura de la memoria . . . 17

2. Fases del desarrollo **19**

2.1. Iteración cero . . . 19

2.1.1. Estudio del estado del arte . . . 19

2.1.2. Estudio de los análisis formales . . . 20

2.1.2.1. Papers de la primera iteración . . . 21

2.1.2.1.1. Notación Maude (sintaxis) . . . 24

2.1.2.1.2. Conjunto de reglas (semántica) . . . 25

2.1.2.1.3. Simulación . . . 27

2.1.2.1.4. Análisis formales . . . 32

2.1.2.2. Papers de la segunda iteración . . . 34

2.1.2.2.1. Subconjunto BPMN . . . 34

2.1.2.2.2. Notación Maude . . . 36

2.1.2.2.3. Semántica y Simulación . . . 38

2.1.2.2.4. Análisis formales . . . 39

2.1.3. Documento General de Requisitos (DGR) . . . 41

2.1.4. Diagramas UML . . . 42

2.1.5. Diagramas de secuencia . . . 46

2.1.6. Figma . . . 48

2.1.7. Selección de los requisitos a implementar en cada iteración 50
2.2. Iteración uno 51
2.2.1. Análisis de requisitos 51
2.2.2. Implementación 55
2.2.2.1. Configuración inicial 55
2.2.2.2. Página de bienvenida 56
2.2.2.3. Servidor web 57
2.2.2.4. Página principal 58
2.2.2.5. Parser 68
2.2.2.6. Análisis formales implementados 73
2.2.2.7. Resultados obtenidos 75
2.2.2.8. Asincronismo en Node.js 79
2.3. Iteración dos 83
2.3.1. Análisis de requisitos 83
2.3.2. Implementación 87
2.3.2.1. Cliente web 87
2.3.2.2. Parser 92
2.3.2.3. Análisis formales implementados 101
2.3.2.4. Resultados obtenidos 112

3. Conclusiones y Líneas Futuras **115**
3.1. Conclusiones 115
3.2. Líneas futuras 116
3.2.1. Puesta en producción de la aplicación 116
3.2.2. Procesamientos más refinados en las salidas 116
3.2.3. Nuevos algoritmos de optimización 117
3.2.4. Liveness 118

Apéndice A. Documento General de Requisitos **123**

Apéndice B. Manual de Instalación **129**
B.1. Sistema operativo 130

B.2. Versiones y las herramientas necesarias 130
B.3. Instalación de dependencias 130
B.4. Despliegue del servidor 131

Apéndice C. Manual de Usuario **133**

C.1. Landing Page 134
C.2. Página principal 135
C.2.1. Análisis presentados en la primera iteración 138
C.2.2. Análisis presentados en la segunda iteración 138

1

Introducción

1.1. Motivación

Un **proceso de negocio** engloba el conjunto de actividades que realiza una organización para llevar a cabo sus objetivos. Por tanto, son cruciales para su correcto funcionamiento, pudiendo llevarlas al éxito o al fracaso. Es por ello que surge el **modelado de procesos de negocio**, un área importante dentro de la ingeniería del software que se dedica a desarrollar productos basados en flujos de trabajo como sistemas distribuidos y de información.

Business Process Model and Notation (BPMN) [19] es la notación *de facto* para diseñar y modelar gráficamente estos procesos de negocio y fue publicado como un estándar ISO en el año 2013. La importancia de los procesos de negocio crea la necesidad de realizar análisis formales sobre sus modelos para poder evitar ejecuciones erróneas y optimizar los recursos que utilicen. Es por ello que aproximadamente desde el año 2008, Francisco Durán, profesor de la E.T.S. de Ingeniería Informática y encargado de tutorizar este trabajo, y sus colaboradores, han estado desarrollando una serie de técnicas de análisis para diferentes propiedades de interés sobre procesos BPMN, entre ellas:

- Análisis temporal: Cálculo del tiempo mínimo, máximo y medio de un proceso BPMN.
- Grado de paralelismo: Cálculo del máximo número de nodos que pueden ser ejecutados por el proceso en un instante de tiempo.
- Model checking: Comprobación de propiedades de lógica temporal lineal sobre modelos BPMN.
- Liveness: Propiedad que garantiza que cuando la ejecución de un proceso termina no quedan tareas activas ni mensajes pendientes de ser recibidos.

- Análisis de recursos: Como resultado de este análisis se obtiene información sobre el uso de los recursos, como el tiempo de uso, el tiempo medio de ejecución para un número de dado de simulaciones concurrentes, el tiempo de bloqueo en las actividades y en las puertas de eventos, etc.

- Optimización de recursos: Uso de las métricas anteriores para, entre otras cosas, calcular combinaciones de recursos óptimas en función de su coste y tiempo de ejecución, y posibles cuellos de botella en los procesos.

La mayoría de estos algoritmos están desarrollados en **Maude** [34], un lenguaje declarativo basado en lógica de reescritura que permite un alto nivel de abstracción y rendimiento, lo cual facilita la implementación de sistemas concurrentes y no deterministas. Estas características son clave para simular un alto volumen de procesos BPMN de manera relativamente eficiente. Sin embargo, a pesar del potencial e interés de estas técnicas de análisis y su implementación parcial, su uso es bastante complicado, ya que involucra diferentes herramientas y scripts escritos en diferentes lenguajes. Además, las distintas técnicas utilizan diferentes presentaciones de BPMN en Maude, y el uso de los distintos métodos de análisis requiere conocimiento experto y uso de varias herramientas. Por ende, el objetivo principal de este proyecto es desarrollar una aplicación web que automatice, de forma sencilla y eficiente, las diferentes técnicas de análisis sobre procesos BPMN utilizando una interfaz unificada.

1.2. Objetivos

La aplicación resultante de este proyecto tiene como objetivo principal ser la interfaz que conecta los diagramas BPMN con los posibles análisis formales que se pueden realizar sobre ellos. Para conseguirlo, a alto nivel:

Se permite el **modelado de diagramas BPMN** tanto creados en la aplicación como ya existentes, siempre que sigan el estándar BPMN 2.0. Para ello, en lugar de desarrollar nosotros mismos la representación gráfica, se utiliza un conjunto de librerías del proyecto *bpmn-js* [3], desarrollado por Camunda. Esta librería permite la gestión de modelos BPMN (creación, modificación y guardado) y genera una representación del modelo BPMN en formato XML respetando el estándar.

Las técnicas de análisis hacen uso de información temporal, probabilística y de recursos como parte de los modelos, la cual no está contemplada en el estándar BPMN 2.0 y por tanto tampoco en la herramienta (bpmn-js). Por tanto, se utilizan las facilidades que aporta el proyecto de Camunda para **extender su librería** y agregar dicha información a los ficheros XML que representan los diagramas.

Se exponen los **resultados de los análisis** sobre los diagramas BPMN (tiempo de ejecución, grado de paralelismo, model checking con propiedades LTL (Linear Temporal Logic) [13], liveness y análisis de recursos). Estos análisis fueron desarrollados por mi tutor y su equipo, de forma que mi labor es integrarlos en la aplicación utilizando una arquitectura *cliente-servidor*. De manera que, en el cliente, el usuario seleccione qué análisis quiere realizar sobre su diagrama, el cual es ejecutado en el servidor y cuyo resultado es renderizado de vuelta en la página web.

Las salidas Maude de los análisis tienen mucha información que requiere conocimiento experto para ser de utilidad. Es por esto que, como parte de este trabajo, se realizan una serie de programas en JavaScript [20] para **manipular y depurar los datos** proporcionando información de calidad como tablas, gráficas y contraejemplos.

Se permite **traducir** las especificaciones XML (eXtensible Markup Language) [18] de los procesos BPMN a representaciones Maude. Los diagramas BPMN se representan internamente mediante archivos XML, pero los algoritmos Maude existentes requieren de una entrada con un formato específico para su ejecución. Para la ejecución automática de los análisis desde la aplicación web se ha desarrollado en Java [32] un algoritmo que traduce cualquier fichero XML que represente un diagrama válido en su correspondiente formato Maude. Se ha elegido Java para esta cuestión porque nos permite, de forma sencilla, crear una jerarquía de clases con los nodos BPMN y su correspondiente representación en Maude, haciendo el código más legible y extensible.

Aunque la mayoría de las técnicas de análisis han sido implementadas en Maude y el objetivo de este TFG es unificar su uso, hay otros métodos que no han sido implementados y que se han desarrollado en su totalidad como parte del proyecto. En concreto, en uno de los trabajos se menciona a alto nivel un **algoritmo de optimización** multi-objetivo con el que encontrar la combinación de recursos que minimice el coste y el tiempo de ejecución. Siguiendo esta idea, se implementa como parte de este trabajo, un algoritmo de optimización de *descenso de*

gradiente que permite encontrar un mínimo local sin necesidad de analizar todas las combinaciones de recursos, basándose en distintos parámetros especificados por el usuario a través de la aplicación, como son el número mínimo y máximo de recursos que quiere tener en cuenta y el coste por hora que le supone dicho recurso.

1.3. Metodología

En ingeniería del software, se define a las metodologías software como un marco de trabajo utilizado para estructurar, planear y controlar el proceso de desarrollo de nuevos sistemas de información.

En este proyecto se utiliza una metodología **iterativa incremental**, con el objetivo de obtener cuanto antes una versión funcional del proyecto para posteriormente continuar con una evolución sostenida añadiendo nuevas características.

De esta forma, se organizó el trabajo en **dos fases principales** para obtener una versión funcional del proyecto en cada una de ellas. Cada una de estas fases son descompuestas a su vez en sucesivas iteraciones, donde cada iteración consiste en los pasos habituales de análisis, diseño, implementación y pruebas. Al plantear el desarrollo como un proceso iterativo guiado por funcionalidades se han podido realizar pruebas de cada uno de los tipos de análisis en cuanto estos estaban disponibles.

Antes de explicar la metodología de cada una de estas dos iteraciones, cabe destacar que existe una fase previa que hemos denominado como **iteración 0** y que es un tanto diferente.

En primer lugar, se comenzó con la búsqueda de proyectos o *trabajos similares*, de los cuales se pudiera obtener información de partida e investigar cómo se podría diferenciar nuestra aplicación de ellos para así aportar más valor. En segundo lugar, se continuó con el estudio de cada uno de los *papers* publicados por el tutor de este trabajo y sus colaboradores. Como se ha comentado previamente, tratan acerca de la elaboración o propuesta de análisis formales que pueden realizarse sobre diagramas BPMN. Era importante conocer exactamente cómo funcionaba cada análisis y cuál era su formato de entrada-salida. Después, se decidió elaborar documentación importante para el proyecto como el *Documento General de Requisitos* y diversos diagramas *UML* y de *secuencia* que representarían la arquitectura y el funcionamiento de ciertos programas y partes del proceso. Finalmente, se realizó un primer diseño del cliente de la aplicación web usando la herramienta gráfica textitFigma [15].

Las otras dos iteraciones (de alto nivel) del proyecto se centran principalmente en el desarrollo la aplicación, creando un entorno donde los usuarios puedan crear y cargar diagramas BPMN que puedan ser analizados con las técnicas implementados en Maude de manera sencilla y eficiente.

En el apartado relacionado con las fases del trabajo se habla en detalle de las fases e iteraciones del proyecto pero, de forma resumida, estas se basarán en:

- Iteración 1: Implementación de las técnicas de análisis temporal, de grado de paralelismo, de model checking con propiedades LTL y de la verificación de la propiedad conocida como liveness. Esto supone una gran cantidad de trabajo porque para obtener una versión funcional que implemente estas técnicas se debe desarrollar una primera versión de la aplicación web al completo, incluyendo el servidor, el cliente integrado con el modeler de Camunda, el script en Java para traducir de XML a Maude y por último implementar todas las comunicaciones entre cada componente.

- Iteración 2: En esta segunda fase se parte de la versión anterior de la aplicación, por lo que la cantidad de trabajo será menor, pero más técnica. Esto se debe a que en esta fase se integran con lo anterior las técnicas de análisis de recursos, las cuales requieren un subconjunto de los procesos BPMN bastante más complejo y por lo tanto necesitamos extender en gran medida la transformación de XML a Maude. Además, se diseña un sistema que obtiene la salida del análisis de uso de los recursos y muestra los resultados mediante gráficas y se implementa el algoritmo de optimización de descenso de gradiente mencionado con anterioridad.

1.4. Tecnologías usadas

De forma resumida, la aplicación web se desarrolla en **JavaScript**, utilizando **Node.js** [6] para el back-end junto con el framework de **Express.js** [14] para crear el servidor HTTP de forma rápida y sencilla. Además, se utiliza **Webpack** [35] para unificar los ficheros tanto en el servidor como en el cliente, para organizar las dependencias en el cliente con las librerías de Camunda mencionadas anteriormente, y para hacer "minify y uglify" del código para hacerlo más ligero y poder utilizarlo en producción, cosa que está propuesta como trabajo futuro.

A nivel funcional y tecnológico, se diferencian tres componentes principales dentro de la aplicación web: el **servidor**, el **cliente** y el script encargado de traducir los ficheros XML que representan a los diagramas a su correspondiente representación en Maude, de ahora en adelante denominado **Parser**, por simplicidad.

1.4.1. Servidor web

Se ha decidido utilizar **Node.js** para el desarrollo del servidor. Node.js [16] es un entorno de ejecución basado en JavaScript, es orientado a eventos, asíncrono y utiliza el **motor V8 de Google** [17]. Este motor desarrollado en primer lugar para el famoso navegador Google Chrome fue una revolución en el desarrollo web debido al gran aumento de velocidad de procesamiento que aportó. En primer lugar, transforma el código JavaScript en una lista de tokens, que posteriormente pasan a formar parte de un abstract syntax tree, que permite generar el código *ByteCode* que el navegador interpreta y ejecuta de forma más eficiente. Todo esto hace de Node.js un entorno altamente escalable.

Además, se utiliza su manejador de paquetes, **npm** [1], que permite manejar de forma eficiente y sencilla un gran número de dependencias.

Para la creación del servidor, se ha utilizado el framework **Express.js** que permite crear servidores HTTP e infraestructuras web flexibles y eficientes en Node.js de manera rápida y sencilla, al mismo tiempo que ofrece un gran número de llamadas a distintas APIs y de configuraciones *middleware*.

Para gestionar las versiones de Node.js y npm se ha usado **Node Version Manager (nvm)** [26], un software que permite mantener en un mismo dispositivo distintas versiones de Node,js y de npm, ayudando también a resolver algunos problemas de privilegios que genera npm, facilitando el desarrollo.

Una herramienta clave para el desarrollo del proyecto es **WebPack** [27] [25]. Para entender por qué, es necesario conocer cómo se organizan los módulos y ficheros JavaScript en los navegadores y fuera de ellos. Antes de la creación de Node.js, los programas escritos en este lenguaje solo podían ser ejecutados en navegadores siguiendo dos estrategias:

- Un script para cada funcionalidad, lo cual no es una solución muy escalable debido a que cargar y renderizar muchos scripts puede causar *bottlenecks* a nivel de red.

- Un único script para todo el código, lo cual evita el anterior problema pero puede llegar a ser muy complicado de mantener.

Con ambas estrategias, los archivos iban enlazados normalmente a ficheros **HTML** para que fueran usados en las páginas. Cuando Node.js entró en juego, el paradigma cambió totalmente, ya que como JavaScript ya no solo se ejecutaba en el navegador había que encontrar una manera de referenciar y usar código que se encontrara en otros archivos. La solución vino de la mano de la sentencia **require**, enmarcada dentro del recién creado **CommonJS** [5], un proyecto que tenía como objetivo establecer convenios sobre cómo trabajar sobre el ecosistema de módulos de JavaScript fuera de los navegadores. Esto era una buena solución para proyectos que usarán el lenguaje en el lado del servidor, sin embargo, no tenía soporte para los navegadores.

Posteriormente surgen los **ECMAScript Modules (ECM)** [24], la forma estándar de utilizar módulos tanto fuera como dentro de los navegadores. Sin embargo, no está soportado por todos los navegadores y versiones o librerías de JavaScript, pudiendo llegar a ocurrir que un mismo programa funcione bien en un navegador y mal en otro. Esto se denomina *backwards compatibility* y es uno de los grandes problemas de añadir características nuevas a los lenguajes de programación.

Para solucionar todos estos problemas surgen herramientas como webpack. Webpack es un software que te permite utilizar módulos, haciendo el código más legible y mantenible, para posteriormente empaquetar todo el código de tu aplicación en un solo fichero, solucionando el bottleneck a nivel de red. Al mismo tiempo, crea un grafo de dependencias, resolviéndolas por completo, y se combina con loaders como babel para transformar el código en versiones antiguas compatibles con todos los navegadores. En el caso concreto de este proyecto, también es muy útil para resolver las dependencias del proyecto bpmn-js de Camunda que utilizamos para integrar su modeler en nuestro cliente web.

Para los análisis formales se ha usado Maude, un lenguaje declarativo basado en lógica de reescritura que permite un alto grado de expresividad, sin perder simplicidad y eficiencia. En Maude se pueden representar sistemas secuenciales, concurrentes, deterministas y no deterministas [23]. Estas características hacen que sea una opción muy interesante para el trabajo propuesto debido a que se pueden representar y lanzar numerosas simulaciones concurrentes de procesos al mismo tiempo de manera simple y extensible.

1.4.2. Cliente web

En el lado del cliente se ha seguido un enfoque minimalista, usando puro HTML, Css3 y JavaScript, junto con ciertas librerías.

Destacar en primer lugar el uso de **jQuery** [21], una librería ligera, eficiente y simple de usar que permite simplificar determinadas acciones muy comunes en el lado del cliente como modificar y acceder al *Document Object Model* (DOM), interactuar con los documentos HTML, manejar eventos, crear animaciones, etc. Además, jQuery realiza por sí mismo ciertas manipulaciones sobre el código que hace que sea portable entre diferentes navegadores.

Como se ha comentado, **Bpmn-js** es un proyecto desarrollado y mantenido por *Camunda* que permite integrar en cualquier aplicación web un *modeler* con el que cualquier usuario puede crear, editar, visualizar y cargar diagramas BPMN que respeten el estándar 2.0. Es un producto de código abierto distribuido bajo licencia gratuita que permite a los usuarios reproducir, copiar, extender y utilizar el software. En este proyecto se han utilizado sus funciones básicas para modelar diagramas en la web junto con su *property panel*, una reciente extensión que permite mostrar y editar diversos detalles del diagrama que se está visualizando. Además, se han **extendido** las características de este panel, aportando más funciones. Para ello, se ha accedido al código fuente del proyecto y se han agregado formularios que permitan a los usuarios para añadir información sobre el *tiempo de ejecución* de tareas y flujos, sobre los *recursos* que usan las tareas y sobre la *probabilidad* que tiene un flujo que sale de un gateway de ser seleccionado. Más adelante se detalla el porqué de estas extensiones.

La última librería utilizada se llama **C3.js** [4] y se utiliza para generar gráficas dinámicas en los navegadores web. Está basada en otra librería llamada D3.js, pero ofrece una API más sencilla de usar y es más liviana. En el proyecto se utiliza para generar gráficas que reflejan el uso de los recursos de un proceso a lo largo del tiempo. Está distribuido bajo la licencia MIT al igual que bpmn-js.

De la misma manera que en el servidor, se ha usado **Webpack** para generar un único fichero JavaScript que organice todas las dependencias y contenga todo el código de esta parte de la aplicación, tanto el código de c3.js y bpmn-js como los documentos HTML y los ficheros Css3.

Para finalizar, se ha utilizado **Figma** [15] para diseñar las vistas de la página web. Figma es un software que permite diseñar de forma sencilla prototipos y diseños web. Te facilita un buen número de funciones y plantillas y además te permite generar una parte del código Css3 necesario para obtener el resultado que se ha creado.

1.4.3. Parser

Como se ha comentado previamente, a lo largo del trabajo nos referiremos como **Parser** al programa que traduce de XML a Maude. Los diagramas BPMN se representan internamente con ficheros XML, pero todos los scripts Maude desarrollados por el tutor del trabajo y sus colaboradores toman como entrada una representación en Maude de los diagramas, la cual está diseñada y elaborada por ellos de forma manual. Es por ello que era estrictamente necesario desarrollar un programa que de forma automática reciba cualquier XML que represente un diagrama BPMN acorde con el estándar y genere su correspondiente representación Maude.

Se ha decidido utilizar Java para esta labor, debido a su naturaleza orientada a objetos y su facilidad para elaborar código robusto y extensible, algo necesario debido a la metodología usada. Además, se utilizan diferentes librerías del lenguaje como **Simple API for XML (SAX)** [31] . Se estudiaron otras opciones como **Document Object Model API for XML processing (DOM)** pero SAX es más recomendable para lectura de documentos y DOM para manipularla. Esto se debe a que el primero funciona en base a eventos que se disparan cuando el parser interno que utiliza la librería comienza a leer elementos, termina de leerlos, etc. Este sistema de eventos hace que sea más sencillo generar código extensible, como detallaremos más adelante.

1.5. Estructura de la memoria

Se ha decidido que la manera de estructurar la memoria sea similar a la manera en la que se desarrolló el proyecto. Para ello, dividiremos la memoria en tres secciones donde cada una referencia a una iteración, para poder explicar en detalle cómo se llevó a cabo cada una. Después, se hablará acerca de las conclusiones obtenidas y sobre los trabajos futuros, finalizando con un anexo que contendrá toda la documentación generada y los manuales de instalación y de uso.

2

Fases del desarrollo

Como se comentó previamente, se ha seguido a una metodología iterativa incremental que se ha dividido en tres iteraciones. A continuación se procede, como parte principal de esta memoria, a detallar todo lo propuesto y realizado en cada una de ellas.

2.1. Iteración cero

Esta iteración se corresponde cronológicamente con el inicio del proyecto. Fue entonces cuando se decidió que era mejor realizar una serie de procesos iniciales antes de comenzar a escribir código y adentrarnos en el proceso de desarrollo.

2.1.1. Estudio del estado del arte

El **estado del arte** se podría definir como el conjunto de conocimiento acumulado que existe sobre un área específica [28]. En lo referente a nuestro proyecto, este proceso se centró en buscar si existían en el mercado herramientas con funciones similares a las que nosotros queríamos desarrollar, es decir, que realizarán análisis formales sobre diagramas BPMN.

Tras realizar una investigación, se encontró un proyecto llamado **bProVe** [29] desarrollado por *Processes and Service Lab*. BProVe es una herramienta para realizar análisis sobre procesos BPMN que cumplan con el estándar 2.0 basándose en operaciones formales sobre una semántica definida por ellos. Además, proporcionan diferentes maneras de utilizar su software: como aplicación web, como un programa de escritorio, o como un plug in de eclipse. La arquitectura varía según la forma en la que se use la herramienta, pero a alto nivel está basada en tres pilares: *Modelling Enviroment*, *BProVe WebService* y *BProVe Framework*. Modelling Enviroment es la capa de cliente, en la cual los usuarios crean o cargan diagramas BPMN, lo *parsean* utilizando un botón y posteriormente seleccionan qué tipo de análisis quieren realizar. Una vez el usuario ha seleccionado el análisis la información pasa a BProVe WebSerivce, este componente recibe

la información vía HTTP y se encarga de realizarle una serie de transformaciones para que dé comienzo el análisis por parte de BProVe Framework, el cual los realiza utilizando Maude para la semántica de BPMN, LTL Maude para ejecutar premisas y MultiVesta para realizar model checking estadístico.

Consideramos que la herramienta es fácil de usar y proporciona buena documentación. Sin embargo, pensamos que nuestra aportación tendría valor añadido si podíamos ser capaces de crear una interfaz también intuitiva y a la vez moderna que realizará además análisis temporales, de paralelismo y de recursos, ya que esta información se queda fuera del alcance del estándar BPMN 2.0 y por tanto no está contemplado por ellos.

2.1.2. Estudio de los análisis formales

Una vez se estudiaron diferentes sistemas similares, se comenzó a realizar un estudio completo de cada una de las **publicaciones o *papers*** de congreso y revista que el equipo del tutor de este TFG había realizado para poder tener un grado de comprensión más alto de lo que es lo qué se iba a implementar y cómo. Concretamente, el estudio se centró en los siguientes papers:

1. *Verifying Timed BPMN Processes Using Maude.* [11]
2. *Computing the Parallelism Degree of Timed BPMN Processes.*[9]
3. *Stochastic analysis of BPMN with time in rewriting logic.* [10]
4. *Analysis of the Runtime Resource Provisioning of BPMN Processes Using Maude* [8]
5. *Analysis of Resource Allocation of BPMN Processes* [7]
6. *A Rewriting Logic Approach to Resource Allocation Analysis in Business Process Models* [12]

Las dos primeras publicaciones están muy relacionados entre sí ya que ambas utilizan el mismo subconjunto de construcciones BPMN, con la distinción de que se centran en explicar análisis diferentes. Por ello, en primer lugar se explican estas dos para posteriormente abarcar el resto de trabajos, que fueron publicados años más tarde, utilizando un subconjunto más grande y complejo de construcciones y realizando análisis formales centrados principalmente

en los recursos. Como se detalla en las fases del trabajo, los análisis descritos en estos dos primeros estudios fueron los desarrollados en la primera iteración, mientras que los análisis descritos en el resto de ellos se implementaron en la segunda.

2.1.2.1. Papers de la primera iteración

Estos proyectos surgen de la importancia de modelar correctamente procesos de negocio y de cómo el realizar análisis formales sobre los mismos pueden llevar a su correcta y efectiva ejecución. En estos papers, Francisco Durán y Gwen Salaün tratan de proponer técnicas para poder responder a preguntas como [11]: "¿está mi modelo representando precisamente lo que yo quiero que modele?, ¿está libre de errores?, ¿se están conservando ciertas propiedades de interés?, ¿cuál es el grado de paralelismo y el mínimo tiempo de ejecución?, etc."

Todas estas cuestiones son importantes, pero es una tarea complicada que puede incluso llegar a ser *indecidible* si se considera la expresividad completa de BPMN, es decir, si se consideran aspectos como los comportamientos cíclicos, los datos externos o el tiempo. Es por esto, que en estos trabajos se considera un **subconjunto** del estándar 2.0 de BPMN, el cual abarca *tareas*, *eventos de inicio y fin*, *gateways* y *flujos*. Además de propiedades temporales sobre tareas y flujos.

Sobre estos elementos, se proporcionan técnicas de análisis que son independientes del diagrama y que no requieren ninguna interacción por parte del usuario. Concretamente, se explica cómo calcular el **tiempo mínimo, máximo y medio de ejecución** de un diagrama. Así como el **cálculo del grado de paralelismo**, es decir, el número máximo de tareas que se ejecutan a la vez en un mismo instante de tiempo. Además, se puede comprobar si un diagrama cumple la propiedad llamada **liveness**, que verifica si este está fuera de deadlocks. Por último, se puede aplicar **model checking** de propiedades basadas en LTL para verificaciones temporales sobre el diagrama. Estas propiedades se basan en patrones que tienen una sintaxis bien definida y es necesario que sean especificados por parte del usuario o desarrollador, ya que varían con cada diagrama.

Para conseguir estos objetivos, se llevaron a cabo tres tareas principales: el desarrollo de una representación Maude para cada diagrama BPMN *(sintaxis)*, la elaboración de un conjunto de reglas que modele el comportamiento de cada elemento BPMN *(semántica)* y la creación de una forma de ejecutar estos procesos *(simulación)*. Antes de explicar cómo se realizaron estos

componentes, es importante mencionar a alto nivel cómo funciona cada uno de los elementos de BPMN considerados en este trabajo, estos son:

- Eventos (Comienzo y fin): Representan, respectivamente, el inicio y el final de un proceso.

- Tarea: Representa una actividad atómica, que tiene exactamente un flujo de entrada y un flujo de salida. Adicionalmente y fuera del estándar, se considera que tienen un tiempo finito a partir del cual se completarán.

- Puerta: Permiten controlar la divergencia y convergencia de los flujos. En BPMN existen distintos tipos de puertas: exclusiva, inclusiva, paralela, basada en eventos y compleja. Sin embargo solo se consideran las tres primeras, ya que las otras dos se centran en manejar eventos de sincronización centrados principalmente en aspectos relacionados con los datos, cosa que no se tiene en cuenta.

  - Puerta exclusiva: Solo uno de sus flujos es activado.
  - Puerta inclusiva: Uno o más de sus flujos pueden ser activados.
  - Puerta paralela: Todos sus flujos son activados.

  A lo largo del trabajo, se denominará como *split* aquella puerta de carácter divergente, es decir, aquella que tenga un flujo de entrada y varios de salida. Respectivamente, se denominará como *merge*, aquella con carácter convergente, que une varios flujos de entrada en uno de salida.

- Flujos: Permite representar que dos nodos se ejecutan uno después del otro. Al igual que las tareas, también tienen un tiempo finito para su realización, de manera que se puedan modelar retrasos que pueden existir desde que termina una tarea hasta que se pueda empezar otra.

La figura 1 representa gráficamente estos elementos acorde al estándar. Adicionalmente, aparece un punto que se utiliza para mostrar el punto en el que se encuentra la ejecución del diagrama.

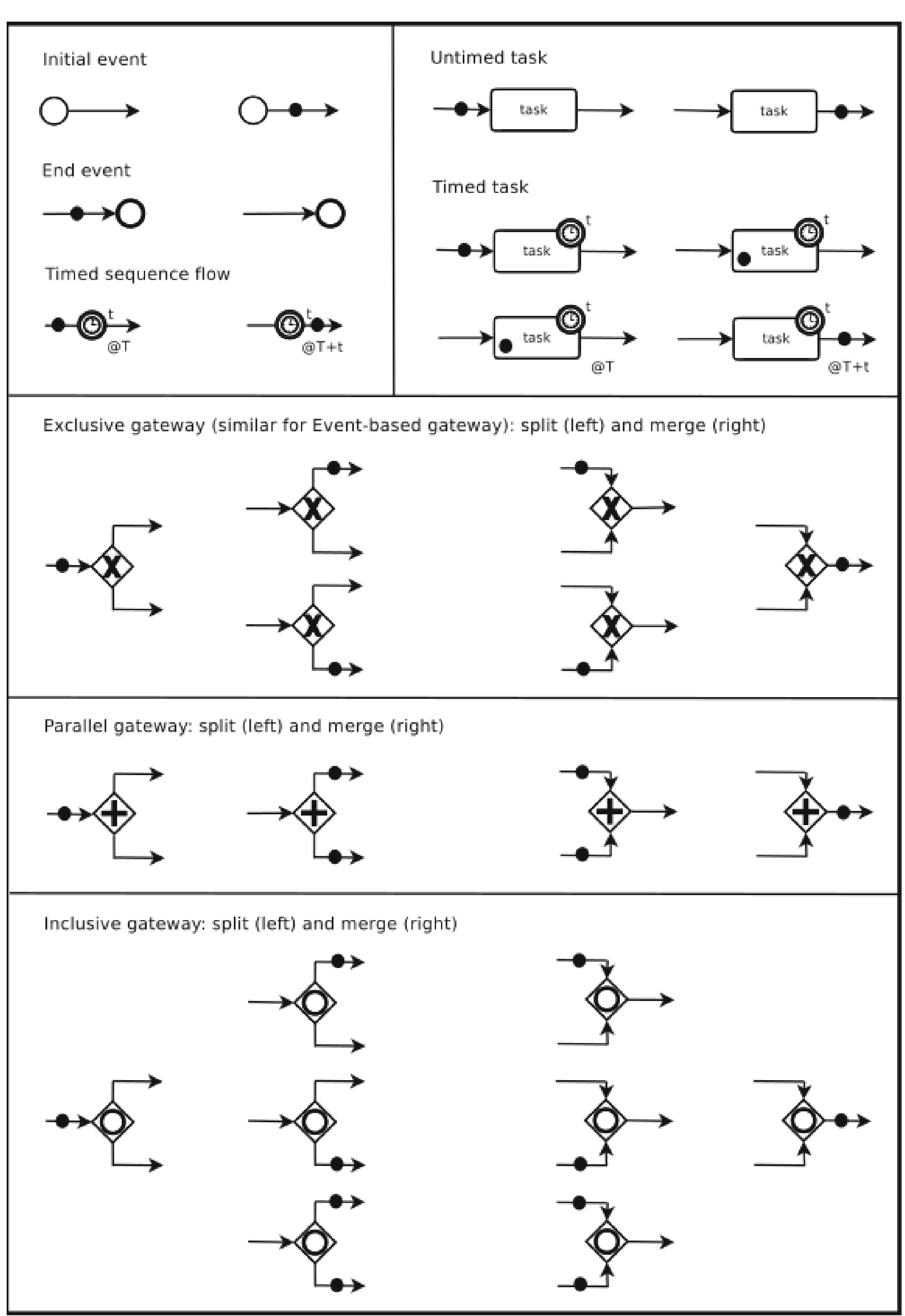


Figura 1: Elementos considerados en la primera iteración. Fuente: [11]

2.1.2.1.1. Notación Maude (sintaxis)

El primer elemento necesario para ejecutar los análisis formales en Maude sobre los diagramas BPMN es tener una manera de representarlos en el lenguaje, es decir, crear una *notación Maude* para diagramas BPMN.

Esta notación es diferente para cada proceso y está basada en dos entidades principales: un conjunto de nodos y un conjunto de flujos. Un flujo está representado por un término del siguiente tipo: *flow(id, t)* donde 'id' es el identificador del flujo y 't' es su tiempo de duración. Por su parte, los nodos pueden ser del tipo: evento, tarea y puerta. Atendiendo a cada posible subtipo de evento y puerta posible. Los nodos de tipo evento de comienzo (final resp.), tienen un identificador propio y otro que lo enlazan con el flujo de salida (entrada resp.). Los de tipo tarea tienen un identificador propio y otros dos que enlazan tanto con el flujo de entrada como con el de salida, además de un campo para especificar el nombre de la tarea y otro para representar su tiempo de duración. Por último, los nodos de tipo puerta tienen un campo haciendo referencia a su tipo en cuestión (exclusiva, inclusiva o paralela) y a su vez se dividen en dos clases según a como modelan el flujo: split o merge.

En la figura 2 se puede ver un sencillo diagrama BPMN junto con un extracto de su notación Maude, figura 3, la cual ha sido generada por el Parser creado como parte de este TFG.

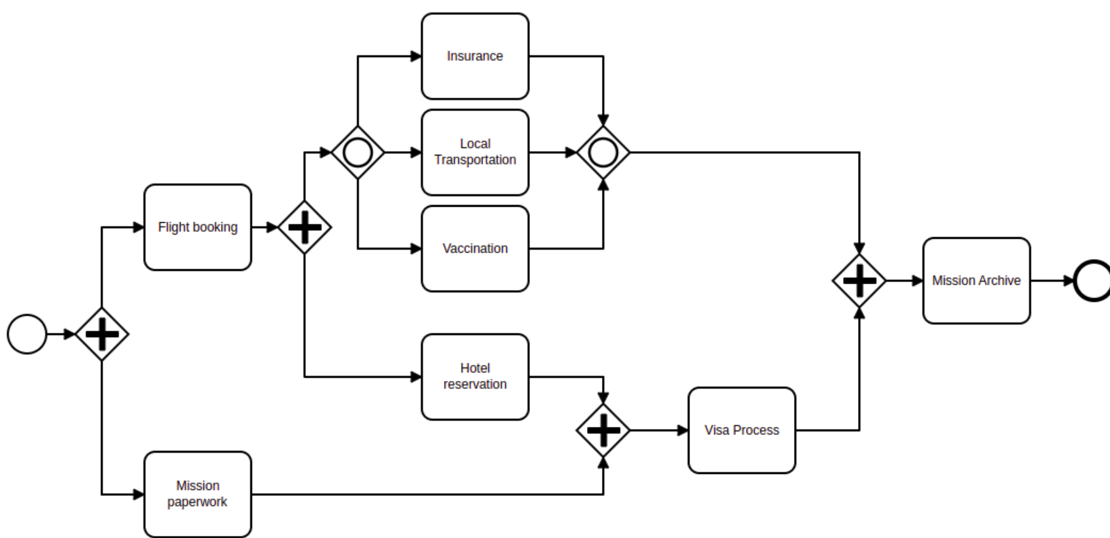


Figura 2: Diagrama de ejemplo de la primera iteración.

```
mod BPMN-EX is
    pr BPMN-SEM .
    ops StartEvent1 Activity10an6ky Activity0fv6xup Activity0nqytk0 Gateway1ni69fl Activity0fgwmex Activity19xtuno Activity0mtneg2 Gateway08fr1n6
    Activity0ys72vt Activity14311m8 Event0yfe4a1 Gateway13ieug4 Gateway0nzgyf0 Gateway0nbdn6b Gateway0bfwbzd : -> NId .
    ops Flow15n4v5t Flow1uifcvi Flow04adi4c Flow1yd5ajb Flow0bijnw8 Flow1n9vrqp Flow1oejj9o Flow1ke9zrp Flow1idfq6c Flow0rig596 Flow0vdf8wy
    Flow1p0kegr Flow1uvbejs Flow08kl2u4 Flow0hzkp4y Flow0yatx7c Flow1ncqs1y Flow0s3usbh Flow1dxiidz : -> FId .
    op fls : -> Set{Flow} .
    op nds : -> Set{Node} .
    eq init = token(Flow15n4v5t,0) .
    eq fls
        = (
            flow(Flow15n4v5t,0),
            flow(Flow1uifcvi,0),
            flow(Flow1yd5ajb,0),

            ...

            flow(Flow0s3usbh,0),
            flow(Flow1dxiidz,0)
          ) .
    eq nds
        = (
            start(StartEvent1,Flow15n4v5t),
            task(Activity10an6ky,"Flight booking",Flow1uifcvi,Flow1yd5ajb,3),
            split(Gateway1ni69fl,inclusive,Flow0bijnw8,(Flow1oejj9o,Flow1ke9zrp,Flow1idfq6c)),

            ...

            merge(Gateway08fr1n6,inclusive,(Flow0rig596,Flow0vdf8wy,Flow1p0kegr),Flow1uvbejs),
            end(Event0yfe4a1,Flow1dxiidz)
          ) .
endm
```

Figura 3: Notación Maude del diagrama de ejemplo.

Las primeras líneas de la figura 3 definen el conjunto de nodos (nds) y flujos (fls) y sus respectivos identificadores (NId y FId). Posteriormente, se define el flujo inicial (enlazado al nodo de comienzo) y el resto de nodos y flujos. Por ejemplo, se puede ver la tarea con nombre “Flight Booking” del diagrama definida dentro del conjunto de nodos nds de la siguiente manera: *task(Activity10an6ky,”Flight booking”,Flow1uifcvi,Flow1yd5ajb,3)*, donde:

- Activity10an6ky: Identificador del flujo, definido en la tercera línea.
- Flight booking: Nombre de la tarea, el cual puede verse en el diagrama.
- Flow1uifcvi y Flow1yd5ajb: Respectivamente el flujo de entrada y de salida. Ambos definidos en el conjunto de flujos fls.
- 3: Tiempo de duración

2.1.2.1.2. Conjunto de reglas (semántica)

En Maude la semántica de BPMN se modela utilizando **tokens** como eje central. Un token almacena dos campos: el id del nodo que referencia (evento, tarea, puerta o flujo) y el tiempo que le queda para terminar de consumirse.

Un *evento de comienzo* puede ser consumido en cualquier momento, lo cual generará un token referenciando a su nodo de salida, comenzando el proceso. Respectivamente, el proceso

terminará en el momento en el que un token que sea entrada de un evento final sea consumido. El **tiempo** se modela en este proyecto como una duración discreta que afecta a tareas y flujos, de forma que si un token asociado a un flujo cuenta con un tiempo $d > 0$, entonces en d unidades de tiempo este token se consumirá y se generará otro referenciando al nodo al que apuntara el flujo. De forma similar ocurre con una *tarea* y el *flujo* que salga de ella. La semántica de las *puertas* es diferente según el tipo y se explicará con más detalle una vez se vea cómo se expresan las reglas en Maude.

Como se viene diciendo, Maude es el lenguaje de programación escogido para modelar la semántica de estos elementos. Además, se utiliza **Real-Time Maude**, una extensión del mismo que soporta la especificación formal y el análisis de sistemas en tiempo real. Ambos se basan en **lógica de reescritura** [33], una extensión de la lógica ecuacional habitual que fue pensada para la especificación de sistemas concurrentes y no deterministas de tiempo real. Gracias a ella, se pueden representar sistemas distribuidos como un conjunto de estados y de reglas, que pueden ser aplicadas sobre ellos en base a ciertas condiciones. Estas reglas pueden modificar los estados, permitiendo definir formalmente todas estas computaciones concurrentes y no deterministas de una forma legible. Tienen la siguiente forma: *crl [L] : t =>t' if C*. Donde L es el nombre de la regla, t es el estado de inicio, t' el estado final y C la condición.

En Maude, dichos estados representan *estados del sistema concurrente* que se está modelando. Estos sistemas están formados por un conjunto de **objetos y mensajes**. Los sistemas orientados a objetos son especificados en módulos donde se definen *clases y subclases*, con la siguiente sintaxis de declaración: $class\ C \mid a_1 : S_1, ..., a_n : S_n$. Donde C es el nombre de la clase, a_i son los identificadores de los atributos y S_i sus tipos. Posteriormente, los *objetos* se crean de la siguiente manera: $<O : C \mid a_1 : v_1, ..., a_n : v_n>$. Donde O es el nombre del objeto y v_i es el valor del atributo a_i.

De esta forma, se puede representar con detalle una transición en un sistema orientado a objetos, como se muestra en la figura 4.

$$
\begin{array}{l}
\textbf{crl}\ [r]\ :\\
\quad < O_1 : C_1 \mid atts_1 > \ \ldots\ < O_n : C_n \mid atts_n >\\
\quad M_1\ \ldots\ M_m\\
\quad \Rightarrow\\
\quad < O_{i_1} : C'_{i_1} \mid atts'_{i_1} > \ \ldots\ < O_{i_k} : C'_{i_k} \mid atts'_{i_k} >\\
\quad < Q_1 : C''_1 \mid atts''_1 > \ \ldots\ < Q_p : C''_p \mid atts''_p >\\
\quad M'_1\ \ldots\ M'_q\\
\quad \textbf{if}\ Cond\ .
\end{array}
$$

Figura 4: Regla de un sistema orientado a objetos con mensajes en Maude. Fuente: [11]

Para simular el transcurso del tiempo es necesario modelar el incremento del tiempo global y el decremento del tiempo restante de cada token del sistema, todo de forma proporcional. Para hacerlo de una forma más eficiente, se definen dos reglas: **delta y mte (maximal time elapsed)**. Delta define el efecto del paso del tiempo para cada elemento del sistema mientras que mte devuelve el máximo número de unidades de tiempo que pueden pasar sin que ninguna acción sea realizada. Por tanto, la estrategia es utilizar mte para determinar cuántas unidades de tiempo se puede "avanzar" el sistema de una sola vez, y aplicarlo con delta a todos los tokens activos. Evitando realizar una transformación sobre cada elemento por unidad de tiempo.

2.1.2.1.3. Simulación

Para la simulación se utilizan este tipo de reglas sobre dos objetos llamados *Process y Simulation*. El objeto Process, almacena la información del diagrama a analizar mediante el conjunto de nodos y flujos definidos previamente (nds y fls) y no cambia durante la ejecución. Mientras que el objeto Simulation guarda dinámicamente el conjunto de tokens que se van procesando junto con el tiempo global del sistema. Sobre él se ejecutarán cada una de las reglas de reescritura que especifican la semántica de BPMN, permitiendo realizar los análisis formales. En la figura 5 se muestra la definición de estos objetos y la figura 6 muestra un ejemplo de cómo se podrían instanciar, todo según la sintaxis Maude expresada previamente.

```
class Process | nodes : Set {Node}, flows : Set {Flow} .
class Simulation | tokens : Set {Token}, gtime : Time .
```

Figura 5: Definición de los objetos Process y Simulation.

```
eq system
    = < p : Process | nodes : nds, flows: fls >
      < s : Simulation | tokens : token (initial, 0), gtime:0 > .
```

Figura 6: Declaración de los bjetos Process y Simulation.

Para dar semántica a los diagramas, se debe definir al menos una regla por cada tipo de nodo. El **evento de comienzo**, es el encargado de iniciar el flujo del proceso BPMN y como se comentó previamente, este tipo de nodo almacena un identificador y un puntero a su flujo de salida. Por ende, cuando en el objeto Simulation aparezca un token con tiempo cero apuntando a un nodo de comienzo, este se debe consumir para crear otro que apunte a su flujo de salida. Maude permite expresar este tipo de transiciones de forma sencilla, siguiendo el esquema de reglas de escritura que hemos visto, lo cual se refleja en la definición de la regla **startProc**, en la figura 7.

```
---- Initiation of the process
rl [startProc] :
    < PId : Process |
        nodes : (start(NId, FId), Nodes),
        flows : (flow(FId, T), Flows),
        Atts >
    < SId : Simulation |
        tokens : (token(NId, 0), Tks),
        Atts1 >
  =>
    < PId : Process |
        nodes : (start(NId, FId), Nodes),
        flows : (flow(FId, T), Flows),
        Atts >
    < SId : Simulation |
        tokens : (token(FId, T), Tks),
        Atts1 >
  [print "startProc " PId] .
```

Figura 7: Regla startProc. Modela el comportamiento del evento de comienzo

La parte izquierda de la regla, expresa que esta se accionará para cualquier proceso que cumplan las siguientes condiciones:

1. Tener un nodo de comienzo start(NId, FId), donde recordemos que **NId** es su identificador y **FId** es el identificador del flujo de salida al que apunta, que también estará definido como flow(**FId**,T), siendo T su tiempo asociado.

2. Tener un token en el objeto Simulation con tiempo cero apuntando al nodo de comienzo: token(**NId**, 0)

El consecuente, indica lo que le ocurrirá al proceso tras la ejecución, en este caso, seguirá intacto en su totalidad, excepto que se habrá consumido el token que referenciaba al nodo start y se ha creado uno que apunta a su flujo de salida, de identificador **FId**.

Por su parte, para que se ejecute la regla del **evento de fin**, debe de haber un token con tiempo restante cero en el objeto Simulation que apunte al flujo de entrada de un nodo final. Esta regla consumirá dicho token sin generar ninguno más, haciendo que el proceso eventualmente se detenga.

Las **tareas** se representan con dos reglas, debido al aspecto temporal que las caracteriza. Hay una regla para iniciar una tarea y otra para completarla, llamadas respectivamente **initTask** (figura 8) y **execTask** (figura 9). La primera se activa cuando en el objeto Simulation hay un token de tiempo cero apuntando al flujo de entrada de una tarea, lo que consume este token y crea uno que hace referencia a la tarea. La segunda se activa cuando el token que apunta a la tarea llega a tener un tiempo asociado de cero unidades, entonces, es consumido y se crea otro que apunta al flujo de salida de la tarea.

```
---- Initiation of a task
rl [initTask] :
    < PId : Process |
        nodes : (task(NId, TaskName, FId1, FId2, T), Nodes),
        flows : Flows,
        Atts >
    < SId : Simulation |
        tokens : (token(FId1, 0), Tks),
        Atts1 >
  =>
    < PId : Process |
        nodes : (task(NId, TaskName, FId1, FId2, T), Nodes),
        flows : Flows,
        Atts >
    < SId : Simulation |
        tokens : (token(NId, T), Tks),
        Atts1 >
  [print "taskInit " TaskName] .
```

Figura 8: Regla initTask. Modela el comportamiento de las tareas (1 / 2)

```
---- Execution / completion of a task
rl [execTask] :
    < PId : Process |
        nodes : (task(NId, TaskName, FId1, FId2, T), Nodes),
        flows : (flow(FId2, T2), Flows),
        Atts >
    < SId : Simulation |
        tokens : (token(NId, 0), Tks),
        Atts1 >
  =>
    < PId : Process |
        nodes : (task(NId, TaskName, FId1, FId2, T), Nodes),
        flows : (flow(FId2, T2), Flows),
        Atts >
    < SId : Simulation |
        tokens : (token(FId2, T2), Tks),
        Atts1 >
  [print "TASK COMPLETION ===>>> " TaskName] .
```

Figura 9: Regla execTask. Modela el comportamiento de las tareas (2 / 2)

La **puerta exclusiva** tiene la semántica más sencilla de su grupo. Su funcionamiento consiste en escoger de forma no determinista uno de sus flujos de salida en el split, para posteriormente recogerlo en el merge, teniendo definida una regla para cada uno. En la regla del *split* (figura 10) se puede ver como se expresa el no determinismo, ya que se definen el conjunto de flujos de salida como *(FId2, FIds)*, patrón que puede ser activado por cualquiera de ellos.

```
---- Split gateway - exclusive
rl [splitGatewayExclusive] :
    < PId : Process |
        nodes : (split(NId, exclusive, FId1, (FId2, FIds)), Nodes),
        flows : (flow(FId2, T), Flows),
        Atts >
    < SId : Simulation |
        tokens : (token(FId1, 0), Tks),
        Atts1 >
  =>
    < PId : Process |
        nodes : (split(NId, exclusive, FId1, (FId2, FIds)), Nodes),
        flows : (flow(FId2, T), Flows),
        Atts >
    < SId : Simulation |
        tokens : (token(FId2, T), Tks),
        Atts1 >
  [print "splitExclusive " NId] .
```

Figura 10: Regla splitGatewayExclusive. Modela el comportamiento del split exclusivo.

El *merge* de esta puerta es sencillo, debido a que como solo se genera un único flujo, se crea una regla que capture el momento en el que objeto Simulation haya un token con tiempo cero en uno de los flujos de entrada del merge, haciendo que se consuma y generando un token que apunte a su flujo de salida.

Por su parte, la regla del split **puerta paralela** genera un token en cada uno de sus flujos

de sus salidas en el split. Sin embargo, hay que prestar más atención al merge, y es que para que el merge pueda ser activado en una puerta paralela se debe esperar a que todos sus flujos entrada lleguen a la puerta, quedando bloqueados los primeros en llegar. Para ello, en la regla **mergeGatewayParallel** (figura 11) se especifica que cada flujo de entrada del merge debe estar referenciado por un token en el objeto Simulation con tiempo cero, lo cual se logra con el uso de la condición de la regla: *if AllTokensParallel*, donde AllTokensParallel es una función auxiliar que de forma recursiva realiza esta comprobación. Destacar que en Maude, la comprobación de la condición se realiza cada vez que el estado del objeto Simulation cumpla el patrón declarado en la parte izquierda de la regla. En este caso, ocurre cada vez que llegue al merge uno de los flujos de salida de la regla paralela, ejecutándose la regla en el momento en el que llegue el último merge. Posteriormente, se eliminan todos los tokens usando otra función: *removeAllTokensParallel* y se crea uno con el flujo de salida.

```
---- Merge gateway - exclusive
rl [mergeGatewayExclusive] :
   < PId : Process |
       nodes : (merge(NId, exclusive, (FId1, FIds), FId2), Nodes),
       flows : (flow(FId2, T), Flows),
       Atts >
   < SId : Simulation |
       tokens : (token(FId1, 0), Tks),
       Atts1 >
  =>
   < PId : Process |
       nodes : (merge(NId, exclusive, (FId1, FIds), FId2), Nodes),
       flows : (flow(FId2, T), Flows),
       Atts >
   < SId : Simulation |
       tokens : (token(FId2, T), Tks),
       Atts1 >
  [print "mergeExclusive " NId] .
```

Figura 11: Regla mergeGatewayParallel. Modela el comportamiento del merge paralelo.

La última puerta contemplada, y también más complicada de definir, es la **puerta inclusiva**. Esta escoge de forma no determinista tanto los flujos como el número de flujos que se escogen. Su split se define de forma sencilla gracias a la gran capacidad expresiva del lenguaje. Sin embargo, en el merge se debe definir el bloqueo de los primeros nodos a la espera del resto, cuando no se saben cuántos ni cuáles son los que deben llegar hasta tiempo de ejecución. La única forma viable de resolver este reto es, cada vez que se detecte que un nodo ha llegado al merge, recorrer el flujo hacia atrás en busca de ramas activas que provengan del split, lo cual se realiza con una función recursiva que se usa a modo de condición al igual que en la regla paralela.

2.1.2.1.4. Análisis formales

Con la notación Maude (sintaxis), el conjunto de reglas (semántica) y los objetos Process y Simulation (Simulation), se pueden realizar análisis formales sobre diagramas BPMN en el lenguaje Maude. En este paper se explican los análisis temporales, de grado de paralelismo, de la propiedad llamada liveness y model checking.

La **verificación de propiedades temporales** se consigue aplicando frameworks y librerías propias del lenguaje Maude sobre los componentes semánticos descritos. Dado un diagrama expresado mediante los objetos Process y Simulation, la función *execTime*, explora todas las posibles soluciones del diagrama, es decir, todos aquellos flujos en los que se pueda alcanzar un estado final, y computa las métricas temporales contempladas: **tiempo de ejecución mínimo, máximo y medio**. La exploración se realiza utilizando una *búsqueda en amplitud* sobre el espacio de estados, limitando la profundidad o el tiempo de ejecución para evitar quedar atrapados indefinidamente en un posible bucle infinito creado al diseñar el diagrama.

Para calcular el **grado de paralelismo**, se recorren no solo todos los estados finales, sino todos los estados alcanzables para poder encontrar aquel que tenga un mayor número de tokens, lo cual se corresponderá con el número de nodos que se están ejecutando en paralelo en un momento dado.

La verificación del **liveness** sobre un proceso se realiza mediante una simulación en la que se recorren todas las posibles ejecuciones y se comprueba si un proceso se queda o no atascado en un *deadlock*. En este contexto, se entiende por deadlock todo estado no terminal en el que no se pueda transitar a otro estado. Una construcción de BPMN da problemas a la hora de realizar este análisis: los *bucles infinitos*. Un bucle infinito puede no contener un deadlock, sin embargo, su ejecución no termina y por tanto esta simulación tampoco.

Para solventar este problema y asegurar que el análisis acaba, se **limita** la profundidad de búsqueda. Desgraciadamente, esto hace que este análisis sea incompleto, ya que en los casos en los que el análisis termina por esta razón no se puede afirmar que el proceso está libre de deadlocks, debido a que existe la pequeña posibilidad de que no se hayan podido comprobar todas las ejecuciones posibles por falta de memoria.

Además de la simulación y la definición de funciones a resolver en el contexto de los modelos de procesos, Maude ofrece otras herramientas, como por ejemplo su model checker. En este caso utilizamos su **model checker de estados explícitos**, que nos permite verificar propiedades de *lógica temporal lineal (LTL)* sobre la ejecución de un diagrama BPMN. Este tipo de análisis permite comprobar, entre otras cosas, que el diagrama está libre de *deadlocks*, que una cierta tarea siempre se ejecuta o que una cierta tarea se ejecuta siempre que se haya completado otra. Todas estas propiedades LTL dependen del diagrama en cuestión y deben ser especificadas por el usuario. Sin embargo, como parte del proyecto se especificarán en la propia aplicación unas ayudas indicando los patrones que se pueden usar para escribir estas propiedades.

Por ejemplo, sobre el diagrama de la figura 2 se podría escribir la siguiente propiedad LTL para comprobar si la tarea con nombre "Visa process" es ejecutada en algún momento siempre que antes se haya ejecutado la tarea "Flight Booking", Figura 12. El resultado que devuelve Maude sobre esta proposición es true, lo cual tiene sentido ya que el diagrama es sencillo y se puede comprobar visualmente. En caso de que la propiedad no se cumpla el model checker devolverá un contraejemplo, una traza de la ejecución del proceso que no satisface la propiedad. Con esta traza, un usuario experimentado puede identificar el problema, o al menos la razón del contraejemplo, para poder corregirlo.

```
reduce modelCheck ( InitSystem, [] (Flight Booking -> <> Visa Process )) .
```

Figura 12: Ejemplo de propiedad LTL sobre el model checker de Maude.

2.1.2.2. Papers de la segunda iteración

Continuamos ahora explicando los tres últimos trabajos en los que se basan los análisis formales que se utilizan en este trabajo.

Este conjunto de estudios gira en torno a los **recursos**. Por recursos, entendemos cualquier persona o cosa que puedan representar valor y que sea necesaria para la ejecución de ciertas actividades dentro de un proceso de negocio. Por ejemplo, un camión y su conductor en una empresa de transporte. Las compañías están constantemente tratando de encontrar nuevas formas de aumentar sus beneficios mediante un mejor uso de los recursos que emplean. Por ello, estos estudios proporcionan análisis que tratan de apoyar a la persona que se encarga de diseñar los procesos de negocio (mediante diagramas BPMN, en nuestro caso) que interactúen con cualquier tipo de recurso, permitiendo que evalúe el correcto uso de los mismos y el rendimiento del diseño, además de permitir comparar entre los resultados obtenidos con distintos números de instancias para cada recurso y con distintas técnicas de reaprovisionamiento.

Los análisis siguen el mismo *modus operandi* que en los ya comentados, es decir, primero se crea la notación Maude que representa al diagrama BPMN y después se realiza el análisis usando diferentes herramientas y frameworks propios del lenguaje, apoyándose entre otras cosas en simulaciones usando los objetos Process y Simulation. No obstante, estos papers realizan análisis más complejos ya que utilizan un subconjunto más grande de construcciones BPMN, además de ciertas propiedades adicionales que están fuera del estándar: **expresiones estocásticas para el tiempo de ejecución de tareas y flujos, probabilidades en los splits de los gateways y los propios recursos**. La idea es realizar múltiples simulaciones concurrentes del proceso negocio que compitan entre sí por una serie de *recursos compartidos*, de manera que se pueda analizar la evolución del uso de los mismos sin necesidad de tener que realizar pruebas en sistemas con recursos reales.

2.1.2.2.1. Subconjunto BPMN

Como se ha mencionado, se considera un subconjunto mayor de BPMN que permite expresar un mayor número de casos de uso y situaciones más reales. Se consideran diagramas colaborativos, estos incluyen la definición de **pools y lanes**, que permiten separar el flujo del diagrama en partes que normalmente son realizadas por roles o recursos diferenciados.

Las actividades y los flujos anteriormente tenían asociado un tiempo discreto para su rea-

lización, ahora ese tiempo también puede venir dado por una **expresión estocástica** que permite añadir más indeterminismo a las simulaciones. Dichas expresiones estocásticas vienen representadas por distribuciones de probabilidad, concretamente dos: *gaussianas y uniformes*. Además, las tareas podrán enviar **mensajes**, que son una forma de representar comunicaciones entre nodos, pudiendo también activar **eventos de mensajes**, otro elemento nuevo que también se contempla en esta versión y se explicará a continuación. En los splits de las puertas exclusivas e inclusivas se modelan condiciones basadas en datos permitiendo que el usuario pueda especificar la **probabilidad** que tiene cada rama de ser escogida. En las puertas exclusivas solo se puede escoger una rama, por lo que para que un diagrama sea considerado válido la suma de las probabilidades de cada una debe ser igual a uno. Por su parte, en la inclusivas cada rama es independiente y las probabilidades solo deben tener un valor entre cero y uno inclusive. Se consideran también los **eventos de mensajes**, los de **temporizadores** y las **puertas basadas en eventos**. Todos están relacionados ya que el split de una puerta basada en eventos tendrá en cada una de sus ramas un evento asociado, siendo dicha rama seleccionada cuando su evento es activado. Un evento de mensaje será activado si recibe un mensaje y un evento de temporizador cuando su tiempo llegue a cero. Una **tarea** puede tener asociada un número de **recursos** de los cuales necesita para ser completada, quedando bloqueada si el número de instancias del recurso que necesitas no está disponible en ese momento. En la figura 13 se puede ver una representación de los elementos soportados.

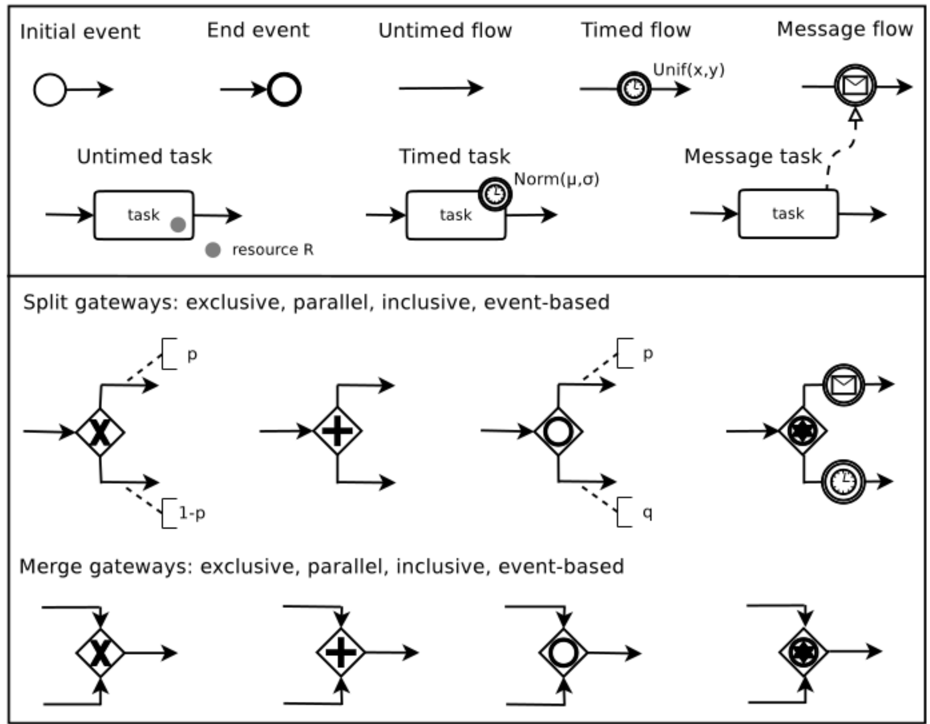


Figura 13: Construcciones BPMN soportadas.

La forma en la que se utiliza Maude es muy similar a las anteriores salvo por ciertas modificaciones adicionales. Por esto no es necesario volver a hablar sobre lógica de reescritura ni sobre cómo se aplica a Maude, así como el uso de los objetos y las simulaciones concurrentes en el lenguaje, pudiendo pasar directamente a explicar la nueva notación Maude (sintaxis), el conjunto de reglas usado (semántica), las simulaciones y los análisis.

2.1.2.2.2. Notación Maude

Las tareas ahora guardan la información de los recursos que necesitan para poder ser llevadas a cabo. Dicha información se almacena con el nombre de cada recurso y el literal "empty" en el caso de que no necesite ninguno. Lo mismo ocurre también con los mensajes que estas pueden enviar. En las tareas y en los flujos de secuencia el tiempo puede ser un valor numérico (caso determinista) o una expresión estocástica dada como una de las dos distribuciones consideradas. Por último, los eventos de mensajes y los de temporizadores se representan también mediante flujos que almacenan su información.

En la figura 14, se muestra el diagrama de ejemplo que se utilizará para explicar los resultados de estos análisis adicionales. Justo después se muestra, en formato reducido por legibilidad,

su respectiva notación Maude usada en esta nueva iteración, figura 15. En ella se puede ver la información referente a los nuevos elementos. Se pueden flujos simples y flujos que referencian a eventos de mensajes y timers. En cuanto a los nodos, vemos tareas con tiempos discretos y estocásticos (tarea “sign in”), con información de recursos y mensajes. Finalmente, en el split de tipo exclusivo podemos ver como se definen probabilidades para cada rama, sumando un total de uno.

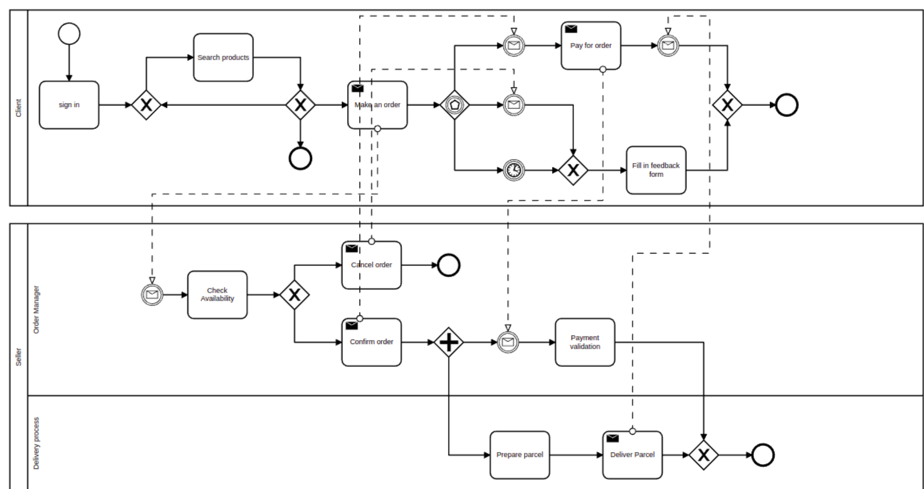


Figura 14: Diagrama de ejemplo de la segunda iteración.

```
mod BPMN-EX is
    pr APMAUDE .
    ops initial Activity1jmbxjc Gateway0zfeidj Gateway11fg87n Activity02lykja Event0nxtw05 Event1ariler Activity00gvz2o Gateway0uiqpvg Gateway1qih4pl
    Gateway1wwmvfb Activity0ukzxks Activity1hpa373 Activity07fhp5a Gateway1vabvf6 Event1x7tu6b Gateway0vfez8x Activity0xmi0g9 Activity0tg691s Gateway08lntin
    Event1lyfevy Activity0o8yo54 Activity046hgah Activity1azt1v8 : -> NId .
    ops Flow0mgo136 Flow1h1h7yv Flow1qxfkyr Flow0j3qhrh Flow1l2x50r Flow0ko7elk Flow1tqzcnd Flow0s35djz Flow1j46j1t Flow1mq3h5l Flow106q39m Flow0oepogb
    Flow15851a3 Flow02frltg Flow1x1go3b Flow1me5d6k Flow08wljwd Flow1r7p07o Flow030sryu Flow0zfw35g Flow0ypjdjx Flow00e6cby Flow1mwcv8o Flow1jhmz75
    Flow08irusd Flow01vbsni Flow18149oz Flow059w0x6 Flow14fhlgt Flow1kbch30 Flow1qr3q9w Flow0k2paad Flow103808z : -> FId .
    ops mFlow15851a3 timeout mFlow08wljwd mFlow1r7p07o mFlow059w0x6 : -> Id .
    --- MessageFlow id declaration
    ops Flow0t64l8i Flow0181ju5 Flow0tcxp75 Flow091ws5f Flow15e79x6 : -> FId .
    --- MessageFlow mid declaration
    ops mFlow0t64l8i mFlow0181ju5 mFlow0tcxp75 mFlow091ws5f mFlow15e79x6 : -> Id .
    ops drone employee : -> Id .
    op fls : -> Set{Flow} .
    op nds : -> Set{Node} .
    op res : -> Set{Resource} .
    op resources : -> Map{Id,Time} .
    eq fls
        = (
            flow(Flow0mgo136,0),
            flow(Flow1h1h7yv,0),
            flow(Flow1me5d6k,0,timer(timeout, 60)),

            ...

            flow(Flow08wljwd,0,message(mFlow08wljwd,"")),
            flow(Flow1r7p07o,0,message(mFlow1r7p07o,"")),
            flow(Flow15e79x6,message(mFlow15e79x6,""),Flow0ypjdjx)
          ) .
    eq nds
        = (
            start(initial,Flow0mgo136),
            task(Activity1jmbxjc,"sign in",Flow0mgo136,Flow1h1h7yv,6,(drone),empty),
            merge(Gateway0zfeidj,exclusive,(Flow1h1h7yv,Flow0ko7elk),Flow1qxfkyr),
            split(Gateway11fg87n,exclusive,Flow0j3qhrh,((Flow0ko7elk, 0.6) (Flow1tqzcnd, 0.2) (Flow0s35djz, 0.2))),

            ...

            task(Activity0o8yo54,"Cancel order",Flow0k2paad,Flow1kbch30,4,empty,(mFlow08wljwd)),
            task(Activity046hgah,"Confirm order",Flow1qr3q9w,Flow14fhlgt,12,empty,(mFlow1r7p07o)),
            task(Activity1azt1v8,"Deliver Parcel",Flow01vbsni,Flow1mwcv8o,6,empty,(mFlow15851a3)),
            end(Event1lyfevy,Flow00e6cby)
          ) .
endm
```

Figura 15: Notación Maude del diagrama de ejemplo.

2.1.2.2.3. Semántica y Simulación

La semántica, al igual que en los artículos usados en la primera iteración, se define usando lógica de reescritura sobre el lenguaje Maude mediante un conjunto de reglas que aplican transiciones sobre el estado actual del análisis. Antes se manejaban dos objetos en estas simulaciones:

- Process: Almacena el diagrama a analizar y no cambia durante el análisis.
- Simulation: Almacena el tiempo global del sistema junto con el conjunto de tokens (dupla con el identificador del nodo y el tiempo que le falta para consumirse). Sobre este objeto se aplicaban todas las reescrituras que simulan la evolución del sistema y permiten los análisis.

En este caso, se siguen usando estos objetos pero con una serie de modificaciones y extensiones. El objeto Simulation tiene una serie de atributos adicionales tales como el conjunto de eventos del sistema (mensajes y temporizadores) y el conjunto de recursos, además de almacenar las métricas que se van obteniendo. La representación del objeto es la siguiente (figura 16).

```
< s : Simulation |  tokens : ...,             --- conjunto de tokens
                    gtime : ...,              --- tiempo global
                    resources : ...,          --- conjunto de recursos
                    events : ...,             --- conjunto de eventos
                    process-execs : ....,     --- métrica asociada al tiempo de ejecución
                    sync-times : ...,         --- métrica asociada al tiempo de sincronicazión
                                              --- de las puertas tipo merge
                    task-times : ...,         --- métrica asociada al tiempo que se necesita
                                              --- para ejecutar las tareas (bloqueos por recursos)
                    ... >
```

Figura 16: Objeto Simulation de la segunda iteración.

En estos análisis no se simula un solo proceso a la vez, sino que se realizan al mismo tiempo múltiples simulaciones concurrentes. Por ello, los tokens deben tener un atributo adicional, el **Tid**, que representa el identificador de la ejecución a la que pertenecen. Para ayudar a soportar la concurrencia, los tokens se guardan en un **scheduler**, implementado como una cola de prioridad que ordena según el tiempo restante de los tokens en orden ascendente. Puede darse el caso de que haya tokens con tiempo restante cero que no puedan ser ejecutados. Estos pueden ser tokens que esperen en un merge al resto de ramas, que representen a una tarea que necesita de un recurso que no está disponible o que representen flujos asociados a eventos,

los cuales no pueden ser consumidos hasta que el evento se active (en el caso de un mensaje por su recepción y en el de un timer cuando llegue a cero). Para evitar estados de bloqueo se proporciona una función que busca el siguiente token activo en este caso en el scheduler y lo mueve hacia la cabeza de la cola.

El **conjunto de eventos** del objeto Simulation funciona de manera que cada vez que se active uno, por ejemplo, cuando una tarea envía un mensaje, se crea un elemento que lo represente en dicho conjunto, el cual será una condición necesaria para que se activen las reglas de reescritura de ciertas tareas o puertas basadas en eventos.

Por su parte, el **conjunto de recursos** almacena para cada recurso, el nombre, las instancias disponibles, el tiempo total en el que el recurso ha sido usado (resource usage) y los intervalos de tiempo de uso. También se usa un último objeto llamado **workload**, que es una forma ampliamente extendida para parametrizar análisis basados en simulaciones. Permiten definir, por ejemplo, el ratio con el cual se inician ejecuciones de nuevos procesos y el número mínimo y máximo de ejecuciones concurrentes con el que se quiere trabajar.

2.1.2.2.4. Análisis formales

Las técnicas utilizadas en esta iteración se centran en obtener el tiempo de ejecución y distintas métricas relacionadas con los recursos para este tipo de procesos, las cuales ganan interés al poder extraerse de la ejecución de múltiples simulaciones concurrentes que compiten por recursos compartidos. Concretamente, para el tiempo se proponen análisis para obtener, entre otras, el **tiempo de ejecución medio** de los procesos (AET, del inglés Average Execution Time) y el **tiempo medio de sincronización en puertas** (AST, del inglés Average Synchronization Time), que representa el tiempo mínimo de sincronización entre los tokens necesarios para activar un merge inclusivo o paralelo, es decir, el tiempo que transcurre desde que llega el primer token a uno de los flujos de entrada del merge hasta que llega el último de los necesarios para activarlo.

En cuanto a los **recursos**, se computan distintas propiedades:

- GTU: El tiempo global de uso teniendo en cuenta todas las instancias para un recurso dado, cuando se ejecutan un número dado de simulaciones concurrentes de un diagrama.

- Average GTU: La media del tiempo global de uso para cada instancia de un recurso dado sobre una serie de ejecuciones concurrentes de un proceso.

- UP: El porcentaje de uso de un recurso con respecto al tiempo global que tardan en realizarse un número dado de simulaciones concurrentes de un proceso.

Para ello se usa la información que se va almacenando en el objeto Simulation durante el transcurso de las ejecuciones. Cuando todas terminan, se calculan las propiedades mencionadas y se proporcionan como salida. La figura 17 muestra los resultados de ejecutar el diagrama de ejemplo de la figura 14 sobre 100, 200, 400, 800 y 1600 simulaciones concurrentes, reflejando entre otras el AET, el AST de un merge (g8) y de un evento de fin (ee), el tiempo total de ejecución y las propiedades referentes a los recursos junto con el tiempo real que tarda en realizarse el análisis en segundos.

Num. inst.	AET	Var	AST_{g8}	AST_{ee}	Total time	Resources						Anal. time
						GTU_e	GTU_e^1	UP_e	GTU_d	GTU_d^1	UP_d	
100	107	190	70	57	327	316	158	48	852	284	87	6s
200	160	37	81	108	599	506	253	42	1665	555	93	30s
400	301	213	107	252	1202	1202	601	43	3493	1164	97	225s
800	550	4	156	501	2367	1978	989	42	6953	2318	98	1748s
1600	910	56	250	862	4263	4005	2003	47	12648	4216	99	11828s

Figura 17: Resultados experimentales obtenidos del ejemplo propuesto. Fuente: [12]

Finalmente, en el artículo se propone pero **no se implementa** un tipo de análisis que se puede realizar sobre los diagramas gracias a la información extraída. Este análisis tiene especial importancia porque se ha implementado e integrado en la aplicación web como parte del trabajo. El análisis propuesto tiene como objetivo encontrar la **combinación óptima de recursos**. La idea reside en obtener un valor de *coste* para cada una de todas las combinaciones posibles. El problema se acota haciendo que el usuario especifique para cada recurso el rango de instancias disponibles. Por ejemplo, tres recursos A, B y C con un rango de [1,2,3...,10] instancias cada uno. De esta forma se podría calcular el valor de coste para cada combinación (p. ej., A=3, B=2 y C=10) y se escogería como combinación final aquella con valor mínimo.

Este valor es el resultado de una función, llamémosla *F*, que recibe el AET resultante de las simulaciones para una combinación de recursos de entrada y el coste por unidad de tiempo de cada recurso. Destacar que ciertos parámetros adicionales deben ser proporcionados por el

usuario como el coste por unidad de tiempo y el ya mencionado rango de instancias para cada recurso.

Obtener la combinación óptima de recursos para un proceso de negocio es algo extremadamente útil, pero por desgracia esta estrategia no es factible para un caso real que necesite de un gran número de simulaciones concurrentes, de un elevado número de recursos o de un gran rango de instancias. Con el ejemplo tan pequeño que hemos puesto con los recursos A, B y C y sus respectivos rangos, se tendría que aplicar F sobre las mil combinaciones posibles con todas las múltiples simulaciones concurrentes que se deben realizar sobre cada combinación. Si para hacer los números redondos suponemos un número de mil ejecuciones concurrentes por combinación, se deberían realizar un millón de simulaciones para obtener el resultado óptimo.

Por tanto, para que el análisis se pueda aplicar a casos reales, se propone crear un **algoritmo ávido** que encuentre en mucho menos tiempo una solución aceptable, dentro de un *mínimo local*. Este enfoque se basa en partir de una combinación inicial y aplicar F sobre ella para obtener ese valor de coste. Posteriormente, se continúa ejecutando el mismo número de simulaciones para todas las combinaciones *adyacentes* de recursos, escogiendo aquella con un valor de coste menor y repitiendo este proceso hasta que se obtiene un mínimo local de F, devolviendo su combinación como resultado. Destacar que con adyacentes nos referimos a toda combinación en la que el número de instancias para al menos un recurso cambie aumentando o disminuyendo su valor en uno. Estas combinaciones adyacentes serían los vecinos de la posición que se está procesando en el *espacio de búsqueda* del algoritmo.

2.1.3. Documento General de Requisitos (DGR)

Tras investigar los análisis formales presentados en Maude, sobre los cuales se iba a desarrollar este proyecto, fue el momento de definir formalmente los requisitos funcionales y no funcionales que se pretendían abarcar con la aplicación. El proceso de obtención de requisitos fue complejo y se fue adaptando a medida que se iba avanzando con el proyecto. Finalmente, se obtuvo el conjunto de requisitos funcionales y no funcionales especificados en la parte del Anexo dedicada al **Documento General de Requisitos** (Véase el apéndice A). Los requisitos se han implementado en su totalidad entre las dos iteraciones que conforman este proyecto. Más adelante, se explicará cuáles fueron desarrollados en cada una y en qué orden.

2.1.4. Diagramas UML

Como parte del trabajo se ha desarrollado un **diagrama UML (Unified Modelling Language)** del Parser que se quería desarrollar, ya que esta parte de la aplicación sigue los principios de la programación orientada a objetos para definir una jerarquía de clases en torno a los elementos de BPMN de forma que cada uno tenga un comportamiento distinto. De esta forma, el diagrama ayudaría a su implementación y a su comprensión por parte de otro desarrollador que pudiera continuar con el proyecto en un futuro.

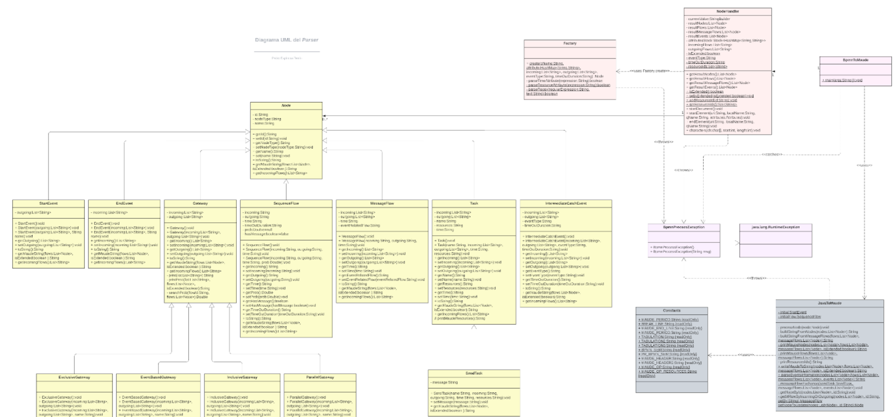


Figura 18: Diagrama UML del parser al completo.

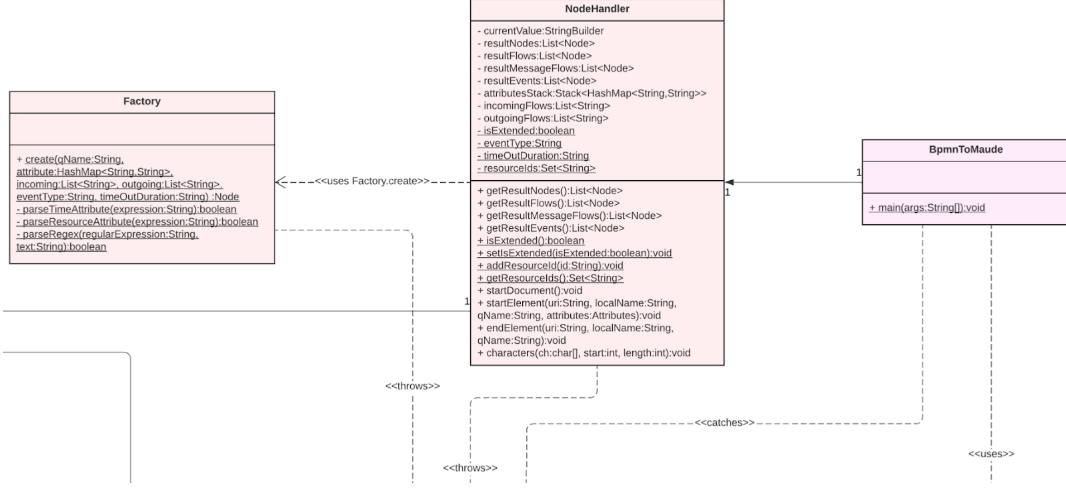


Figura 19: UML centrado en los paquetes xml2Java y main (1 / 4).

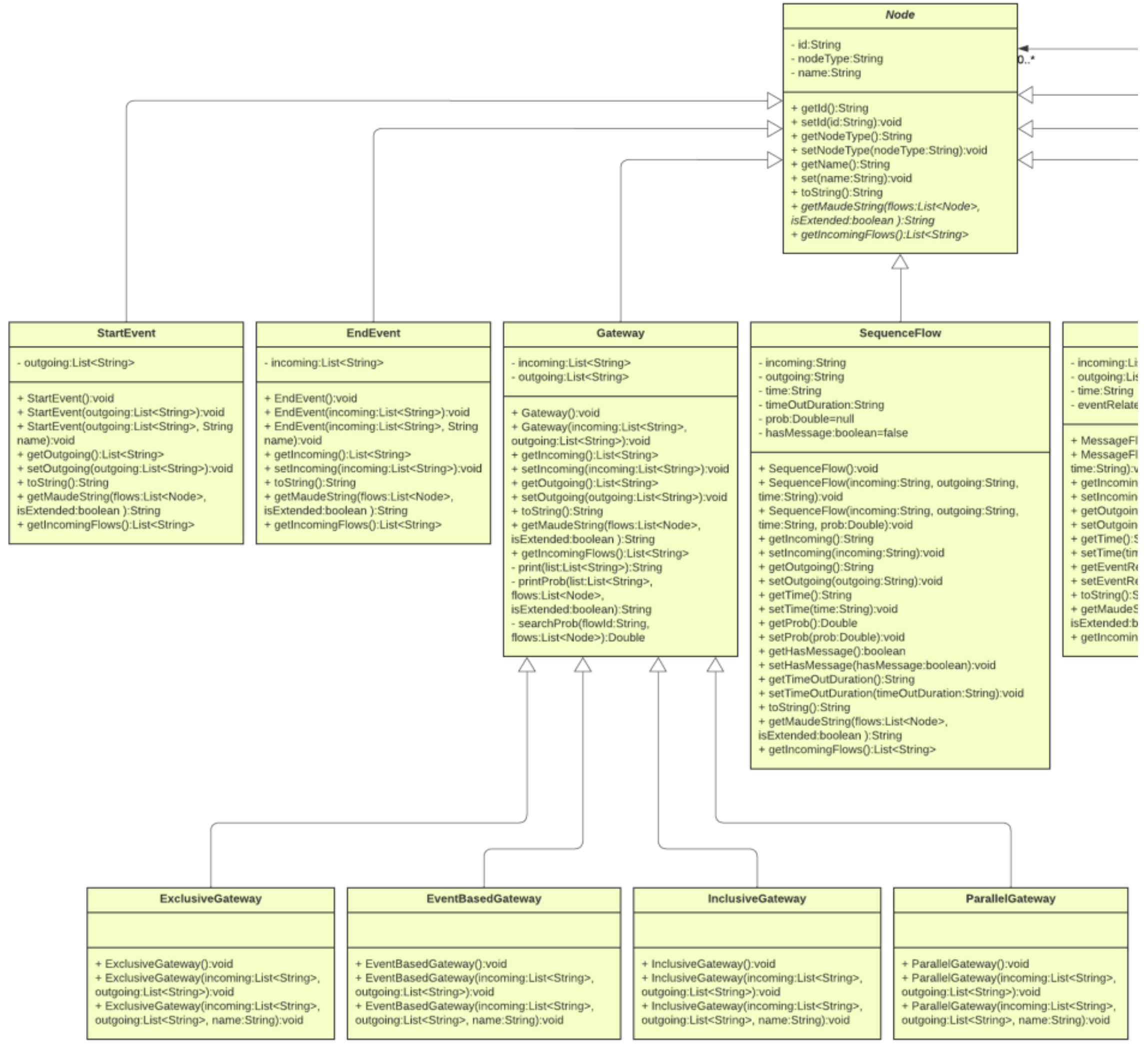


Figura 20: UML centrado en el paquete Elements (2 / 4).

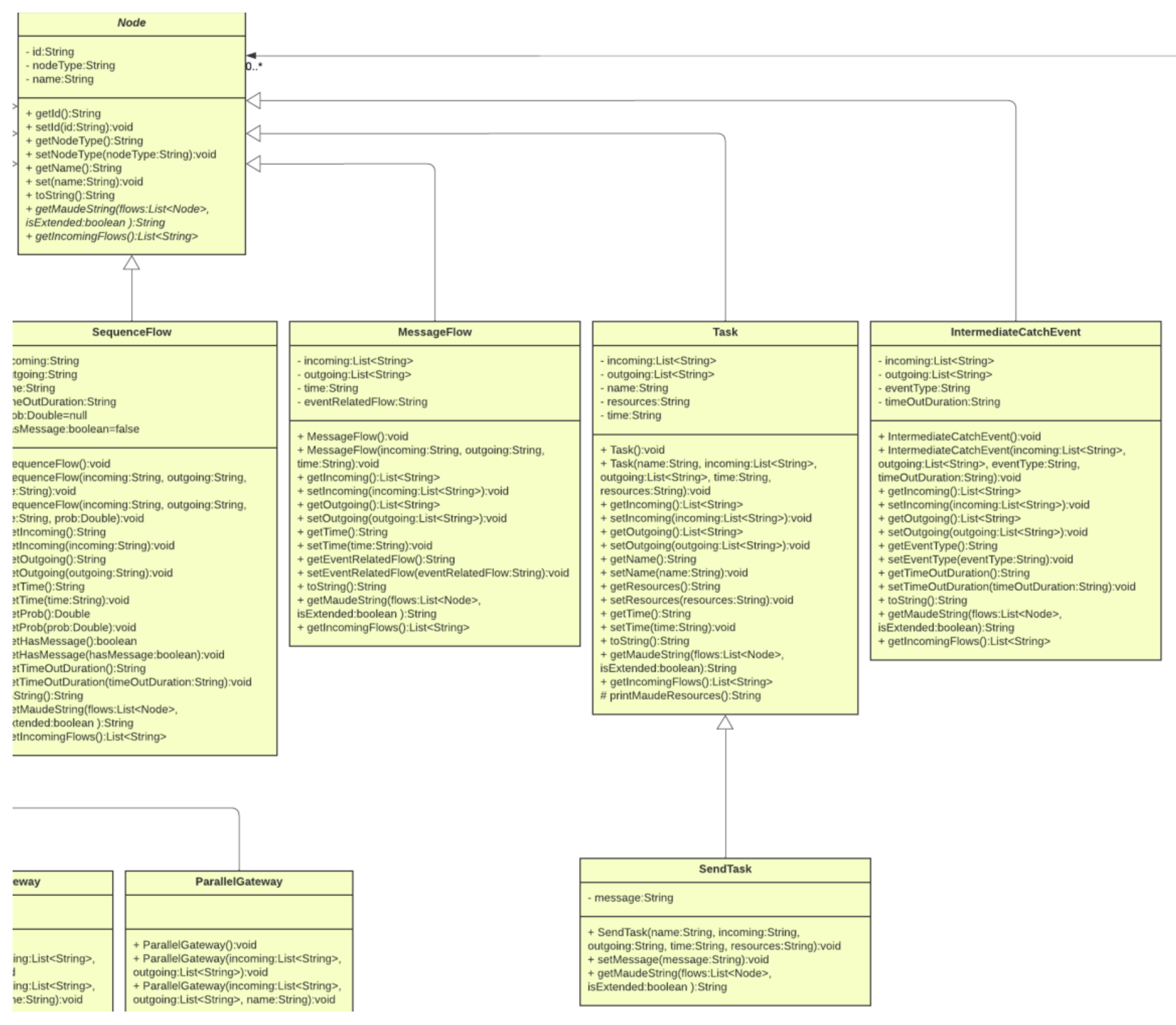


Figura 21: UML centrado en el paquete Elements (3 / 4).

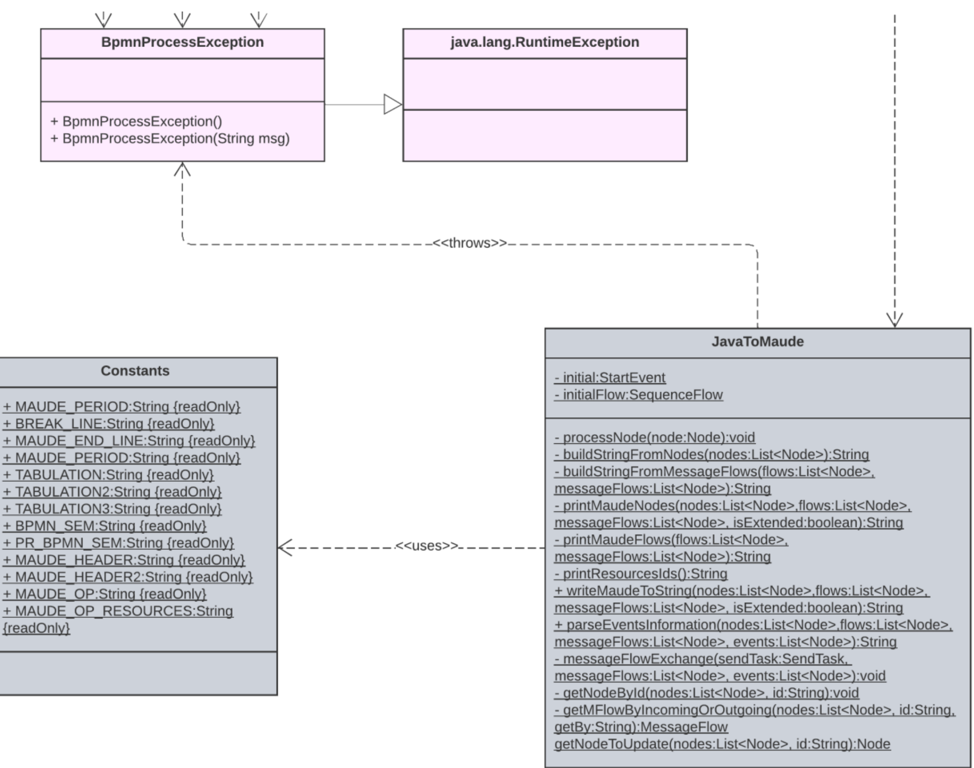


Figura 22: UML centrado en los paquetes java2Maude y Exception (4 / 4)

La figura 18, muestra la visión general del diagrama donde se pueden ver todas las clases definidas para traducir los ficheros XML que representan los diagramas a la notación Maude definida. En ella se pueden también las relaciones que existen entre ellas. Dado el elevado número de clases se muestra una figura centrada en cada paquete del proyecto.

La figura 19 muestra el paquete xml2Java y el main, que contiene la clase **BpmnToMaude**, donde se encuentra el método main que inicia el flujo. Como su propio nombre indica, parte de un diagrama *BPMN* y genera su representación *Maude*. A alto nivel, en primer lugar utiliza un objeto de la clase **NodeHandler**, la cual emplea la librería *SAX* para leer el fichero y almacenar la información de los nodos. La información de cada elemento se almacena en su objeto correspondiente usando el *patrón de diseño* factoría implementado en la clase **Factory**. Cada elemento será representado por una de las clases definidas en el paquete Elements, figuras 20 y 21. Como podemos ver, todas las clases heredan de la clase abstracta **Node**, que define dos métodos abstractos que cada subclase implementa de una forma diferente. Finalmente, una vez la clase NodeHandler devuelve el conjunto de nodos presentes en el diagrama, desde el *main* se interacciona con el método estático de la clase **JavaToMaude** que genera la notación

Maude, apoyándose en la clase **Constants** que define un conjunto de construcciones comunes del lenguaje, figura 22.

2.1.5. Diagramas de secuencia

Para el resto de la aplicación, se ha creado un **diagrama de secuencia** que explica el flujo de datos en la aplicación y cómo interactúan entre sí los distintos componentes que la forman, figura 23.

Los mensajes comprendidos en el rango **[1,8]** muestran la navegación que debe hacer el usuario hasta llegar a la página principal, donde realizará los análisis. Los mensajes **9 y 10** muestran el intercambio de información que realizan cliente y servidor necesaria para que el usuario pueda detener un análisis en proceso a voluntad. Desde el **11 hasta el 14** el usuario interactúa con el cliente para crear o cargar un diagrama BPMN y escoger qué análisis realizar. Posteriormente, el cliente envía un JSON con la información necesaria para que el servidor realice el análisis, paso **15**. Esta información es el fichero XML que contiene la representación del diagrama, la sesión del usuario, el tipo de análisis a realizar y su conjunto de parámetros (solo para aquellos análisis que necesitan de estos). Después, en el servidor se realizan los pasos **16 y 18** que consisten en interactuar con el Parser para obtener la notación Maude del diagrama.

Figura 23: Diagrama de Secuencia del flujo principal de la aplicación.

El paso **17** se da entre estos dos y consiste en un mensaje asíncrono del cliente al usuario que le muestra un formulario con el tiempo que lleva realizándose el análisis y con un botón que le permitirá detener el proceso en cualquier momento. Este formulario solo aparece para aquellos análisis que duran más de un segundo. Luego, en los pasos **19 y 20** el servidor crea un proceso del sistema operativo para poder ejecutar el análisis Maude de manera asíncrona al mismo tiempo que puede seguir procesando solicitudes. Por su parte, el proceso envía su PID para poder asociarlo a la sesión del usuario y así saber qué proceso se debe detener en caso de que el usuario pulse el botón, paso **21**. A partir de aquí, en cualquier momento se puede ejecutar el conjunto de instrucciones **Opt**, justo cuando el usuario presione el botón de Stop Analysis del formulario mostrado en el paso 17. Esto se realiza por medio de una petición HTTP POST que contiene la sesión del cliente hacia la ruta /stop, lo que hará que el servidor obtenga el PID del proceso (**21**) y le envíe un SIGKILL. El proceso recibirá la señal y se detendrá devolviendo al servidor su código de salida, de forma que este le notificará al cliente que el proceso se ha detenido correctamente, cosa que por su parte le notificará al usuario en pantalla.

En caso contrario, el servidor le dará la orden al proceso de realizar el análisis y esperará su respuesta, pasos **28, 29 y 30**. El resto de la ejecución consiste en la transmisión de la información resultante del análisis hacia el cliente y el procesamiento de esta para mostrarla en la pantalla del usuario.

2.1.6. Figma

L herramienta gráfica **Figma** fue utilizada para realizar el diseño de las páginas que formarían parte del sitio web. Figma es una herramienta web colaborativa que permite la importación de plantillas e iconos en los diseños haciendo que se puedan realizar de una forma más eficiente. Además, tiene una opción que te permite generar el Css3 de ciertos elementos simples de la plantilla para que el desarrollador pueda generar de forma semiautomática el tamaño, la fuente de la letra y los colores, facilitando y agilizando su trabajo. En las figuras 23, 24 y 25 se pueden ver los diseños que se realizaron con Figma que, como se mostrará más adelante, no están muy alejados del diseño final de la aplicación. Concretamente, la figura 24 muestra la *Landing Page*, término con el que se hace referencia a la primera página que verá todo usuario de la aplicación. En ella se pretende explicar el propósito de la página y como funciona a alto nivel. Además, se proporciona una sección con enlaces a los papers en los que

se basan los análisis, para que el usuario interesado pueda aumentar sus conocimientos en el tema. Por su parte, la figura 25, representa la página principal del sitio web. En ella, el usuario podrá crear o cargar los diagramas BPMN, para posteriormente analizarlos y observar los resultados. Por último, se diseñó el aspecto que tendrían los diversos formularios utilizados en la web para, entre otras cosas, especificar información previa a algunos tipos de análisis, figura 26.

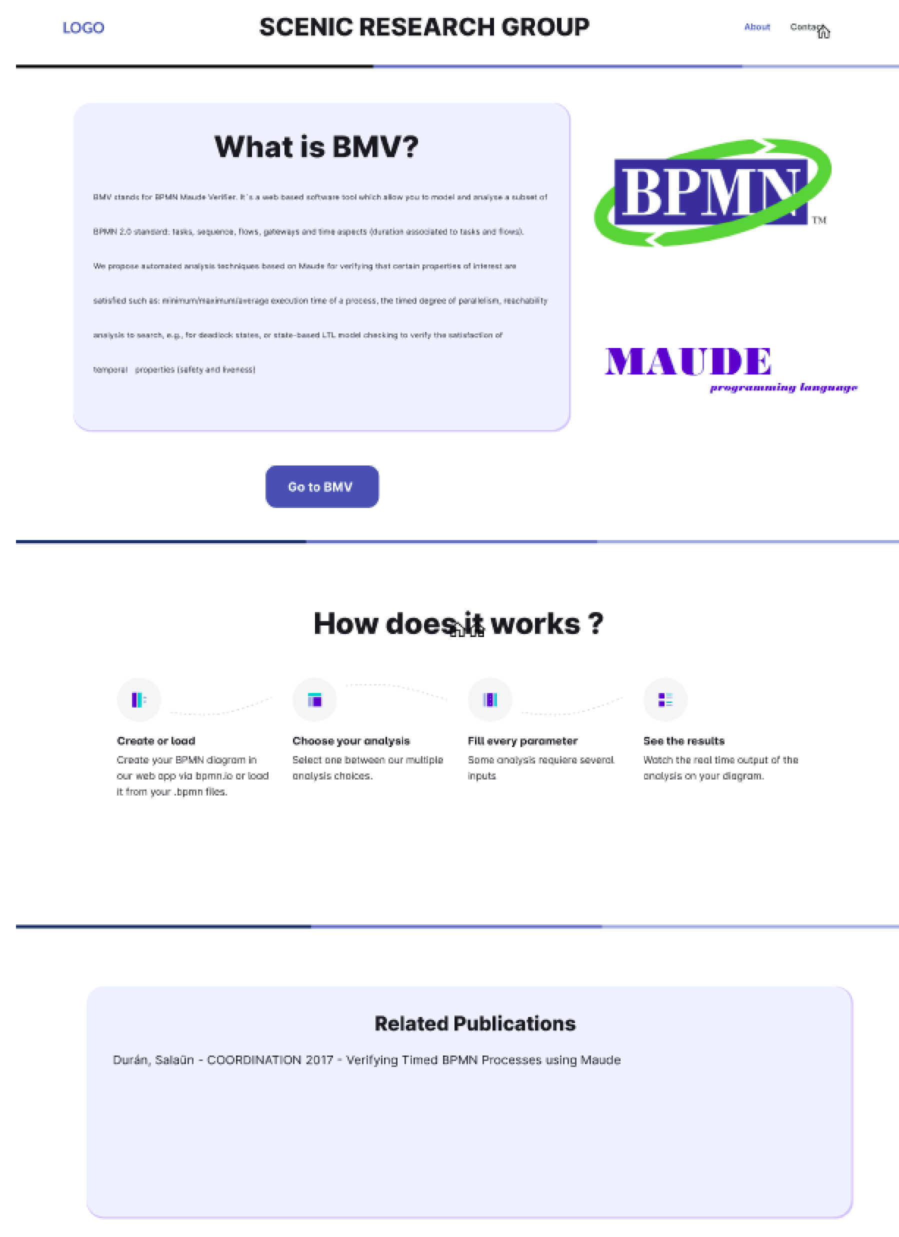


Figura 24: Diseño gráfico del Landing Page.

Figura 25: Diseño gráfico de la página principal.

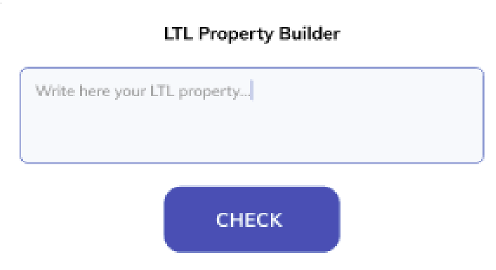


Figura 26: Diseño gráfico del formulario LTL.

2.1.7. Selección de los requisitos a implementar en cada iteración

El último paso de esta iteración cero fue escoger qué requisitos se iban a implementar por cada fase. Recordemos que la metodología escogida fue la iterativa incremental, con el propósito de ser capaces de crear en cada fase una versión final del producto sobre la cual fuera relativamente sencillo continuar añadiendo funcionalidades.

En este punto, se encontró una distinción clara entre los análisis formales que se podían realizar sobre los diagramas. Esta no era otra que la diferencia con respecto al subconjunto de elementos de BPMN con el que interactuaba cada análisis, ya que los que se basan en los recursos, presentados en los tres últimos papers comprenden un subconjunto bastante más complejo de BPMN (eventos de mensajes y de temporizadores, puertas basadas en eventos, mensajes entre tareas, etc.) el cual se debe representar en Maude. De esta forma, se decidió implementar en la iteración uno los análisis *temporales, de grado de paralelismo, de model check y de liveness*, dejando el resto para la segunda. Por ende, se desarrollaría una primera versión de la aplicación en la primera iteración, la cual sería posteriormente extendida con los requisitos implementados en la segunda. Prestando especial atención a la extensión del *Parser*, que es el componente encargado de generar la representación Maude y sería el que presentaría un mayor número de cambios.

2.2. Iteración uno

2.2.1. Análisis de requisitos

Como se ha comentado, el objetivo de esta fase es la implementación una primera versión funcional de la aplicación, creando cada uno de los componentes necesarios para realizar a través de la ella análisis temporales, del grado de paralelismo y de model checking. Para ello, se implementaron los requisitos funcionales y no funcionales detallados en la siguiente página.

Destacar que los análisis y el subconjunto de BPMN que se tiene en cuenta en esta iteración es más sencillo que el de la segunda. Sin embargo, esto se compensa con el hecho de que aquí se parte de cero, por lo que es necesario crear toda la infraestructura del cliente, servidor, parser y sus conexiones entre ellos.

Requisitos funcionales

Identificador	Nombre	Información	Dependencias
RF - 01	Crear diagrama	Se permitirá la creación de diagramas BPMN directamente en la aplicación de forma gráfica, sencilla y eficiente.	RNF - 01
RF - 02	Cargar diagrama	Se permitirá la visualización de diagramas BPMN que cumplan con el estándar 2.0 seleccionando un archivo local con extensión .bpmn.	RNF - 01
RF - 03	Editar diagrama	Se permitirá la edición de diagramas BPMN creados o cargados en la aplicación.	RNF - 01
RF - 04	Descargar diagrama	Se facilitará la descarga de los diagramas creados, cargados o editados en la aplicación a través de un botón ubicado sobre el modeler.	RNF - 01
RF - 05	Analizar diagrama	Se podrán realizar análisis formales sobre cada uno de los diagramas.	RNF - 02
RF - 06	Tiempo de ejecución	Se computará el tiempo de ejecución mínimo, máximo y medio.	RF - 05
RF - 07	Grado de paralelismo	Se calculará el grado de paralelismo. Esto es, el máximo número de tareas que se pueden estar ejecutando de manera simultánea.	RF - 05
RF - 08	Model Check	Permitirá comprobar propiedades de lógica temporal lineal siguiendo una semántica basada en tokens.	RF - 05 RNF - 03 RNF - 04
RF - 09	Liveness	Permite verificar si el diagrama está libre de deadlocks. Esto es, si termina para todas sus posibles ejecuciones.	RF - 05
RF - 18	Procesar salidas	Todos y cada uno de los análisis devuelven información no estructurada que requiere conocimiento experto para ser de utilidad. Por ende, se proporcionará un mecanismo para dar formato a las salidas y hacer que sean útiles para cualquier usuario.	RF - 05
RF - 19	Parser	Los diagramas BPMN están representados internamente por documentos XML, pero los análisis formales desarrollados en Maude necesitan de una especificación determinada	-

		de los procesos para funcionar. Por tanto se desarrollará un algoritmo que permitirá generar una representación Maude correcta a partir de todo diagrama BPMN válido.	

RF - 21	Página de bienvenida	Se dedicará una espacio del sitio web para explicar de manera general que es y como se puede utilizar la web, proporcionando referencias a los papers publicados por mi tutor y su equipo que tienen relación con la página para que cualquier usuario interesado pueda tener más información sobre cada tipo de análisis.	-
RF - 22	Parar análisis	Se debe permitir que los usuarios puedan parar un análisis que se encuentre en ejecución de forma sencilla y rápida	RF - 05 RNF - 03 RNF - 04
RF - 23	Mostrar tiempo	Para los análisis no inmediatos (duración superior a 2s) se mostrará un cronómetro dinámico que le vaya indicando al usuario cuánto tiempo lleva realizándose el proceso.	RF - 05 RNF - 03 RNF - 04
RF - 24	Añadir información adicional	Se debe permitir que el usuario añada a los diagramas la información adicional necesaria para realizar los análisis de forma fácil e intuitiva.	-

Requisitos no funcionales

Identificador	Nombre	Información	Dependencias
RNF - 01	Cumplir con el estándar	Todos y cada uno de los diagramas BPMN usados en la aplicación deberán cumplir con el estándar 2.0 de BPMN.	RF - 01 RF - 02 RF - 03 RF - 04
RNF - 02	Diagramas válidos	Se deberá informar al usuario acerca de si su diagrama está correctamente diseñado antes de ser analizado.	RF-05
RNF - 03	Mover formularios	Se deben poder arrastrar los formularios para facilitar el escribir sobre ellos al mismo tiempo que se puede observar el diagrama, de forma que la utilización de la página sea más cómoda para el usuario.	RF - 08 RF - 09 RF - 10 RF - 21
RNF - 04	Cerrar formularios	Se deben poder cerrar todos los formularios pulsando un botón sin necesidad de realizar el análisis.	RF - 21

2.2.2. Implementación

2.2.2.1. Configuración inicial

Al ser un proyecto realizado en **node.js**, se utilizan las facilidades que aporta su manejador de paquetes (npm) para administrar las dependencias. Por ello, se utiliza el comando *npm init*, el cual se utiliza para crear un paquete donde unificar todo el código y sus dependencias. Estas se almacenan en dos ficheros JSON (generados y administrados por los comandos de npm) llamados **package y package-lock** en los que se anotan las dependencias del código y sus versiones para que, entre otras cosas, otro desarrollador pueda instalarlas todas a la vez. Lo más común es utilizar el comando con el flag -y que omite el asistente y crea un package.json por defecto con el nombre del proyecto, el cual tiene una estructura similar a la siguiente:

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Figura 27: Package.json por defecto.

Este archivo no tiene aún ninguna dependencia, pero de estas existir, se encontrarían bajo el campo *dependencies y devDependencies.*

Destacar que el campo scripts, permite escribir comandos que pueden ser ejecutados escribiendo *npm run scriptname* desde el directorio donde se encuentra el fichero. En este caso se podría ejecutar *npm run test* que únicamente escribiría el mensaje indicado en la figura junto con el comando exit 1 que cerraría el CLI.

Nuestro proyecto utiliza Webpack, como se explicó previamente. Por ello, esta característica de npm que permite la ejecución sencilla de scripts nos ha sido útil para interactuar con las utilidades de webpack de forma rápida y legible. El campo scripts del proyecto final tiene la siguiente apariencia:

```
"scripts": {
  "buildDev": "rm -rf dist && webpack --mode development --config
              webpack.server.config.js && webpack --mode development --config
              webpack.dev.config.js",
  "buildDevDel": "rm -rf dist && rm serverDiagrams/*.bpmn && webpack --mode development
                 --config webpack.server.config.js && webpack --mode development
                 --config webpack.dev.config.js",
  "buildProd": "rm -rf dist && webpack --mode production --config
               webpack.server.config.js && webpack --mode production --config
               webpack.prod.config.js",
  "start": "node ./dist/server.js",
  "reboot": "npm run buildDev && npm start"
},
```

Figura 28: Campo scripts del archivo del package.json final.

Destacar que en Webpack se diferencian dos modos de trabajo, **desarrollo y producción**, debido a que se utilizan diferentes estrategias y características de Webpack propias de cada una de las fases. Por ejemplo, en la fase de producción, propuesta para trabajos futuros, se puede utilizar Webpack para realizar minify y uglify sobre los ficheros para reducir su peso, de forma que se rendericen más rápido en los navegadores.

Durante la fase de desarrollo, normalmente se ha utilizado el comando *npm run reboot* para cargar los cambios realizados en el servidor, este llama a los otros dos scripts buildDev y start. Como su propio nombre indica, buildDev se encarga de construir el proyecto para la fase de desarrollo. Para ello, elimina la carpeta dist, que como se explicará más tarde es la carpeta que genera Webpack con los ficheros unificados y después, inicia Webpack en el modo development y llama al script start que arranca el servidor web.

2.2.2.2. Página de bienvenida

Una vez se detalla la dinámica de trabajo con node.js y npm, podemos continuar explicando la fase de desarrollo. Se crea la **página de bienvenida**, aquella en la que se da información acerca de la página. Para ello, se partió del diseño de la figura 24 y se realizaron distintas modificaciones. Esta parte del proyecto se basa en utilizar HTML y Css3 y los resultados se almacenaron en los ficheros **.html y style.css**. Como se puede ver en las figuras 29 y 30, se realizaron diversos cambios en el diseño, tales como el cambio del encabezado, ajuste de tamaños y la actualización del icono de Maude por uno más actual.

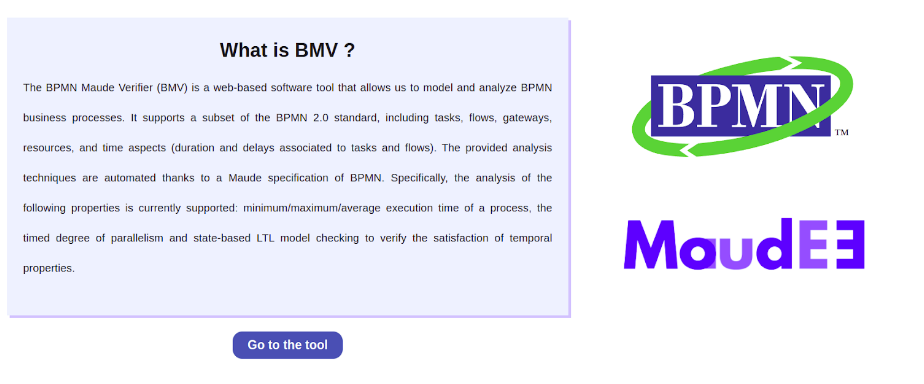


Figura 29: Landing Page (1 / 2).

Figura 30: Landing Page (2 / 2).

2.2.2.3. Servidor web

El siguiente paso fue crear el **servidor** y hacer que el *endpoint raíz* redirigiera a esta página. Como se ha comentado, el servidor se ha desarrollado en Node.js junto con el framework **Express.js**.

```
import express from 'express';
const app = express(),
  DIST_DIR = __dirname,
  HTML_FILE_HOME = path.join(DIST_DIR, 'index.html'),
  HTML_FILE_BMV = path.join(DIST_DIR, 'bmv.html')
//Middleware
app.use(express.static(DIST_DIR));
app.use(express.urlencoded({
  extended: true
}));
app.use(express.json())
//Get home page
app.get('/', (req, res, next) => {
  res.sendFile(HTML_FILE_HOME);
});
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`App listening to ${PORT}....`);
  console.log('Press Ctrl+C to quit.');
});
```

Figura 31: Versión inicial del servidor web.

Para ello, primero se instaló la librería con el comando *npm install express.js –save*. Después se importó en el servidor y se creó el objeto *app* de Express. Mediante este, se le proporcionaron al servidor las configuraciones de middleware necesarias. La primera se utiliza para permitir servir archivos estáticos como ficheros css, JavaScript o imágenes. Las otras dos, se encargan de recibir las solicitudes con formato json y hacer que su contenido sea accesible directamente como atributos del objeto *request*, simplificando el manejo de estas.

Posteriormente se define un endpoint que captura las peticiones GET a la URL por defecto de la aplicación ("/") y se devuelve directamente la página de bienvenida, con la función *sendFile*. Finalmente, seleccionamos el puerto en el que escuchará el servidor y lo arrancamos, quedando listo para atender peticiones.

2.2.2.4. Página principal

Para crear la **página principal** se partió del diseño de la figura 25, con la idea de que en el primer rectángulo se integrara el *modeler* de Camunda. Mientras que el segundo fuera donde se mostrarán los resultados. Además, se añadieron una serie de botones justo encima

del modeler para permitir la *creación, carga y descarga* de los diagramas, junto con otro que accionaría un desplegable con un botón para realizar cada uno de los tipos de análisis que se debían implementar. En la figura 32 se muestra la apariencia final de la página principal.

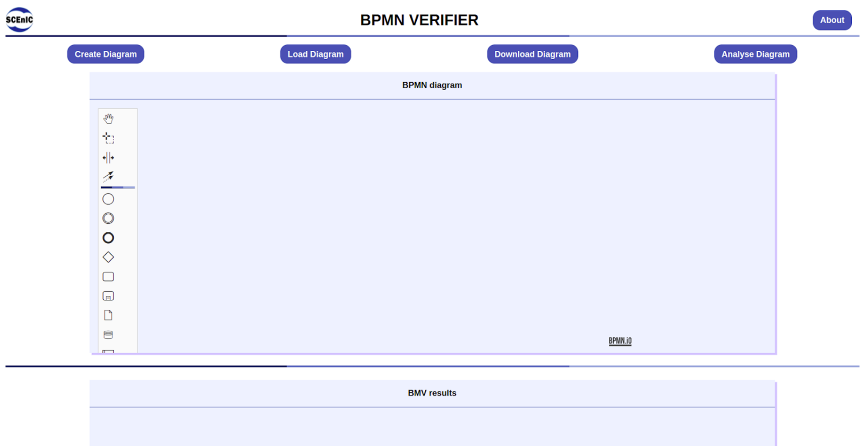


Figura 32: Página principal.

Para la programación de esta página, además de tener que implementar las interacciones con el servidor para cada análisis, se debía integrar el proyecto de Camunda para poder utilizar su modeler. Ello implicó familiarizarse con la documentación del proyecto, instalar sus dependencias y asegurarse de que todo funcionase correctamente.

Camunda tiene en su página un apartado que explica cómo instalar bpmn-js e incluirlo en tu aplicación [2]. Hay dos formas de hacerlo, la primera es instalar el modeler a modo de *caja negra* con la importación de un script de un repositorio de *unpkg.com* que ellos poseen. La segunda, la cual hemos usado nosotros, requiere un proceso de instalación más complejo mediante npm y Webpack, pero permite el accesof a cada uno de los componentes que forman el proyecto, facilitando extender sus funcionalidades, cosa que necesitábamos en nuestra aplicación para permitir a los usuarios agregar de forma gráfica las propiedades temporales, probabilistas y de recursos que quedan fuera del estándar y no están incluidas en su modeler.

Primero se instaló librería ejecutando el comando *npm install bpmn-js*. Después, fue necesario instalar Webpack e integrarlo en la aplicación, por todo lo comentado acerca de las importaciones de JavaScript en la introducción. De esta forma, podríamos unificar en los ficheros JavaScript del proyecto las dependencias de su librería con las nuestras. En la figura 33 se muestran las líneas del fichero **webpack.dev.config.js** que define la configuración de Webpack utilizada en el proyecto.

En ellas, básicamente se especifican los ficheros JavaScript de cada página que actúan como **entry points**, es decir, sobre los cuales se unificarán el resto. Se especifica el directorio de salida, el cual está en el *root* del proyecto bajo la carpeta dist. Aquí es donde se guardan los nuevos ficheros *.js, .css y .html* creados por la herramienta. Bajo el campo **rules**, se declaran las transformaciones que se realizarán sobre cada tipo de fichero y los *loaders* que actuarán sobre ellos. Por último, se utiliza un plugin llamado *HTMLWebPackPlugin* que permite especificar qué ficheros JS no serán usados por cada página HTML, ya que tenemos un fichero principal para cada una de las páginas.

```
module.exports = {
    entry: {
      main: './src/js/main.js',
      index:'./src/js/'
    },
    output: {
      path: path.join(__dirname, 'dist'),
      //publicPath: '/',
      filename: '[name].js'
    },
    mode:"development",
    target: 'web',
    devtool: 'source-map',
    module: {
      rules: [
        {
          test: /\.js$/,
          exclude: /node_modules/,
          loader: "babel-loader",
        },
        {
          test: /\.bpmn$/,
          use: 'raw-loader'
        },
        {
          // Entry point is set below in HtmlWebPackPlugin in Plugins
          test: /\.html$/,
          use: [
            {
              loader: "html-loader",
              //options: { minimize: true }
            }
          ]
        },
        {
          test: /\.css$/,
          use: [MiniCssExtractPlugin.loader, "css-loader"],
        }
      ]
    },
    plugins: [
      new HtmlWebPackPlugin({
        template: './src/html/index.html',
        filename: "./index.html",
        excludeChunks: [ 'server', 'main' ]
      }),
      new HtmlWebPackPlugin({
        template: './src/html/bmv.html',
        filename: "./bmv.html",
        excludeChunks: [ 'server', 'index' ]
      }),
      ]
  }
```

Figura 33: Fichero de configuración de Webpack.

El último paso se realiza en el fichero principal del código del cliente: **main.js**. Dicho paso consiste en acceder al modeler a través de una importación e instanciarlo para incluirlo en la página, además de referenciar los ficheros css descargados a través de la instalación del paquete npm (bpmn-js)y añadir el código que aporta la forma de incluir propiedades temporales, probabilísticas y de recursos.

La creación del modeler se puede ver en la figura 34, donde se especifica que se enlazará

con un elemento de HTML que tiene *canvas* como id. Como se comentó anteriormente, este elemento es el primer rectángulo de la vista principal.

```
// MODELER
const modeler = new Modeler({
  container: '#canvas',
  propertiesPanel: {
    parent: '#property-panel'
  },
  additionalModules: [
    propertiesPanelModule,
    propertiesProviderModule,
    timePropertiesProviderModule
  ],
  moddleExtensions: {
    time: timeModdleDescriptor,
    prob: probModdleDescriptor,
    res:resourceModdleDescriptor
  }
});
```

Figura 34: Creación del modeler en el fichero main.js.

En *additionalModules* se especifica el elemento **propertyPanel**, el cual está desarrollado por Camunda pero no viene incluido en el modeler por defecto. Es un menú lateral que aparece en el contenedor del diagrama y que permite acceder a los datos de cada nodo (nombre, id,...). Se decidió usar este panel debido a que, investigando la documentación del proyecto, se vio que se le podían añadir secciones, permitiendo dar información adicional a cada tipo de nodo, lo cual era justo lo que se necesitaba conseguir.

El parámetro principal es aditionalPropertiesProviderModule. Este módulo crea una sección dentro del panel para especificar cada campo (tiempo, recursos y probabilidades) y se las asocia únicamente a las tareas y a los flujos. En la figura 35 se puede ver la vista que mostraría el panel por defecto, llamada *General*, junto con la sección *Time*. Por su parte, en la figura 37, se pueden ver respectivamente las secciones *Resources y Probability*, creadas como parte de este proyecto. Detallaremos únicamente la implementación de la sección en referencia al **tiempo**, ya que el resto es similar.

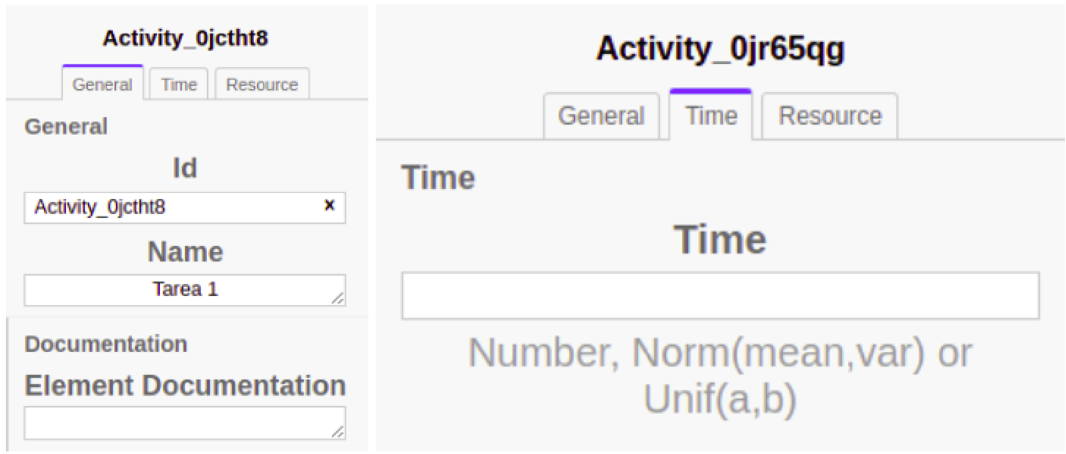


Figura 35: Secciones General y Time del propery panel.

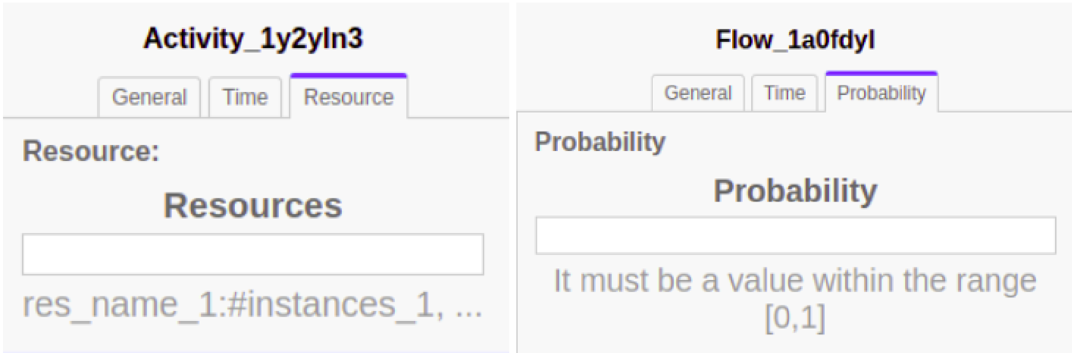


Figura 36: Secciones Resource y Probability del property panel.

Como hemos comentado, *aditionalPropertiesProviderModule* es el fichero principal a la hora de extender la funcionalidad del properyPanel de bpmn-js.

Para ello, se asocia un evento al manejador del panel por cada sección que se quiere añadir, figura 37. Se ha programado de forma que estos eventos comprobarán si el nodo pulsado por el usuario es de un tipo que pueda necesitar de una o varias de estas secciones adicionales, como podrían ser las tareas o los flujos. De manera que de ser el caso, se habiliten los campos necesarios para que el usuario pueda introducir la información.

```
export default function(group, element, translate) {
  if (is(element, 'bpmn:SequenceFlow') || is(element, 'bpmn:Task')) {
    group.entries.push(entryFactory.textField(translate, {
      id : 'Time',
      description : 'Number, Norm(mean,var) or Unif(a,b)',
      modelProperty : 'time'
    }));
  }
}
```

Figura 37: Función del fichero time.js

En el caso del tiempo, se comprueba que el elemento seleccionado ha sido bien una tarea o un flujo (únicos nodos que modelan tiempo) añadiendo al panel de la sección un pequeño formulario dándole información al usuario al mismo tiempo que le permite escribir el valor temporal asociado al nodo, véase de nuevo la figura 35.

Además de esto, se ve en la creación del modeler otro campo llamado *moodleExtensions* que hace referencia a un descriptor *moodle* por cada sección. Estos descriptores son unos archivos JSON que se deben crear con determinados campos que serán utilizados por la librería para incluir la información que el usuario ha escrito en el panel en el fichero XML que representa el diagrama. Esto es necesario para guardar dicha información y poder procesarla en el Parser.

Se continuó interaccionando con el modeler al mismo tiempo que se desarrollaron un conjunto de funciones también en el lado del cliente para terminar de implementar los requisitos **RF - 01, RF - 02, RF - 03 y RF - 04**, que se corresponden respectivamente con la creación, carga, edición y descarga de diagramas BPMN. La figura 38 muestra cómo se puede visualizar en la aplicación el diagrama de ejemplo de la figura 02.

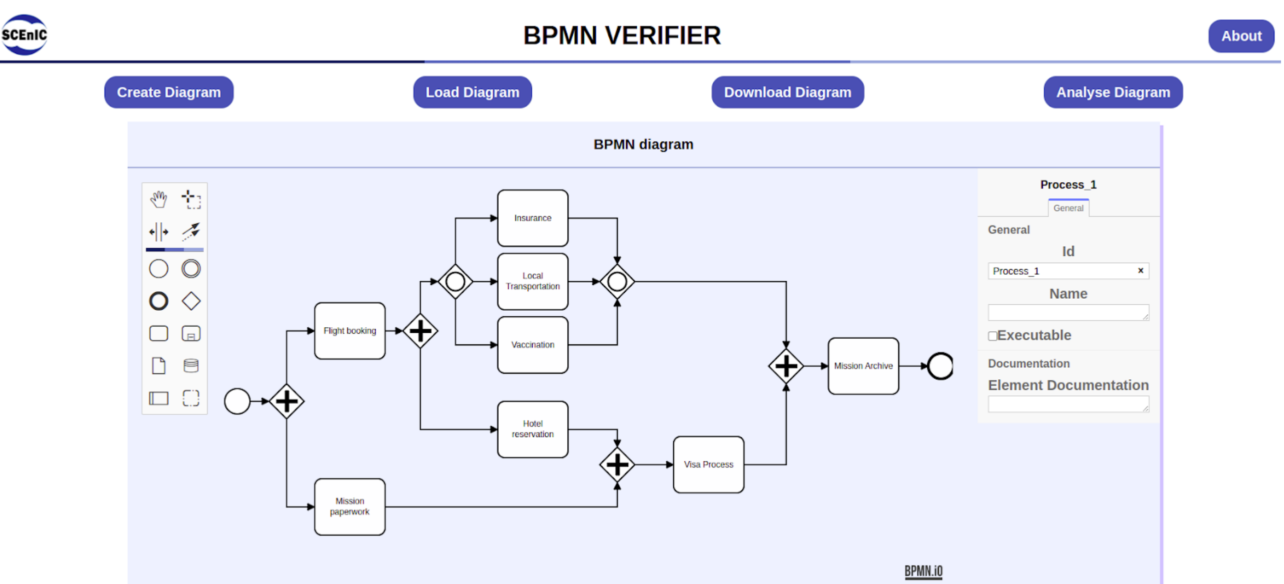


Figura 38: Diagrama de ejemplo creado en la aplicación.

La aplicación dispone de un **menú lateral** desplegable que muestra un botón para cada tipo de análisis el cual se acciona pulsando sobre el botón *Analyse Diagram*, arrastrando el contenido de la página a la derecha. De forma que cada análisis comienza cuando el usuario pulse sobre su correspondiente etiqueta, figura 39.

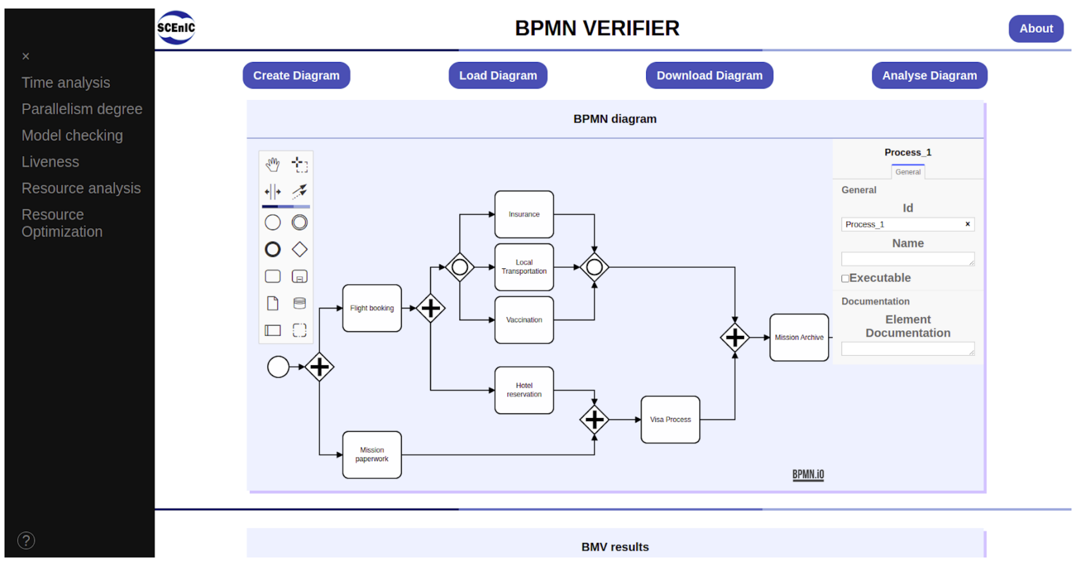


Figura 39: Menú de análisis.

En esta iteración sólo se implementaron los cuatro primeros análisis que se pueden ver en la figura. De ellos, todos son independientes del diagrama y comienzan en el momento en el que el usuario hace click sobre ellos, excepto el **model checking**. Como se explicó, el model checking permite comprobar si el diagrama satisface una proposición LTL, que puede ser comprobar si hay estados de bloqueo, si una tarea se realiza en todas las posibles ejecuciones, si se realiza una detrás de otra, etc. Para que el usuario aporte dichas proposiciones, la aplicación dispone de un formulario que aparece a modo de pop-up una vez que se pulse sobre su correspondiente etiqueta del menú.

El formulario se puede ver en la figura 40 e implementa, como todos los formularios de la web, los requisitos no funcionales **RNF - 03 y RNF - 04** que obligan que se puedan *arrastrar y cerrar*, para que el usuario pueda mover el formulario mientras visualiza distintas partes del diagrama y para que pueda cerrarlo sin tener que forzosamente realizar el análisis.

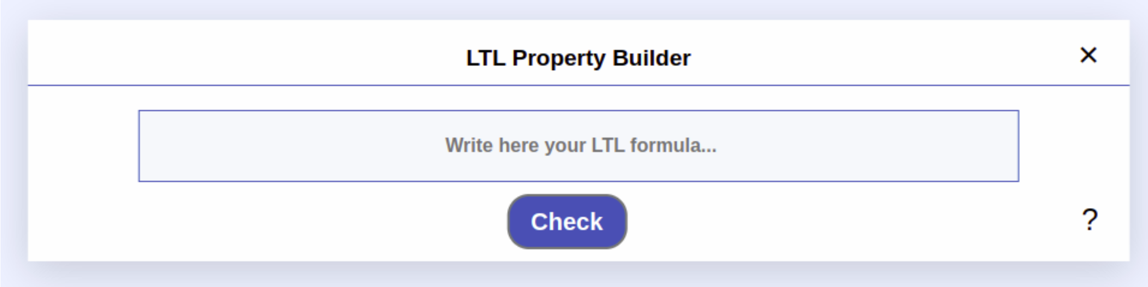


Figura 40: Formulario del análisis de model checking.

Si el usuario presiona sobre el icono "?" se muestra un pop-up que contiene información para el usuario y está presente en todos los formularios. Sin embargo, este requisito fue implementado en la segunda iteración, por lo que se contará en detalle más adelante.

La página principal presenta otro formulario. Este se utiliza para implementar dos requisitos funcionales: **RF - 23 y RF - 24**, que se encargan de que el usuario pueda *detener* un análisis en cualquier momento y de **mostrar el tiempo** que este lleva realizándose. Se decidió implementar esta funcionalidad debido a que ciertos procesos pueden tardar bastante y a que algunos diagramas pueden tener ciclos, haciendo que sus análisis no terminen nunca (realmente lo harán ya que la profundidad del espacio de búsqueda está limitado, pero puede tardar mucho tiempo). Además, al igual que el formulario del model check, implementa los requisitos no funcionales **RNF - 03 y RNF - 04**.

El cálculo del tiempo se implementó con las siguientes funciones:

```
//Chronometer settings for stop-form
//Auxiliar variables
var h,m,s,clockId;
//Init clock at 00:00:00
function initClock(){
  h = 0;
  m = 0;
  s = 0;
  document.getElementById("hms").innerHTML="00:00:00";
  runClock();
}
//Execute write function once per second
function runClock(){
  write();
  clockId = setInterval(write,1000);
}
//Increase the chronometer value by one taking into account minutes and hours.
function write(){
  var hAux, mAux, sAux;
  s++;
  if (s>59){m++;s=0;}
  if (m>59){h++;m=0;}
  if (h>24){h=0;}
  if (s<10){sAux="0"+s;}else{sAux=s;}
  if (m<10){mAux="0"+m;}else{mAux=m;}
  if (h<10){hAux="0"+h;}else{hAux=h;}
  document.getElementById("hms").innerHTML = hAux + ":" + mAux + ":" + sAux;
}
function debounce(fn, timeout) {
  var timer;
  return function() {
    if (timer) {
      clearTimeout(timer);
    }
    timer = setTimeout(fn, timeout);
  };
}
```

Figura 41: Funciones que implementan los requisitos RF - 23 y RF - 24.

Primero, se inicializa el cronómetro con todos los valores a cero y se llama a *runClock* que mediante la función *setInterval* se encargará de llamar a *write* una vez por segundo, actualizando el contenido del cronómetro. En la figura 42 se puede ver una imagen del formulario en cuestión. La funcionalidad de cómo se puede parar un análisis en proceso se explicará más adelante cuando se explique el flujo en el lado del servidor.

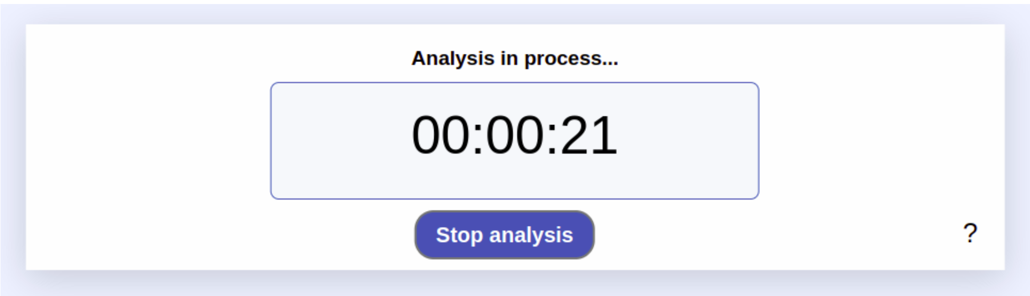


Figura 42: Formulario para ver el tiempo transcurrido y parar los análisis.

2.2.2.5. Parser

Una vez presentados todos los elementos gráficos del lado del cliente, pasamos a explicar el resto de componentes para la realización de los distintos tipos de análisis.

Como se puede ver en el diagrama de secuencia de la figura 23, al pulsar sobre cada tipo de análisis, lo primero que se realiza es el envío al *servidor* de los parámetros necesarios para poder llevarlos a cabo. Estos se envían mediante una petición POST en forma de JSON. Entre estos parámetros, se encuentra el fichero XML que representa al diagrama, que se obtiene a través una API de la herramienta bpmn-js. Posteriormente, en el servidor se llama al *Parser* para transformar el diagrama en su representación Maude equivalente, pudiendo realizar el análisis.

El Parser es un conjunto de programas escrito en Java y representado mediante el diagrama UML de la figura 18, el cual se ejecuta en el servidor a través de un *jar* que ejecuta su método main almacenado en la clase **Bpmn2Maude**. Este método lee el fichero .bpmn pasado como argumento, el cual contiene la representación del diagrama en formato XML, y llama a una clase denominada **NodeHandler** que es la encargada de leer el contenido del fichero utilizando la tecnología *SAX*, al mismo tiempo que va creando un conjunto de nodos y flujos con los que después se construirá la representación Maude. En la figura 43 se muestra la estructura de los ficheros XML que representan a los diagramas BPMN.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
  <bpmn2:process id="Process_1" isExecutable="false">
    <bpmn2:startEvent id="StartEvent_1">
      <bpmn2:outgoing>Flow_15n4v5t</bpmn2:outgoing>
    </bpmn2:startEvent>
    <bpmn2:sequenceFlow id="Flow_15n4v5t" sourceRef="StartEvent_1" targetRef="Gateway_13ieug4" />
    <bpmn2:task id="Activity_10an6ky" name="Flight booking" time:time="3">
      <bpmn2:incoming>Flow_1uifcvi</bpmn2:incoming>
      <bpmn2:outgoing>Flow_1yd5ajb</bpmn2:outgoing>
    </bpmn2:task>
```

Figura 43: Estructura de los ficheros .bpmn.

Con SAX se leen estos ficheros empezando desde arriba hacia abajo y ejecutando una serie de *callbacks* o funciones cuando va procesando distintos elementos. Concretamente, en esta implementación se han utilizado tres de sus métodos:

El primero es **startDocument**, que se ejecuta una vez antes de comenzar a leer el documento, figura 44. Se ha usado para inicializar las estructuras de datos necesarias. Concretamente, en esta iteración se utiliza únicamente una lista para almacenar los nodos, otra para los flujos y una pila para almacenar los atributos de cada elemento XML (el resto de elementos se detallan en la segunda iteración).

```java
@Override
public void startDocument() {
    resultNodes = new ArrayList<>();
    resultFlows = new ArrayList<>();
    resultMessageFlows = new ArrayList<>();
    resultEvents = new ArrayList<>();
    attributesStack = new Stack<>();
    resourceIds = new TreeSet<>();
}
```

Figura 44: Método 1, startDocument.

Por su parte, **startElement** es ejecutado cada vez que se procesa un nuevo elemento del fichero XML, figura 45. En este método, se crea un HashMap para almacenar los atributos del elemento en forma de clave-valor, emparejando el nombre del atributo en cuestión y su contenido. Este mapa será guardado en la pila mencionada anteriormente: *attributesStack*.

```java
@Override
public void startElement(String uri, String localName, String qName, Attributes attributes) {
    currentValue.setLength(0);
    HashMap<String, String> map = new HashMap<>();
    for (int i = 0; i < attributes.getLength(); i++)
        map.put(attributes.getLocalName(i), attributes.getValue(i));
    attributesStack.push(map);
}
```

Figura 45: Método 2, startElemen.

La razón de utilizar una pila es la siguiente: Hay ciertos elementos, como la tarea de **id=Activity_10an6ky** que podemos ver en el XML presentado, que tienen una serie de atributos y subelementos (en este caso un flujo de entrada y otro de salida) que deben ser procesados. Cada uno de estos subelementos volverán a accionar el método *StartElement*, haciendo que sea necesario volver almacenar sus atributos. Por ello, este tipo de estructura *FIFO* es muy conveniente, debido a que en la ejecución del tercer método, **endElement** (véase figura 46), el cual es ejecutado cuando se procesa la etiqueta de fin de cada elemento, se extraerán los elementos de la pila, permitiendo que se acceda a los atributos en cuestión de cada elemento una vez todos sus posibles subelementos hayan sido procesados y almacenados. En este ejemplo de la tarea de id=Activity_10an6ky, el método *endElement* será ejecutado una vez lo haya sido para sus flujos, extrayendo de la pila sus atributos: **name=“Flight Booking“ y time =“3“**, claro está teniendo almacenados ya en otra lista sus flujos de entrada y de salida.

```java
public void endElement(String uri, String localName, String qName) throws NumberFormatException {
    if (qName.equals("bpmn:incoming") || qName.equals("bpmn2:incoming")) {
        if (incomingFlows == null)
            incomingFlows = new ArrayList<>();
        incomingFlows.add(currentValue.toString().replace("_", ""));// Maude doesnt allow _ in identifiers
    }else if (qName.equals("bpmn:outgoing") || qName.equals("bpmn2:outgoing")) {
        if (outgoingFlows == null)
            outgoingFlows = new ArrayList<>();
        outgoingFlows.add(currentValue.toString().replace("_", ""));// Maude doesnt allow _ in identifiers
    }else if (qName.equals("bpmn:messageEventDefinition") || qName.equals("bpmn2:messageEventDefinition")) {
        eventType = "message";
    }else if (qName.equals("bpmn:timerEventDefinition") || qName.equals("bpmn2:timerEventDefinition")) {
        eventType = "timeout";
    }else if(qName.equals("bpmn:timeDuration") || qName.equals("bpmn2:timeDuration")) {
        timeOutDuration = currentValue.toString();
    }else{
        Node node = Factory.create(qName, attributesStack.peek(), incomingFlows, outgoingFlows,eventType,
                timeOutDuration);
        if (node != null) {
            //TODO do this in factory instead of here
            if (node instanceof SequenceFlow) resultFlows.add(node);
            else if (node instanceof MessageFlow) resultMessageFlows.add(node);
            else if(node instanceof IntermediateCatchEvent) resultEvents.add(node);
            else resultNodes.add(node);
            // Reset lists
            incomingFlows = new ArrayList<>();
            outgoingFlows = new ArrayList<>();
        }
    }
    attributesStack.pop();
}
```

Figura 46: Método 3, endElement.

Los métodos startDocument y startElement se encargan de almacenar la información mientras que **endElement** la recoge y crea cada tipo de nodo. Para hacerlo de una forma extensible y ordenada se aplica el patrón de diseño **Factoría** con el método estático *Factory.create* creando cada nodo (Tarea, Evento, Puerta exclusiva, Inclusiva...) en una clase separada y devolviendo un objeto de tipo Node.

Como se puede ver en los diagramas UML de las figuras 21 y 22, se crea la clase **Node** como abstracta y siendo la más alta de la jerarquía. Esto se hace para no tener que manipular una lista con elementos de distintos tipos, cosa que conllevaría tener que introducir numerosos *if-else* dentro del código para poder realizar el comportamiento apropiado según el tipo de nodo. Por el contrario, se definen de forma abstracta en la clase Node los métodos necesarios y se implementan de forma diferente en cada subclase, atendiendo a sus atributos. Por ejemplo, el método **getMaudeString** genera para cada nodo su representación Maude haciendo uso de la información que tiene almacenada.

En el método Factory.create se diferencia utilizando un *switch* el tipo de nodo que se debe crear mediante la cadena **qName** proporcionada por SAX, el cual se corresponde con la etiqueta del elemento XML, por ejemplo, bpmn:task para el caso de una tarea.

Además, se hace uso de los atributos almacenados en la pila añadiéndolos a cada tipo de nodo. Estos atributos pueden ser proporcionados por los usuarios a través del panel del diagrama. Por ello, se aplican diversos filtros utilizando **expresiones regulares** para controlar y validar que la información es correcta, propagando una excepción en caso contrario que evita la transformación y el análisis, informando al usuario de todo ello.

Una vez se ha recorrido el fichero XML, se tendrá almacenada en **resultNodes y resultFlows**, la lista de los nodos y flujos del diagrama con toda su información, respectivamente. Estas listas son extraídas en el método main y pasadas como argumento al método *writeMaudeString* (figura 47) de la clase **JavaToMaude**, que generará el resultado final con la notación Maude. Principalmente, este método utiliza un objeto de la clase **StringBuilder** para construir dinámicamente el string apoyándose en una clase llamada **Constants** que contiene un listado con construcciones Maude para simplificar el proceso y hacerlo más legible.

```
public static String writeMaudeToString(List<Node> nodes, List<Node> flows,List<Node> messageFlows,
        boolean isExtended){
    StringBuilder sb = new StringBuilder();
    if(isExtended) sb.append(Constants.MAUDE_HEADER_2);
    else sb.append(Constants.MAUDE_HEADER_1);
    sb.append(Constants.TABULATION + "ops " + buildStringFromNodes(nodes) + ": -> NId" + Constants.MAUDE_END_LINE);
    sb.append(Constants.TABULATION + "ops " + buildStringFromNodes(flows) + ": -> FId" + Constants.MAUDE_END_LINE);
    if(isExtended) {
        sb.append(buildStringFromMessageFlow(flows,messageFlows));
        sb.append(printResourceIds());
    }
    sb.append(Constants.MAUDE_OP);
    if(isExtended) sb.append(Constants.MAUDE_OP_RESOURCES);
    if(initial == null) {
        throw new BpmnProcessException("BPMN process does not have start event.");
    }else {
        if(!isExtended && initialFlow != null)sb.append(Constants.TABULATION + "eq init = token(" +
    initialFlow.getId() + "," + initialFlow.getTime() + ")" + Constants.MAUDE_END_LINE);
    }
    sb.append(Constants.TABULATION + "eq fls" + Constants.BREAK_LINE);
    sb.append(printMaudeFlows(flows, messageFlows));
    sb.append(Constants.TABULATION + "eq nds" + Constants.BREAK_LINE);
    sb.append(printMaudeNodes(nodes,flows, messageFlows,isExtended));
    sb.append("endm" + Constants.BREAK_LINE);
    return sb.toString();
}
```

Figura 47: Método writeMaudeToString.

En primer lugar se escribe la cabecera del módulo Maude, común para todos los diagramas. Luego se declaran los identificadores de cada elemento del diagrama, recorriendo la lista de nodos y flujos. El identificador en Maude debe ser una cadena de texto que represente inequívocamente al elemento. Para ello se usa el propio identificador que *bpmn-js* asigna a cada elemento creado con su modeler. En esta declaración de identificadores se busca el nodo de comienzo que debe estar presente en el conjunto de nodos para poder crear el token **initial** con su flujo asociado, ya que este debe estar declarado específicamente dentro del objeto Simulation para poder iniciar los análisis formales, como se explicó anteriormente. En caso de no existir, se lanza una **excepción** explicando el error. Finalmente, se construye la lista de nodos y flujos recopilando toda la información en los métodos *printMaudeNodes y printMaudeFlows*, figura 48.

```
private static String printMaudeNodes(List<Node> nodes, List<Node> flows, List<Node> messageFlows, boolean isExtended) {
    StringBuilder sb = new StringBuilder();
    StringJoiner str = new StringJoiner(",\n");
    sb.append(Constants.TABULATION2 + "= (" + Constants.BREAK_LINE);
    for(Node node : nodes) {
        if(node instanceof Elements.Gateway) {
            str.add(Constants.TABULATION3 + node.getMaudeString(flows,isExtended));
        }else if(node instanceof Elements.SendTask) {
            str.add(Constants.TABULATION3 + node.getMaudeString(messageFlows,isExtended));
        }else {
            str.add(Constants.TABULATION3 + node.getMaudeString(null,isExtended));
        }
    }
    sb.append(str.toString() + Constants.BREAK_LINE);
    sb.append(Constants.TABULATION2 + "  )" + Constants.MAUDE_END_LINE);
    return sb.toString();
}

private static String printMaudeFlows(List<Node> flows, List<Node> messageFlows) {
    StringBuilder sb = new StringBuilder();
    StringJoiner str = new StringJoiner(",\n");
    sb.append(Constants.TABULATION2 + "= (" + Constants.BREAK_LINE);
    for(Node node : flows) {
        str.add(Constants.TABULATION3 + node.getMaudeString(null,false));
    }
    for(Node node : messageFlows) {
        str.add(Constants.TABULATION3 + node.getMaudeString(null,false));
    }
    sb.append(str.toString() + Constants.BREAK_LINE);
    sb.append(Constants.TABULATION2 + "  )" + Constants.MAUDE_END_LINE);
    return sb.toString();
}
```

Figura 48: Métodos printMaudeNodes y printMaudeFlows.

Una vez más se puede ver la utilidad de una jerarquía de clases bien definida. De otra forma, habría que diferenciar entre cada tipo de elemento para llamar a su método correspondiente. Aún así, vemos que es necesario en el primer método diferenciar entre las puertas, las tareas que envían mensajes y el resto de nodos. Esto se explicará más adelante ya que tiene que ver con la siguiente iteración. Con esto terminamos el Parser, capaz de generar automáticamente la notación Maude para cualquier diagrama BPMN válido.

2.2.2.6. Análisis formales implementados

Siguiendo con el flujo, una vez se tiene en el servidor la representación Maude, se realizan el resto de funciones necesarias para implementar el requisito funcional **RF - 22** que, como comentamos anteriormente, especifica que se debe proporcionar una manera para que el usuario pueda detener los análisis que están en ejecución. Para ello se realizan varios procesos que interactúan con el formulario que contiene el botón de *Stop Analysis*, figura 42. El más importante es la implementación de una **sesión**. Entendiéndose por sesión una forma de identificar cada ventana abierta de la aplicación y enlazarla con los análisis que desde ella se van a realizar, con el objetivo de que, si el usuario de esa sesión pulsa el botón de parar, se pueda ubicar de manera eficiente cuál de todos los posibles análisis en ejecución es el que se debe terminar.

Para ello, en el lado del cliente hay una función que se ejecuta en el momento en el que el usuario abre la web en un navegador nuevo o en una ventana distinta. Esta función hará una simple petición GET al endpoint **/session** que ejecuta una función en el servidor que le devolverá una cadena única que representa la sesión de esa ventana para ese usuario. Este valor se utilizará como un **token** que el usuario enviará al servidor en cada análisis en el cuerpo de la solicitud, junto con el fichero XML del proceso y el resto de parámetros. De esta forma, una vez se obtiene la salida del Parser para el diagrama de entrada, se crea un **proceso del sistema operativo** que será el encargado de ejecutar el análisis Maude, asociándose su *Process Identifier (PID)* con la sesión mediante un HashMap. Finalmente, cuando en el formulario de la figura 42 el usuario decida pulsar el botón Stop analyse, se realizará una petición POST al servidor hacia el endpoint **/stop** que contendrá un JSON con la sesión del usuario. De este JSON, como se comentó anteriormente, y gracias a las funciones de middleware aportadas por el framework Express.js, se puede obtener fácilmente el valor de la sesión accediendo al objeto

request (req en el código) como req.body.session, figura 49.

```
app.post('/stop', (req, res) => {
    var Uid = req.body.session
    var maudeProcessObject = mapUIdPId.get(Uid)
    if (maudeProcessObject != undefined) kill(maudeProcessObject.pid);
});
```

Figura 49: Post al endpoint /stop.

Finalmente, se le manda un **kill** al proceso a través de su PID, terminando la ejecución de Maude. Como se detalla más adelante, la interacción con el proceso Maude se realiza mediante eventos, de tal forma que la terminación abrupta de Maude accionará el evento asociado a su **exit** con código NULL, pudiendo entonces reconocer la parada voluntaria y devolver una respuesta acorde al usuario.

Continuamos con la parte del código del servidor que termina de implementar los requisitos **RF - 06, RF - 07, RF - 08 y RF-09**, los cuales se corresponden con el *análisis temporal, del grado de paralelismo, del model checking y del liveness*, respectivamente. Para ello son necesarios utilizar los parámetros type y formula, los cuales identifican el tipo de análisis a realizar y la proposición LTL en el caso en el que type sea model-check. En la función del servidor se hace referencia al módulo **exec-maude**, el cual es el encargado de realizar los análisis.

El análisis se apoya en la creación de un proceso del sistema operativo para lanzar Maude a través de la librería **childprocess** de Node.js. En nuestro caso, el proceso creado es un *bash* de forma que a través de la librería podemos escribir comandos que se ejecutan en background y podemos obtener sus resultados procesando los eventos *stdin.on('data', callback) y stdin.on('error', callback)* los cuales se accionarán si la salida del comando genera datos de salida o errores accionando su respectivo *callback* asociado.

El proceso es el siguiente: primero se ejecuta el comando Maude que inicia el entorno de ejecución del lenguaje y se importan las librerías que cargan las funciones necesarias para ejecutar los análisis junto con la representación maude del diagrama que se obtuvo del Parser. Después se observa el valor del campo type y en base a él se realiza un análisis u otro. Todos los comandos deben terminar con "\n", ya que lo que se está haciendo es escribir en el búffer de entrada del proceso, es decir, se está escribiendo en la terminal.

Por ello, es necesario escribir ”\n”para simular la orden de ejecutar el comando, que normalmente se realizando mediante la tecla Enter.

```
// open maude
maude.stdin.write('maude \n');
// load libraries and maude representation of the diagram
maude.stdin.write('load ../src/maude/iteration-1/bpmn.maude\n');
maude.stdin.write(maudeRepresentation + '\n')
maude.stdin.write('load ../src/maude/iteration-1/verif.maude\n');
// get command
var command;
  switch (type) {
      case 'LTL-form':
          command = 'red in CHECK : modelCheck(initSystem, ' + formula + ')';
          break;
      case 'Time':
          command = 'red in EXPLORE-EXECUTIONS : execTime(upModule(\'VERIF, false), \'initSystem.Configuration, \'St:Configuration) .'
          break;
      case 'Parallel':
          command = 'red in EXPLORE-EXECUTIONS : parDegree(upModule(\'VERIF, false), \'initSystem.Configuration, \'St:Configuration) .'
          break;
      case 'Liveness':
          command = 'search initSystem =>! < S : Simulation | tokens : NeTks:NeSet{Message}, Atts > C:Configuration .'
          break;
      default:
          command='q'; // error, close maude
    }
// execute command
maude.stdin.write(command+'\n');
```

Figura 50: Ejecución de Maude desde Node.js.

2.2.2.7. Resultados obtenidos

Una vez el análisis ha terminado, se obtiene la información de salida a través del evento mencionado anteriormente y se devuelve a la función principal del servidor, la cual crea una respuesta dentro de un JSON y se la envía de vuelta al cliente.

La información que se recibe tras realizar un análisis temporal para un diagrama dado tendrá un formato similar al siguiente:

```
Maude> rewrites: 4803 in 20ms cpu (20ms real) (240150 rewrites/second)
result Tuple{Nat,Time,Time,Time,Time}: (2,33,15,18,33/2)
```

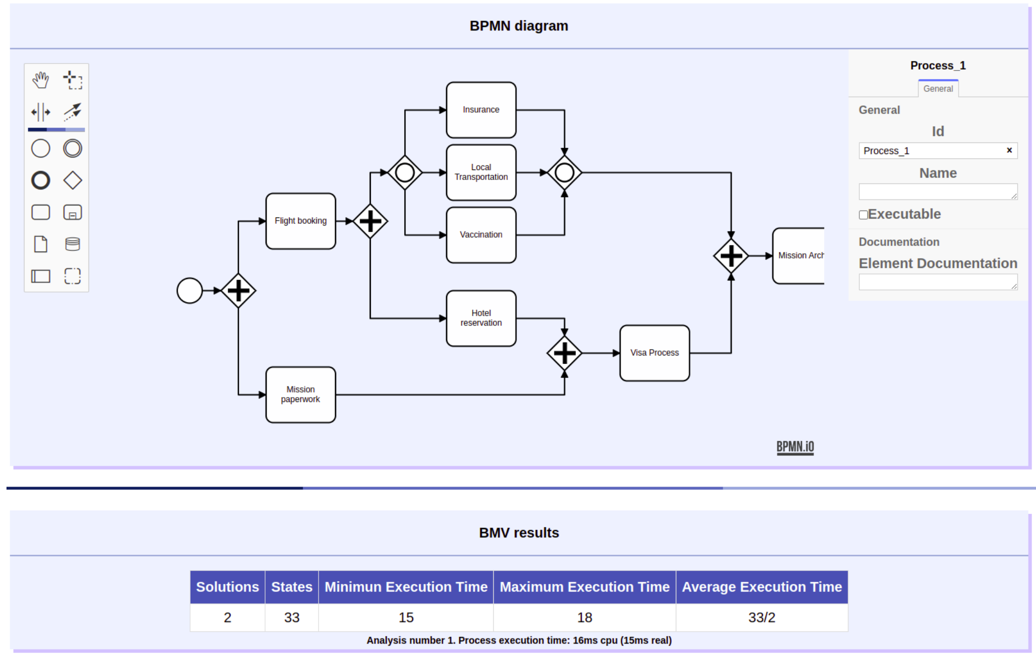
Figura 51: Salida de Madude tras realizar un análisis temporal.

Como podemos ver, es una información poco estructurada que necesita de conocimiento experto y del código fuente de las funciones que se están ejecutando para que tenga algún valor. El resultado del análisis se encuentra en la tupla final donde los dos primeros elementos hacen referencia al **número de soluciones diferentes y estados explorados**, mientras que los tres últimos hacen referencia al **tiempo mínimo, máximo y medio de ejecución** del diagrama. Por ello, se propone el requisito funcional **RF - 18**, implementado en el lado del

cliente y que se relaciona con proporcionar las salidas de los análisis de un modo legible y fácil de entender por el usuario. Destacar que en el análisis de model checking, se introduce la proposición:

$$[](tokenAt(\text{“}Missionpaperwork\text{”}) -><> tokenAt(\text{“}MissionArchive\text{”}))$$

Con ella se quiere comprobar si la tarea “Mission Archive” es siempre ejecutaba en algún punto una vez se completa la tarea “Mission Paperwork”.



Solutions	States	Minimun Execution Time	Maximum Execution Time	Average Execution Time
2	33	15	18	33/2

Analysis number 1. Process execution time: 16ms cpu (15ms real)

Figura 52: Resultado del análisis temporal realizado en la aplicación.

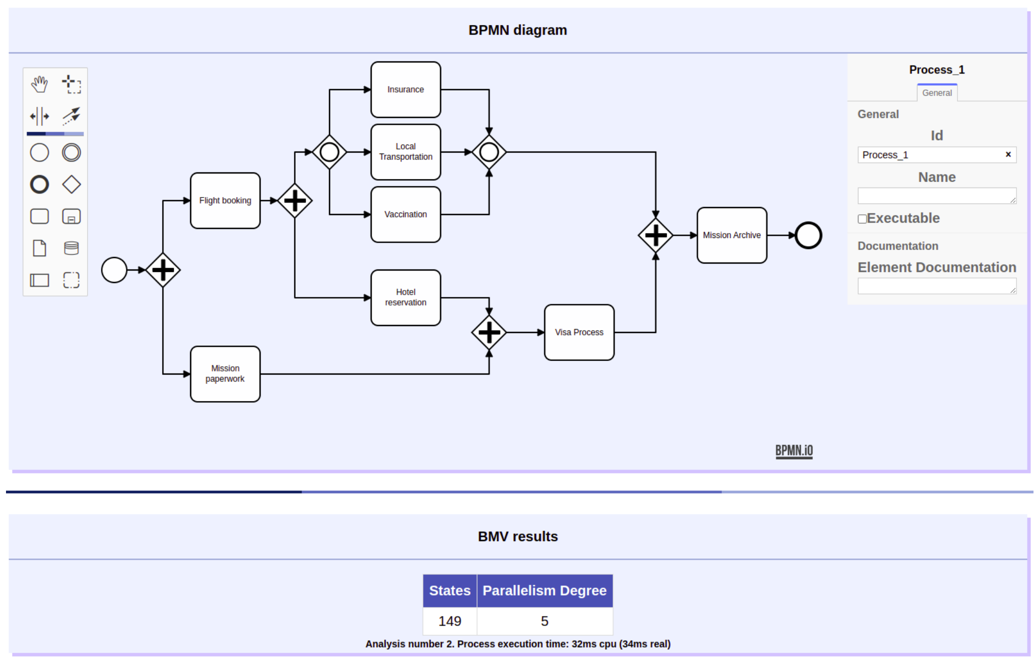


Figura 53: Resultado del análisis del grado de paralelismo realizado en la aplicación.

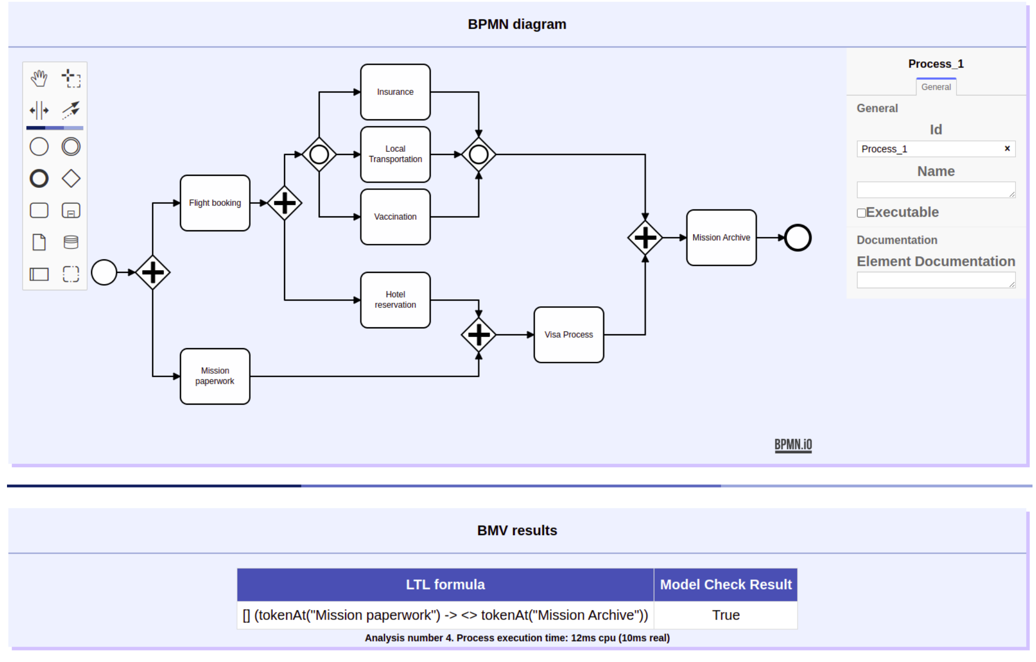


Figura 54: Resultado del análisis de model checking realizado en la aplicación.

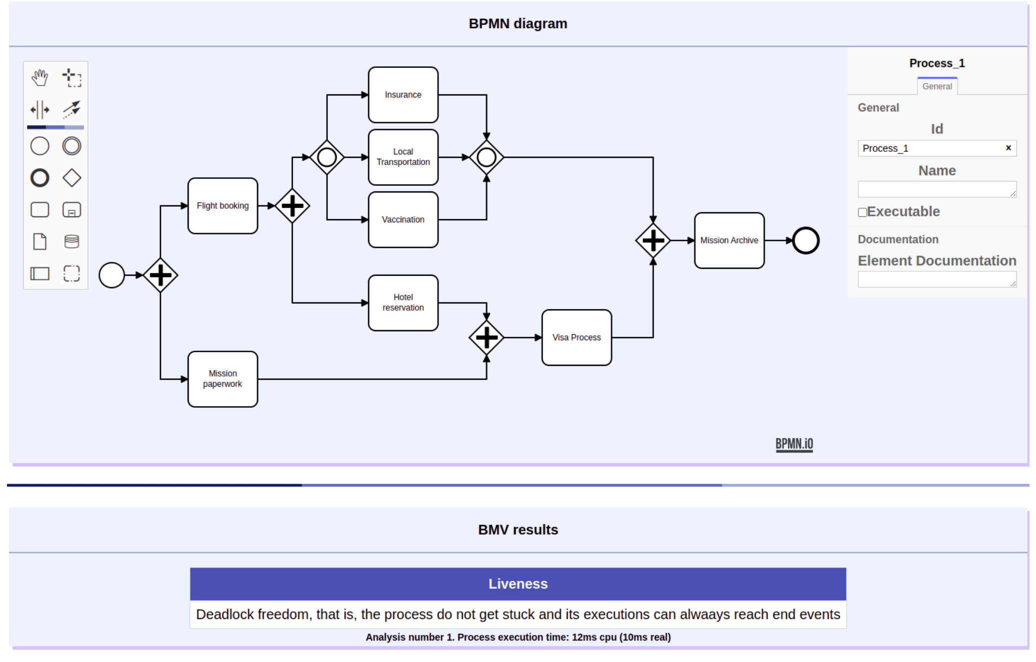


Figura 55: Resultado del análisis de liveness realizado en la aplicación (sin deadlock).

El resultado de la figura 55 muestra que el diagrama esta libre de deadlocks, lo cual es cierto ya que el proceso no tiene bucles infinitos y termina sin quedar bloqueado en ningún punto. Por ello, se muestra en la figura 56 otro análisis de la propiedad liveness sobre un diagrama que está diseñado de forma incorrecta debido a que tiene una puerta exclusiva que divide el flujo en dos ramas que posteriormente convergen en un merge paralelo. Por la semántica del merge paralelo el proceso se queda esperando a que llegue una token por cada rama, lo cual nunca ocurre (*deadlock*).

Estos son los resultados que se pueden obtener de los análisis implementados en la iteración uno. En ella se completaron todos los requisitos funcionales y no funcionales que se detallan al comienzo del apartado, construyendo una primera versión funcional de la aplicación en la que un usuario puede crear o cargar diagramas y realizar cuatro tipos de análisis formales.

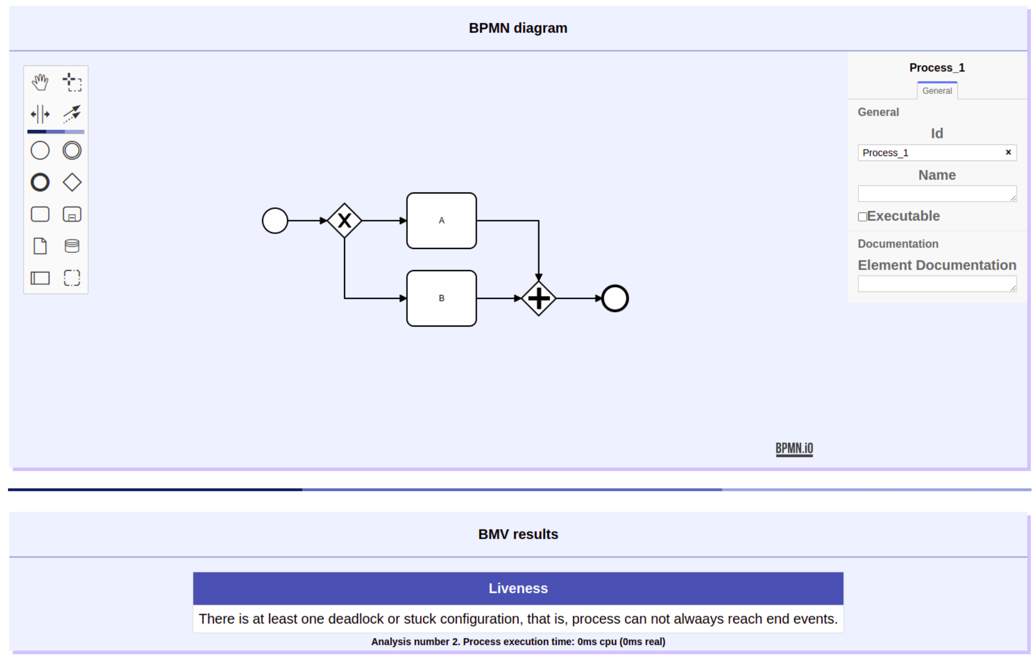


Figura 56: Resultado del análisis de liveness realizado en la aplicación (con deadlock).

2.2.2.8. Asincronismo en Node.js

En este momento es importante explicar como funciona el **asincronismo** que caracteriza a Node.js [22], ya que es una característica que no presentan muchos lenguajes o entornos. Cuando hablamos de asincronismo en programación nos referimos a que varias cosas ocurran al mismo tiempo, por ejemplo, que empiece un proceso sin que el anterior tenga que haber terminado. Por su parte la programación síncrona se refiere a todo lo contrario, ejecutando siempre un único proceso en cada momento.

JavaScript es un lenguaje de programación síncrono, sin embargo, Node.js está orientado a eventos y permite el asincronismo y la programación no bloqueante. A alto nivel, esto se consigue gracias al *event loop*, un proceso inicia Node.js al comienzo de su ejecución y que entre, otras cosas, comprueba si los eventos definidos previamente han sido disparados o si las operaciones asíncronas en proceso se han completado. En caso de ocurrir, ejecutará su respectivo *callback*, una función que modela el comportamiento esperado que debe tener lugar tras el evento o la operación en cuestión.

Por ejemplo, en este entorno de programación todas las operaciones de entrada-salida (I/O, del inglés input-output), como leer o escribir en un fichero, son asíncronas. Esto quiere decir que los mismos patrones para interaccionar con operaciones de I/O que se usan en lenguajes

como Java no sirven, sin embargo, proporciona muchas ventajas para manejarlas de manera eficiente con una sola hebra (sin contar el event loop).

En la programación tradicional, cuando una función es invocada comienza a realizar cierto procesamiento y bloquea al método principal hasta que se completa, devolviendo un resultado.

En la figura 57 se define una función que realiza la multiplicación de dos números recibidos por parámetro. Posteriormente, se llama la función para obtener el resultado de la operación y, finalmente, se imprime en pantalla el resultado.

```
function multiplicar(x, y) {
    return x * y;
}
const resultado = multiplicar(3, 2);
console.log(resultado);
```

Figura 57: Función simple que calcula la multiplicación de dos números.

Si por ejemplo se utiliza este estilo de programación en un **lenguaje bloqueante** para una operación de entrada-salida, el código principal esperaría a que se realice la operación y posteriormente devolvería el resultado. Sin embargo, en Node.js estas operaciones son asíncronas por lo que se debe utilizar el estilo de programación Continuation Passing Style (CPS) que se basa en utilizar *callbacks* para continuar el flujo de trabajo.

La figura 58 muestra cómo se puede obtener el mismo resultado que antes pero usando este nuevo patrón. Simplemente se le añade un parámetro a la función multiplicar. Ese parámetro es el callback, que no es otra cosa que una función en forma de lambda que recibe el resultado de la operación y lo imprime en pantalla.

```
function multiplicar(x, y, callback) {
    callback(x * y);
}
multiplicar(3, 2, result => console.log(result));
```

Figura 58: Función con callback que calcula la multiplicación de dos números.

En este caso la función no realiza ninguna operación asíncrona y el callback es ejecutado al instante, por ello este estilo solo tiene sentido cuando se usan operaciones asíncronas. En la figura 59 se utiliza a modo de ejemplo una función asíncrona denominada *setTimeOut* que

recibe un número de milisegundos tras los cuales debe ejecutar una función. Como se ha comentado, al ejecutar la función se define el evento pero Node.js seguirá procesando el resto del código hasta que este se dispare (en este caso al agotarse el tiempo).

```
function multiplicar(x, y, callback) {
    setTimeout(() => callback(x*y), 1000);
}
multiplicar(3, 2, result => console.log(result));
console.log("Este mensaje se mostrará en consola antes que el resultado de la multiplicacion")
```

Figura 59: Función con operación asíncrona y callback.

La figura 60 muestra la salida del programa, donde se puede ver como Node.js ejecuta el resto del código a la espera de que se dispare este evento, que puede ser cualquier tipo de evento o de operación entrada salida.

```
Este mensaje se mostrara en consola antes que el resultado de la multiplicacion
6
```

Figura 60: Salida de la ejecución del programa de la figura 57.

Este comportamiento es muy útil pero puede llevarnos a tener que anidar números callbacks, lo que se conoce como callback hell, patrón que se muestra en la figura 61.

```
//Ejemplo de Callback Hell
a(function(resultA){
    b(resultA,function(resultB){
        c(resultB, function(resultC){
            d(resultC, function(resultD){
                e(resultD, function(resultE){
                    f(resultE, function(resultF){
                        g(resultF, function(resultG){
                            console.log(resultG)
                        })
                    })
                })
            })
        })
    })
})
```

Figura 61: Ejemplo de callback hell.

Para evitarlo, se crearon en versiones posteriores de Node.js las **funciones asíncronas, los bloques awaits y las promesas** [30], construcciones ampliamente usadas en el desarrollo de la aplicación. En la documentación oficial, se definen las *promesas* como un tipo de objeto

que representa el resultado de una operación asíncrona y que pueda estar disponible inmediatamente, en el futuro o nunca. Reciben como parámetro una función denominada *ejecutor* que a vez tiene dos parámetros: resolve y reject, los cuales resuelven la promesa de forma correcta o incorrecta, respectivamente. Además, tienen un método llamado *then()* con el que se puede acceder al resultado de resolve y otro llamado *catch()* se accede a lo devuelto por reject.

La figura 62 muestra cómo se puede programar el ejemplo anterior utilizando promesas, el cual también produce el mismo resultado. Estas nos aportan una mejor forma de controlar los resultados que producen las operaciones asíncronas. Además, los métodos then y catch, también devuelven promesas, haciendo que se puedan encadenar. Por otra parte, esto podría conducir a códigos similares al callback hell. Normalmente, los códigos que nos llevan a este antipatrón son fruto de tener que encadenar resultados de operaciones asíncronas. Una forma de evitarlo es el uso del operador *await* el cual solo puede ser usado dentro de una *función asíncrona*. Esta función se ejecutará de forma asíncrona dejando a Node.js seguir procesando el código fuera de ella pero haciendo que los awaits definidos dentro esperen el resultado de las promesas sin dejar que el código avance. Pudiendo capturar los errores de las promesas con un *try-catch*, figura 63.

```
function multiplicar(x, y) {
    return new Promise((resolve,reject) => (setTimeout(() => resolve(x*y), 1000)))
}
multiplicar(3, 2).then(res => console.log(res))
console.log("Este mensaje se mostrará en consola antes que el resultado de la multiplicacion")
```

Figura 62: Función con operación asíncrona y promesas.

```
async function multiplicar(x, y) {
    let res = await new Promise((resolve,reject) => (setTimeout(() => resolve(x*y), 1000)))
    console.log(res)
}
multiplicar(3, 2)
console.log("Este mensaje se mostrará en consola antes que el resultado de la multiplicacion")
```

Figura 63: Función asíncrona con await.

Las modificaciones hechas en el código hacen que no sea necesario utilizar el método then() si no que simplemente se puede imprimir en pantalla el valor, el cual será impreso un segundo más tarde sin bloquear Node.js. En el servidor, se utiliza de esta misma forma una función asíncrona que realiza dos operaciones de este tipo: la llamada al *Parser* (que utiliza I/O) y

el análisis Maude (que crea un proceso del sistema operativo en background y espera a su resultado). En ambas se utiliza el operador await para esperar por el resultado para no tener que mezclar el resto del código con distintas llamadas al método then() ni tampoco anidar callbacks. Además, todo ello se encuentra dentro de una sentencia try-catch que captura los distintos errores que pueden surgir en el proceso.

2.3. Iteración dos

2.3.1. Análisis de requisitos

El objetivo de esta iteración es añadir valor a la aplicación integrando análisis más complejos. Como se ha comentado, estos análisis se basan en las técnicas de los tres últimos artículos contemplados. Estos están centrados en realizar *múltiples simulaciones concurrentes* de un proceso que compiten por una serie de *recursos compartidos*. Además, se pueden analizar casos de uso más reales que pueden surgir en distintos procesos de negocio gracias a los nuevos elementos que se añaden al subconjunto BPMN con el que se trata (eventos, mensajes, temporizadores...). Para conseguir esto, se deben añadir en el cliente los componentes necesarios para realizar cada análisis e integrarlos con las respectivas funciones del servidor, en el cual se implementarán las nuevas técnicas para poder ejecutarlos a partir de la representación Maude dada por el *Parser*, que debe ser extendido acorde a los nuevos elementos contemplados.

Los requisitos a implementados en esta iteración se encuentran listados en la siguiente página. Los análisis a realizar se corresponden con los requisitos funcionales **RF - 10 y RF - 11**, de los cuales dependen todos los requisitos desde el **RF - 12** hasta el **RF - 17**. Con el primero **(RF - 10)** se pretende usar toda la información que se puede extraer de los análisis de recursos (AET, AST, GTU, Average GTU y UP) para generar unas *gráficas* que presenten los resultados de forma visual, aportando aún más valor. El segundo **(RF - 11)**, pretende implementar por completo el algoritmo ávido para obtener la combinación óptima (localmente) de recursos. Por último, el resto de requisitos mencionados **(RF - 12, ..., RF - 17)** se corresponden con la implementación de los componentes funcionales que se utilizan para solicitar parámetros al usuario, en relación con cada análisis.

Requisitos funcionales

Identificador	Nombre	Información	Dependencias
RF - 05	Analizar diagrama	Se podrán realizar análisis formales sobre cada uno de los diagramas.	RNF - 02
RF - 10	Análisis de recursos	Se obtendrá diversa información sobre los recursos usados por el diagrama como el tiempo de uso, el tiempo medio de ejecución para un número de dado de simulaciones concurrentes, el tiempo de bloqueo en las actividades y en las puertas de eventos, etc. Esta información será procesada para crear distintas gráficas dinámicas que se mostrarán en la web.	RF - 05 RNF - 03 RNF - 04
RF - 11	Optimización de recursos	La información de los recursos discutida en el requisito de arriba (RF - 10) se utilizará para calcular una combinación de recursos óptima (localmente) en función de su coste y tiempo de ejecución.	RF - 05 RNF - 03 RNF - 04
RF - 12	Número de recursos	Se le solicitará al usuario el número de recursos que utiliza en su diagrama	RF - 09 RF - 10
RF - 13	Número de simulaciones	Se le solicitará al usuario el número de simulaciones concurrentes que quiere ejecutar para el análisis en cuestión.	RF - 09 RF - 10
RF - 14	Nombre de recursos	Se le solicitará al usuario el nombre de cada recurso.	RF - 09 RF - 10
RF - 15	Número de instancias	Se le solicitará al usuario el número de instancias de cada recurso	RF - 09
RF - 16	Rango de instancias	Se le solicitará al usuario el número mínimo y máximo de instancias de cada recurso que quiere que se tengan en cuenta en el análisis.	RF - 10
RF - 17	Coste	Se le solicitará al usuario el coste por hora que le supone cada recurso.	RF - 10
RF - 18	Procesar salidas	Todos y cada uno de los análisis devuelven información no estructurada que requiere conocimiento experto para ser de utilidad. Por ende, se proporcionará un mecanismo para dar formato a las salidas y hacer que sean útiles para cualquier usuario.	RF - 05

Identificador	Nombre	Información	Dependencias
RF - 19	Parser	Los diagramas BPMN están representados internamente por documentos XML, pero los análisis formales desarrollados en Maude necesitan de una especificación determinada de los procesos para funcionar. Por tanto se desarrollará un algoritmo que permitirá generar una representación Maude correcta a partir de todo diagrama BPMN válido.	-
RF - 20	Informar a los usuarios	Se implementarán diferentes botones de ayuda que mostrarán información a los usuarios de la web para explicarles cómo pueden realizar cada uno de los análisis de forma correcta.	RF - 05
RF - 21	Página de bienvenida	Se dedicará una espacio del sitio web para explicar de manera general que es y como se puede utilizar la web, proporcionando referencias a los papers publicados por mi tutor y su equipo que tienen relación con la página para que cualquier usuario interesado pueda tener más información sobre cada tipo de análisis.	-
RF - 22	Parar análisis	Se debe permitir que los usuarios puedan parar un análisis que se encuentre en ejecución de forma sencilla y rápida	RF - 05 RNF - 03 RNF - 04
RF - 23	Mostrar tiempo	Para los análisis no inmediatos (duración superior a 2s) se mostrará un cronómetro dinámico que le vaya indicando al usuario cuánto tiempo lleva realizándose el proceso.	RF - 05 RNF - 03 RNF - 04
RF - 24	Añadir información adicional	Se debe permitir que el usuario añada a los diagramas la información adicional necesaria para realizar los análisis de forma fácil e intuitiva.	-

Los requisitos marcados con (*) indican que ya fueron implementados en la iteración anterior y que su funcionalidad solo va a ser extendida en esta.

Requisitos no funcionales

Identificador	Nombre	Información	Dependencias
RNF - 01	Cumplir con el estándar	Todos y cada uno de los diagramas BPMN usados en la aplicación deberán cumplir con el estándar 2.0 de BPMN.	RF - 01 RF - 02 RF - 03 RF - 04
RNF - 02	Diagramas válidos	Se deberá informar al usuario acerca de si su diagrama está correctamente diseñado antes de ser analizado.	RF-05
RNF - 03	Mover formularios	Se deben poder arrastrar los formularios para facilitar el escribir sobre ellos al mismo tiempo que se puede observar el diagrama, de forma que la utilización de la página sea más cómoda para el usuario.	
RNF - 04	Cerrar formularios	Se deben poder cerrar todos los formularios pulsando un botón sin necesidad de realizar el análisis.	

2.3.2. Implementación

2.3.2.1. Cliente web

Se implementan los componentes relacionados con el requisito **RF - 10** (gráficas con la información de los análisis de recursos). Para ello se hizo que en el momento en el que el usuario seleccionara la opción referente al análisis de recursos (figura 39 - Menú de análisis) se abriera un formulario que le solicitara el número de recursos que hay en su diagrama junto con el número de simulaciones concurrentes **(RF - 12 y RF - 13)**. El formulario se creó con un diseño similar a los anteriores y cumpliendo con los requisitos no funcionales que exigen que se pueda mover y cerrar. En la figura 64 se puede ver una imagen del formulario. Cabe destacar que mediante etiquetas básicas de HTML se asegura que el formulario sólo acepte valores coherentes, en este caso números.

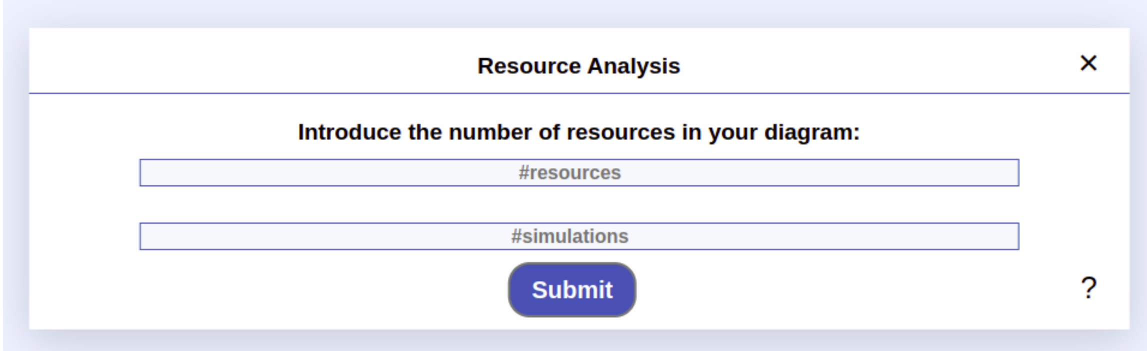


Figura 64: Formulario (1 / 2) del análisis de recursos.

Una vez el usuario especifica los valores y presiona el campo **Submit**, se genera la segunda parte del formulario que le solicita el resto de información necesaria. Concretamente, habrá un campo para escribir el nombre y el número de instancias de cada recurso **(RF - 14 y RF - 15)**. En la figura 65 se muestra esta parte del formulario, que presenta esos campos en particular porque en el anterior se especificó un número de dos recursos.

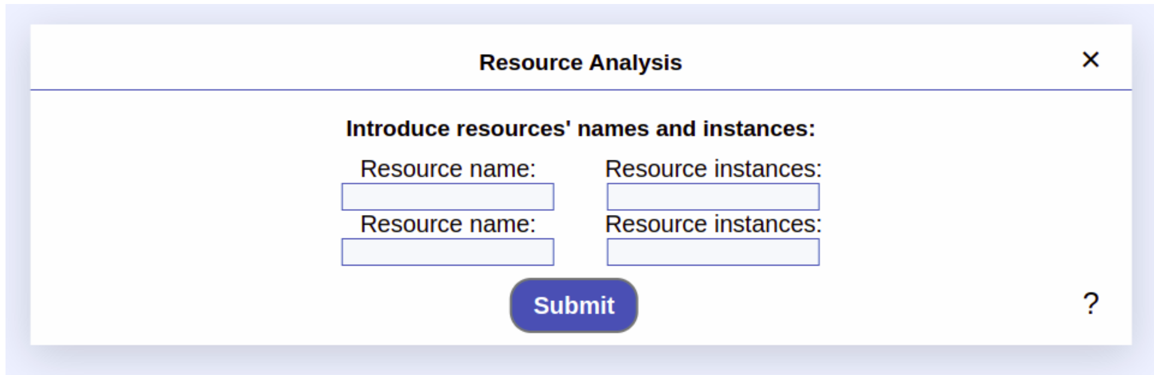


Figura 65: Formulario (2 / 2) del análisis de recursos.

Para el requisito **RF-11**, relacionado con el análisis de optimización de recursos, se crearon otros formularios muy similares. El primero solicita el número de recursos y de simulaciones **(RF - 12 y RF - 13)** y el segundo solicita la información de entrada necesaria para el algoritmo ávido: nombre, instancias y el coste por horas para cada recurso **(RF - 14, RF - 16 y RF - 17)**. Las figuras 65 y 66 muestran cómo se verían, también para dos recursos.

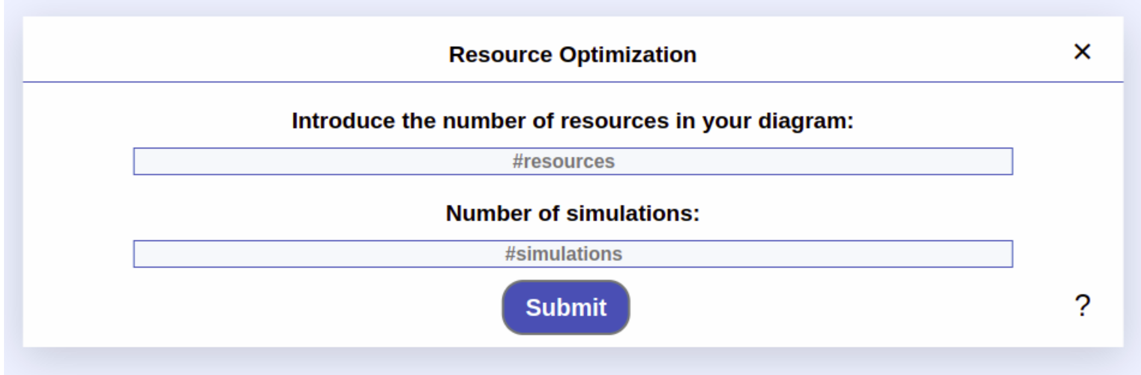


Figura 66: Formulario (1 / 2) del análisis de optimización de recursos.

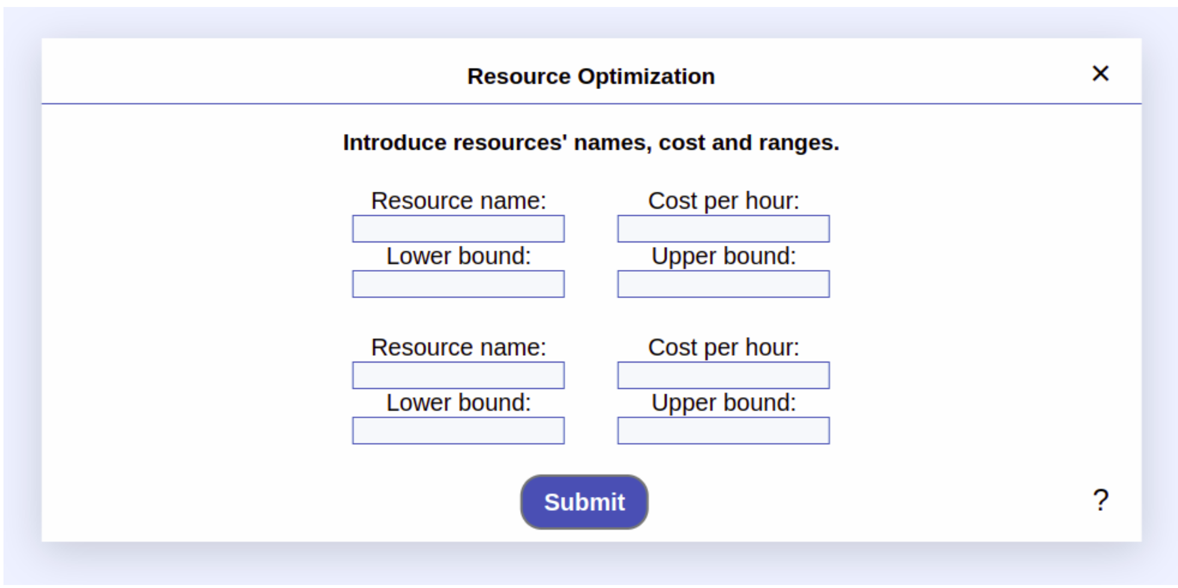


Figura 67: Formulario (2 / 2) del análisis de optimización de recursos.

Posteriormente, para ambos formularios se creó una función que recogiera la información del usuario y la escribiera en el JSON que se le pasaría después al servidor. Es importante destacar que se **asume** que el usuario está escribiendo el mismo número de recursos definidos en el diagrama a través del panel y con el mismo nombre, de lo contrario el análisis fallará.

El último desarrollo en el lado del cliente tiene que ver con la implementación del requisito **RF - 20**, con el cual se establece que se mostrarán información a los usuarios de la web para explicarles cómo pueden realizar cada uno de los análisis de forma correcta. Para ello, se diseñó un botón de ayuda en el menú de análisis que al pulsar sobre él activa un *pop-up* que informa al usuario sobre cómo mostrar la información referente a cada análisis.

En este caso se hace manteniendo un segundo y medio el ratón sobre el botón que activa cada uno. Se puede ver en las figuras 67 y 68 la información para el botón del menú y para el caso del análisis temporal. Las ayudas para el resto de análisis no se muestran ya que son similares.

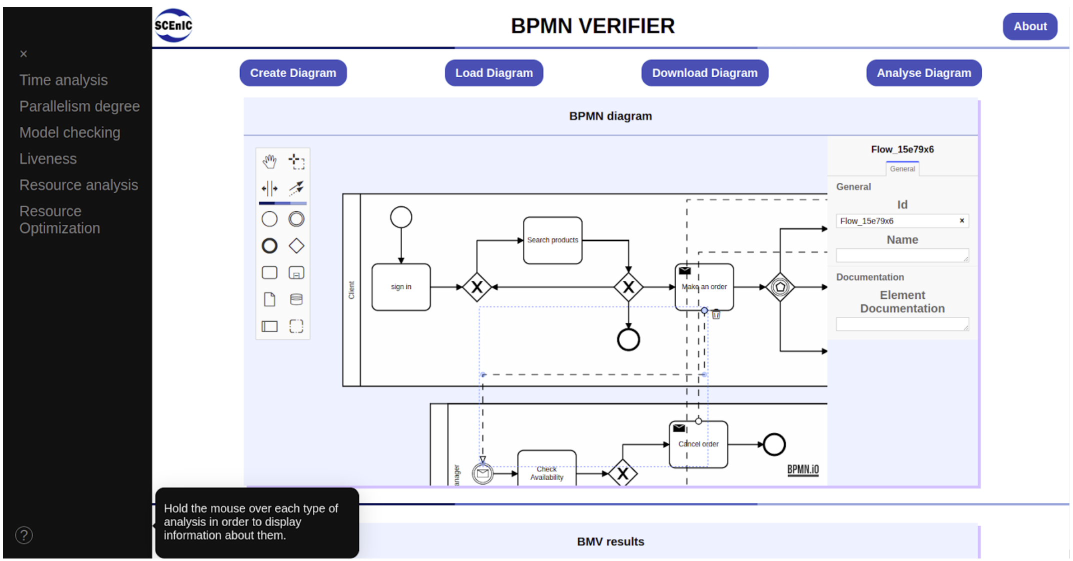


Figura 68: Información de ayuda del botón del menú.

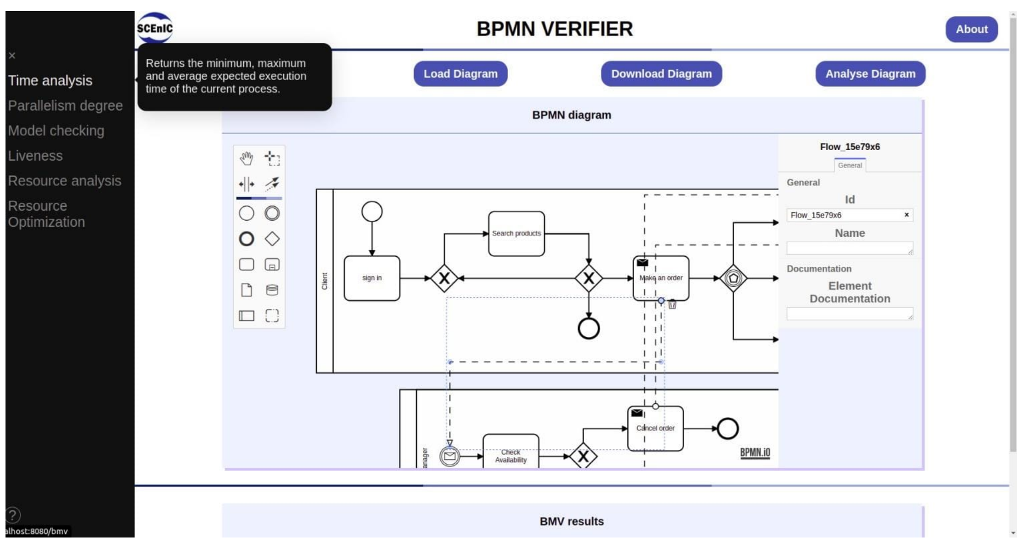


Figura 69: Información de ayuda del análisis temporal.

Finalmente, se implementó un botón en la esquina inferior derecha de cada uno de los formularios. Estos accionan un *pop-up* en el que se informa acerca de qué información se tiene que añadir y cómo: la figura 70 muestra la información de ayuda para el formulario del model checking, la figura 71 la muestra para el que acciona el cronómetro y el mecanismo de parada, la figura 72 para el del análisis de recursos y por último la figura 73 para el de optimización de recursos.

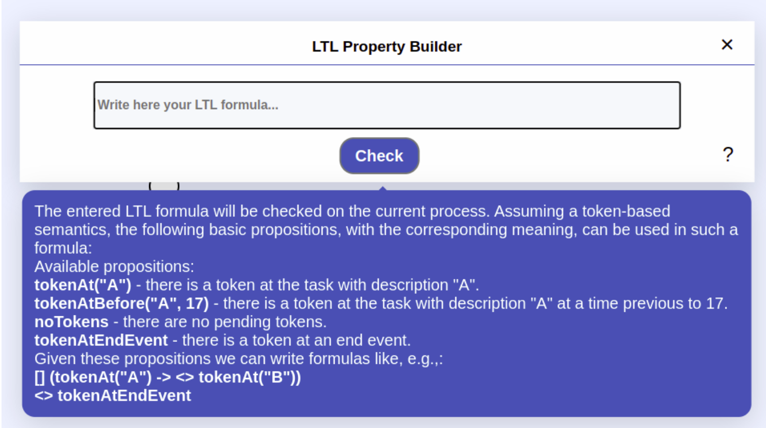


Figura 70: Información de ayuda en el formulario del model checking.

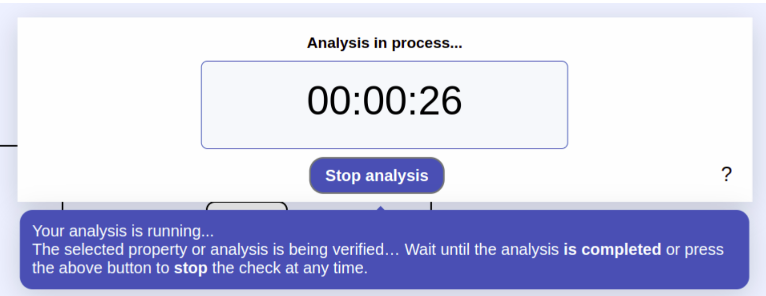


Figura 71: Información de ayuda en el formulario del cronómetro.

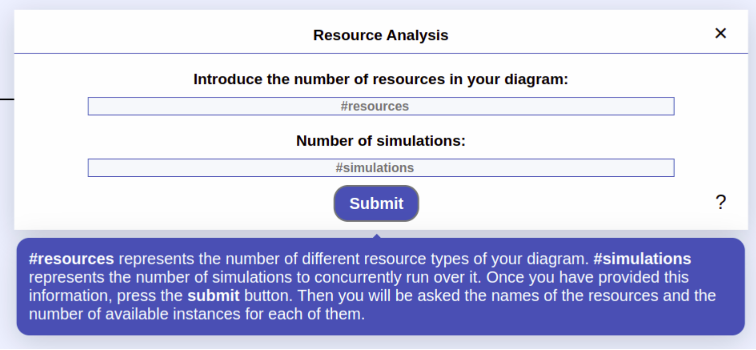


Figura 72: Información de ayuda en el formulario del análisis de recursos.

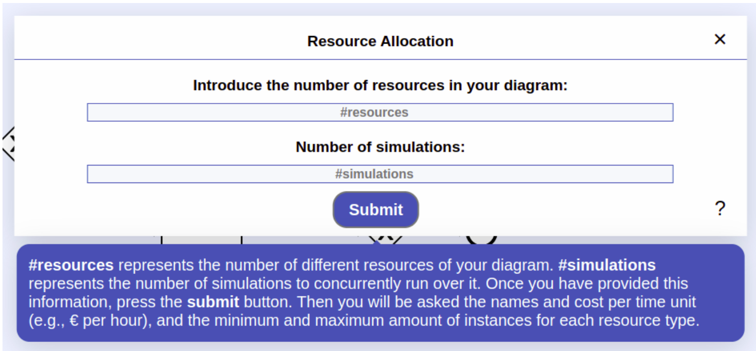


Figura 73: Información de ayuda en el formulario del análisis de optimización.

2.3.2.2. Parser

Siguiendo el flujo del diagrama de secuencia (figura 23), una vez la información del análisis a realizar llega al servidor, lo primero que se hace es llamar al *Paser* para obtener la representación Maude del diagrama objeto del análisis, de igual manera a como se hacía en la iteración anterior. Sin embargo, ahora tenemos información adicional que se debe procesar de distinta manera, por lo que se tiene que extender tanto el conjunto de elementos BPMN con el que interactúa el Parser como la información que genera. De hecho, no solo se genera más información sino que algunos aspectos de Maude son distintos, por lo que se debe diferenciar la notación a la que pertenece el diagrama a analizar obligatoriamente dentro del código. Esto se hace con una variable booleana dentro de la clase encargada de leer el fichero (NodeHandler) a la que acceden el resto. Se puede diferenciar fácilmente la notación que debe generarse atendiendo a los elementos del fichero XML, ya que si contiene eventos o recursos pertenece obligatoriamente a la nueva.

Como se explicó previamente, la primera clase en interactuar con el diagrama es **NodeHandler**, que aplica *SAX* para leerlo y extraer la información. En ella simplemente se añaden tres estructuras de datos. Dos listas para almacenar el conjunto de *mensajes y de eventos* y un conjunto para almacenar el conjunto de los *identificadores de los recursos.* Estos elementos se han separado del conjunto de nodos original para hacer ciertas operaciones más eficientes, como se verá un poco más adelante.

También se realizan diversos cambios en la clase **Factory**, encargada de crear los objetos que referencian a cada tipo de nodo. Primero, se añaden los nuevos elementos: *puerta basada en eventos, tareas que envían mensaje, eventos (de mensaje y de temporizadores), y los propios*

mensajes. En esta clase se valida la información que el usuario introduce a través de los paneles Time y Resource sobre las tareas y los flujos. Para el tiempo antes era sencillo porque se trataba de un número pero ahora pueden haber expresiones estocásticas para las distribuciones normales, uniformes y exponenciales. En cuanto a los recursos que usa cada tarea, se deben especificar como una lista separada por comas donde cada elemento será el nombre del recurso y el número de instancias requeridas separadas por dos puntos. Por ejemplo, una tarea que necesita tres drones y un empleado debe tener escrito en su campo *Resource*: "drone:3, employee:1". Para la validación, se crean tres funciones (figura 74) para ejecutar **expresiones regulares** y comprobar si el valor introducido por el usuario es una de las expresiones correctas. Las dos primeras definen las expresiones regulares para controlar el valor de los campos y la última se utiliza para simplificar el proceso de ejecutar una expresión regular sobre una cadena, ambas pasadas por argumento.

```
private static boolean parseTimeAtributte(String expression) {
    return parseRegex("Norm\\(\\d+\\.\\d+\\,\\d+\\.\\d+\\)",expression) || //Norm
           parseRegex("Unif\\(\\d+\\.\\d+\\,\\d+\\.\\d+\\)",expression) || //Unif
           parseRegex("exp\\(\\d+\\)",expression)                       || //exp
           parseRegex("\\d+(\\.\\d+)?",expression);                        //Number (double or int)
}

private static boolean parseResouceAttribute(String resourcesAttribute) {
    Pattern pat = Pattern.compile("([a-zA-Z]+):\\d+(,[a-zA-Z]+:\\d+)*");
    Matcher mat = pat.matcher(resourcesAttribute);
    if(mat.matches()) {
        NodeHandler3.addResourceId(mat.group(1));
        return true;
    }else throw new BpmnProcessException("Atributo 'resource' no valido");
}

private static boolean parseRegex(String regularExpression,String string) {
     Pattern pat = Pattern.compile(regularExpression);
     Matcher mat = pat.matcher(string);
     return mat.matches();
}
```

Figura 74: Validación con expresiones regulares.

La función que valida la información de los recursos, en caso de no fallar, obtiene el nombre del recurso con la definición de un *group* (partes de la expresión regular escritas entre paréntesis) y lo introduce en el conjunto de identificadores de recursos para utilizarlos más adelante. El resto del campo Resource, es almacenado por la tarea que lo usará a la hora de escribir su información.

Tras estos cambios se lee el contenido del XML referente al diagrama y se obtiene como resultado la lista de nodos, de flujos, de eventos y de mensajes. Entonces se ejecuta la función *writeMaudeString* de la clase **Java2Maude** que devuelve el String con la notación Maude.

En este momento se detectó un inconveniente que dificultará la traducción de los objetos almacenados en la notación Maude, debido a como esta última se había diseñado. En el paquete **Elements** del Parser, se almacena la jerarquía de clases que representan a los elementos BPMN (véanse los diagramas UML de la figuras 20 y 21). Se decidió que esto era conveniente para poder almacenar las características de cada elemento y poder usar la superclase abstracta Node para llamar a los métodos abstractos implementados por cada uno. Por ello, existe una clase llamada **IntermediateCatchEvent**, la cual hereda de evento y sirve para representar tanto a los eventos de mensajes como a los de temporizadores, ya que son realmente similares. Todas las clases del paquete Elements excepto esta definen su método *getMaudeString()* para generar la representación Maude del nodo. Esto se debe a que como se comentó, la notación Maude está definida de tal forma que los eventos de mensajes y de temporizadores no se definen en el conjunto de nodos al igual que el resto de elementos, **sino que se modelan como información adicional que se debe almacenar en su flujo de entrada**, el cual estará bloqueado hasta que llegue el mensaje o se agote el temporizador. El problema reside en que tal y como están definidos los elementos en el XML, introducir la información de los eventos en su flujo de entrada no es algo inmediato, por lo que para ello se desarrolló un algoritmo que se describe a continuación:

1. Se recorre la lista de eventos, que se definió separada del resto de nodos para agilizar este proceso.

2. Los eventos de mensaje normalmente tienen un flujo de entrada, otro de salida y un flujo de mensaje entrante que representa el mensaje que los activará. Se debe acceder al atributo que referencia a su flujo de entrada, ya que algunos eventos tienen y otros no, actuando de forma diferente en cada caso.

    a) Si el evento no tiene flujo de entrada, estamos ante el caso particular de un **evento (de mensaje o tiempo) que inicia un flujo** dentro de un *Lane*. Se debe iterar sobre la lista de flujos de mensaje, separada del resto de flujos para agilizar este proceso, para encontrar aquel mensaje que tiene como *target* el evento en cuestión (recordar que los temporizadores también se inician con la recepción de un mensaje). Entonces, se le asigna al flujo de mensaje el identificador del flujo de salida del evento, de forma que sea este el que mantenga la información del evento.

b) En el otro caso, tenemos un evento intermedio común, para ello se debe recorrer la lista de flujos para encontrar aquel que tiene como *target* el evento, es decir, para encontrar su flujo de entrada. Entonces, en este flujo **(F1)** se actualiza una variable que indica que almacenará información de un evento (mensaje o timer), necesaria para que el flujo escriba la información correspondiente en el método getMaudeString(). Después se debe recorrer la lista de nodos, para encontrar aquel que tiene como flujo de entrada **(F2)**, el que flujo de salida del evento, para intercambiar su flujo de entrada **(F2)** por el del evento **(F1)**, haciendo que se conecte el flujo que entraba al evento con el nodo al que este apuntaba.

De esta forma, se consigue el comportamiento esperado, el cual es hacer que el flujo de entrada del evento se bloquee hasta recibir el mensaje de activación, para continuar con el flujo que salía del evento una vez lo reciba o el contador llegue a cero. El caso 2.a es más sencillo, ya que solo se debe hacer que el flujo que almacena la información apunte al que salía del evento. En el 2.b, además de esto, se debe cambiar la información del flujo de entrada del nodo al que llegaba.

Para ello se definen cuatro métodos, la principal se llama **parseEventsInformation** y se puede ver en la figura 75. Esta se encarga de realizar el proceso, diferenciando entre cada uno de los casos (a y b) y actuando en consecuencia usando tres métodos auxiliares que ayudan en el proceso, figura 76.

```java
public static void parseEventsInformation(List<Node> nodes, List<Node> flows, List<Node> messageFlows,
        List<Node> events) {
    for(Node eventNode : events) {
        IntermediateCatchEvent event = (IntermediateCatchEvent) eventNode;
        if(event.getIncoming().size()>0) { //Normal Event
            //Update event incomingFlow to keep event information
            SequenceFlow flow = (SequenceFlow) getNodeById(flows,event.getIncoming().get(0));
            if(flow != null) {
                if(event.getEventType().equals("message")) flow.setHasMessage(true);
                else flow.setTimeOutDuration(event.getTimeOutDuration());
            }
            //Remove event from the schema as if it was a linked list, so that event incomingFlow points now
            //to eventoutgoingNode
            Node node = getNodeToUpdate(nodes, event.getOutgoing().get(0));
            if(node != null) {
                List<String> incomingFlows = node.getIncomingFlows();
                incomingFlows.remove(event.getOutgoing().get(0));
                incomingFlows.add(event.getIncoming().get(0));
            }
        }else {//Start Event
            MessageFlow messageFlow = getMflowByIncomingOrOutgoing(messageFlows, event.getId(),"outgoing");
            if(messageFlow != null) {
                messageFlow.setEventRelatedNode(event.getOutgoing().get(0));
                event.setIncoming(Arrays.asList(messageFlow.getId())); //We set the list of incoming flows to the
                //start event node so we can handle it in the messageFlowExchange method
            }
        }
    }
    //Add events message information
    for(Node node : nodes) {
        if(node instanceof SendTask) messageFlowExchange((SendTask) node, messageFlows,events);
    }
}
```

Figura 75: Método principal del algoritmo, parseEventsInformation.

```java
/**
 * Given a SendTask and the set of messageFlows, we set the message of the task to the incomingFlow of the event.
 * It is assumed that messageFlow is connected to an event.
 * @param sendTask
 * @param messageFlows
 * @param events
 */
private static void messageFlowExchange(SendTask sendTask, List<Node> messageFlows, List<Node> events) {
    MessageFlow messageFlow = getMflowByIncomingOrOutgoing(messageFlows, sendTask.getId(), "incoming");
    if(messageFlow != null && messageFlow.getOutgoing().get(0).startsWith("Event")) {
        Node node = getNodeById(events,messageFlow.getOutgoing().get(0));
        if(node != null) sendTask.setMessage("m"+node.getIncomingFlows().get(0));
    }
}


private static Node getNodeById(List <Node> nodes, String nodeId) {
    for(Node node : nodes) {
        if(node.getId().equals(nodeId)) return node;
    }
    return null;
}

private static MessageFlow getMflowByIncomingOrOutgoing(List <Node> nodes, String id,String getBy) {
    for(Node node : nodes) {
        if(node instanceof MessageFlow) {
            MessageFlow mf = (MessageFlow) node;
            if(getBy.equals("outgoing") && mf.getOutgoing().get(0).equals(id) ||
               getBy.equals("incoming") && mf.getIncoming().get(0).equals(id)) return mf;
        }
    }
    return null;
}

private static Node getNodeToUpdate(List<Node> nodes, String id) {
    for(Node node : nodes) {
        List<String> incomingFlows = node.getIncomingFlows();
        for(String incomingFlow : incomingFlows) { //usually is going to be only one
            //Once we get the node which is connected to the event, we change its flow with the incoming flow of
            //the event
            if(incomingFlow.equals(id)) return node;
        }
    }
    return null;
}
```

Figura 76: Métodos auxiliares usados en el algoritmo.

La primera de las funciones auxiliares, **messageFlowExchange**, se encarga de añadir la información del evento al flujo de entrada del mismo. Para ello, accede al flujo de mensaje de activación del evento usando la función **getMFlowByIncomingOrOutoing()** definida de esa forma para poder ser utilizada en la función principal y buscar por el atributo de entrada o salida de los nodos. Después se accede al nodo a través de la función **getNodeById()**, reutilizada también en la principal para realizar el intercambio. Por último se usa **getNodeToUpdate()** para buscar en la función principal el nodo al que apunta el flujo de salida del evento.

Destacar que este proceso es complicado de entender y disminuye un poco la eficiencia del parser debido a que se realizan recorridos sobre listas que se podrían evitar con diseño de la notación más acorde en este aspecto. Por ello, se barajó la posibilidad de cambiar toda la notación Maude para representar los eventos como nodos dentro de su conjunto. Sin embargo, esto suponía cambiar todos los algoritmos que realizaban los análisis, además de las reglas semánticas y aquellos que controlaban las simulaciones. Por lo que se decidió dejarlo para trabajos futuros dado el tiempo que supondría familiarizarse con Maude lo suficiente como para poder llevar a cabo con éxito dicha tarea.

Tras realizar este preprocesado para construir los flujos que modelan eventos, se puede generar finalmente la nueva notación usando el método *writeMaudeString()* de la clase **JavaToMaude**, figura 77. Al ser tan diferente a la primera, se utiliza la variable booleana de la clase **NodeHandler** que indica que el diagrama pertenece a la nueva representación para diferenciar entre las construcciones que se deben generar.

```
/**
 * Writes in String the maude specification of the BPMN process
 * @param nodes
 * @param flows
 * @param messageFlows
 * @param isExtended
 */
public static String writeMaudeToString(List<Node> nodes, List<Node> flows,List<Node> messageFlows,
        boolean isExtended){
    StringBuilder sb = new StringBuilder();
    if(isExtended) sb.append(Constants.MAUDE_HEADER_2);
    else sb.append(Constants.MAUDE_HEADER_1);
    sb.append(Constants.TABULATION + "ops " + buildStringFromNodes(nodes) + ": -> NId" + Constants.MAUDE_END_LINE);
    sb.append(Constants.TABULATION + "ops " + buildStringFromNodes(flows) + ": -> FId" + Constants.MAUDE_END_LINE);
    if(isExtended) {
        sb.append(buildStringFromMessageFlow(flows,messageFlows));
        sb.append(printResourceIds());
    }
    sb.append(Constants.MAUDE_OP);
    if(isExtended) sb.append(Constants.MAUDE_OP_RESOURCES);
    if(initial == null) {
        throw new BpmnProcessException("BPMN process does not have start event.");
    }else {
        if(!isExtended && initialFlow != null)sb.append(Constants.TABULATION + "eq init = token(" +
    initialFlow.getId() + "," + initialFlow.getTime() + ")" + Constants.MAUDE_END_LINE);
    }
    sb.append(Constants.TABULATION + "eq fls" + Constants.BREAK_LINE);
    sb.append(printMaudeFlows(flows, messageFlows));
    sb.append(Constants.TABULATION + "eq nds" + Constants.BREAK_LINE);
    sb.append(printMaudeNodes(nodes,flows, messageFlows,isExtended));
    sb.append("endm" + Constants.BREAK_LINE);
    return sb.toString();
}
```

Figura 77: Método writeMaudeString de la clase JavaToMaude.

Para la nueva notación se deben añadir ciertas construcciones Maude como son las cabeceras (utilizando los atributos de la clase **Constants**), los identificadores de los eventos y recursos y la definición del conjunto de recursos.

En la figura 78, se puede ver el método usado para declarar los identificadores de los flujos de mensajes. Difiere del resto en que se deben declarar también los identificadores para los constructores de los mensajes que se almacenan dentro de los flujos. Para ello, se debe acceder a cada uno para obtener la información referente a cada tipo de evento (mensaje o timer). El método *printResourcesID()* simplemente recorre la lista y declara los identificadores de acuerdo al lenguaje Maude. Finalmente, dentro de las clases afectadas por la nueva notación se definen ciertos cambios en sus métodos para que el método *getMaudeString()* escriba su nueva información en los métodos *printMaudeNodes()* y *printMaudeFlows()*. Concretamente, la clase **SendTask**, que hereda de **Task** y añade la información adicional referente a los mensajes, Figura 79. Los splits de las puertas, que ahora deben tener en cuenta la posibilidad de tener probabilidades dentro de sus flujos. Para ello, en la clase **Gateway** se debe comprobar si cada flujo tiene almacenado la probabilidad. Sin embargo, el split solo almacena los identificadores de cada flujo debido a como se lee el XML, por lo que se debe iterar sobre la lista de flujos y encontrar aquel con ese identificador, figuras 79 y 80. La clase **SequenceFlow** (que representa a los flujos), escribe una información diferente dependiendo si es un flujo que contenga

información relativa a un evento de mensaje, de tiempo o de ninguno de ellos, figura 82. Por último, la clase **MessageFlow**, que hereda de **SequenceFlow** y es creada para representar a los flujos que envían mensajes, figura 83.

```java
/**
 * Return a String which contains the lists of 'message flow objects' mid separated by \b
 * MessageFlows initialization is different than a normal flow or node because we need to declare its ID and
 * its mID for message nodes
 * @param nodes
 * @return
 */
private static String buildStringFromMessageFlow(List<Node> flows, List<Node> messageFlows) {
    StringBuilder sb = new StringBuilder();
    sb.append(Constants.TABULATION + "ops ");
    for(Node node : flows) {
        SequenceFlow aux = (SequenceFlow) node;
        if(aux.hasMessage())sb.append("m"+node.getId() + " ");
        if(aux.getTimeOutDuration() != null) sb.append("timeout ");
    }
    sb.append(": -> Id" + Constants.MAUDE_END_LINE);
    sb.append(Constants.TABULATION +"--- MessageFlow id declaration\n");
    sb.append(Constants.TABULATION + "ops ");
    for(Node message : messageFlows) {
        sb.append(message.getId() + " ");
    }
    sb.append(": -> FId" + Constants.MAUDE_END_LINE);
    sb.append(Constants.TABULATION +"--- MessageFlow mid declaration\n");
    sb.append(Constants.TABULATION + "ops ");
    for(Node message : messageFlows) {
        sb.append("m" + message.getId() + " ");
    }
    sb.append(": -> Id" + Constants.MAUDE_END_LINE);
    return sb.toString();
}
```

Figura 78: Construcción de identificadores para los flujos de mensajes.

```java
package Elements;

import java.util.List;
/**
 * Task which is able to send a message
 * @author Pablo Espinosa Tarrio
 *
 */
public class SendTask extends Task {
    String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public SendTask(String name,List<String> incoming,List<String> outgoing,String time,String resources) {
        super(name,incoming,outgoing,time,resources);
    }

    public String getMaudeString(List<Node> messageflows, boolean isExtended) {
        if(message == null) return "task(" + this.getId() + ",\"" + this.getName() + "\"," + super.getIncoming().get(0)
                + "," + super.getOutgoing().get(0) + "," + super.getTime() + "," + super.printMaudeResources() + ",empty)"
        else return "task(" + this.getId() + ",\"" + this.getName() + "\"," + super.getIncoming().get(0) + "," +
                super.getOutgoing().get(0) + "," + super.getTime() + "," + super.printMaudeResources()
                + ",(" + message + "))";
    }

}
```

Figura 79: Clase SendTask.

```java
/**
 * If the object has less incomingFlow than outgoing then it is a split, otherwise it is a merge.
 */
public String getMaudeString(List<Node> flows, boolean isExtended) {
    if (incoming.size() < outgoing.size()) {
        return "split(" + this.getId() + "," + this.getNodeType() + "," + incoming.get(0) + ","
                + printProb(outgoing, flows, isExtended) + ")";
    } else {
        return "merge(" + this.getId() + "," + this.getNodeType() + "," + print(incoming) + "," + outgoing.get(0)
                + ")";
    }
}

/**
 * Print the list of flows of the gateway (incoming or outgoing) separated by ','
 * @param list
 * @return
 */
private String print(List<String> list) {
    StringBuilder sb = new StringBuilder();
    sb.append("(");
    for (int i = 0; i < list.size() - 1; i++) {
        sb.append(list.get(i) + ",");
    }
    sb.append(list.get(list.size() - 1) + ")");
    return sb.toString();
}
```

Figura 80: Método getMaudeString de la clase Gateway (split), Parte 1 / 2.

```java
/**
 * Print flows of the gateway
 * 'flows' is needed because we need to access the flow for every ID (by searching it into the list) to check if
 * it has a probability attribute. If that's the case we need to print the flowID + its probability.
 * Otherwise, but in the case that the diagram 'isExtended' we must print flows ID's separated by " " instead of
 * separated by "," like in the previous maude notation
 * @param list flows ID's of the gateway (incoming or outgoing)
 * @param flows
 * @param isExtended
 */
private String printProb(List<String> list, List<Node> flows, boolean isExtended) {
    StringBuilder sb = new StringBuilder();
    sb.append("(");
    for (int i = 0; i < list.size() - 1; i++) {
        Double prob = searchProb(list.get(i), flows); //prob will be nul if the flow doesnt have a prob attribute
        if (isExtended && prob != null)
            sb.append("(" + list.get(i) + ", " + prob + ") ");
        else if (isExtended && !(this instanceof EventBasedGateway))
            sb.append(list.get(i) + " ");
        else
            sb.append(list.get(i) + ",");
    }
    Double prob = searchProb(list.get(list.size() - 1), flows);
    if (isExtended && prob != null)
        sb.append("(" + list.get(list.size() - 1) + ", " + prob + "))");
    else
        sb.append(list.get(list.size() - 1) + ")");
    return sb.toString();
}

/**
 * It looks for a flow with ID=flowId and return its probabilty, which can be null
 * @param flowId
 * @param flows
 * @return
 */
private Double searchProb(String flowId, List<Node> flows) {
    Double res = null;
    for (Node flow : flows) {
        if (flowId.equals(flow.getId())) {
            SequenceFlow aux = (SequenceFlow) flow; // We are sure its a flow
            return aux.getProb(); // this can be null
        }
    }
    return res;
}
```

Figura 81: Método getMaudeString de la clase Gateway (split), Parte 2 / 2.

```java
@Override
public String getMaudeString(List<Node> flows,boolean isExtended) {
    if(hasMessage) return "flow(" + this.getId() + "," + this.getTime()+ ",message(m" + this.getId() +",\"\"))";
    else if(timeOutDuration != null) return "flow(" + this.getId() + "," + this.getTime()+ ",timer(timeout, " +
            timeOutDuration +"))";
    else return "flow(" + this.getId() + "," + this.getTime()+")";
}
```

Figura 82: Método getMaudeString de la de la clase SequenceFlow.

```java
@Override
public String getMaudeString(List<Node> flows,boolean isExtended) {
    if(eventRelatedFlow == null) return "flow(" + this.getId() + "," + this.getTime()+ ")";
    else return "flow(" + this.getId() + ",message(m" + this.getId() +",\"\")," + eventRelatedFlow + ")";
}
```

Figura 83: Método getMaudeString de la clase MessageFlow.

2.3.2.3. Análisis formales implementados

Siguiendo con la secuencia, el siguiente paso es realizar los análisis en el **servidor**. Primero se abre Maude y se importan los scripts que contienen las diferentes librerías y funcionalidades, así como la notación Maude generada por el Parser. En este caso, se deben diferenciar los ficheros a importar dependiendo de si se va a realizar un análisis relacionado con los recursos o no, como se puede ver en la figura 84.

```javascript
//load files for each kind of analysis. Note that for allocation and resource we are using the same files but with paths.
//That is because of the mechanism to stop parallel multiple concurrent simulations
if(type == 'Allocation'){
    maude.stdin.write('load ../maude/gradient-descent/apmaude.maude\n')
    maude.stdin.write(maudeRepresentation + '\n')
    maude.stdin.write('load ../maude/gradient-descent/run.maude\n')
}else if(type == 'Resource'){
    maude.stdin.write('load ../src/maude/gradient-descent/apmaude.maude\n')
    maude.stdin.write(maudeRepresentation + '\n')
    maude.stdin.write('load ../src/maude/gradient-descent/run.maude\n')
}else{
    maude.stdin.write('load ../src/maude/iteration-1/bpmn.maude\n');
    maude.stdin.write(maudeRepresentation + '\n')
    maude.stdin.write('load ../src/maude/iteration-1/verif.maude\n');
}
```

Figura 84: Importación de librerías y funcionalidades en Maude.

Nótese que se están usando dos condiciones separadas para el tipo de análisis **Allocation** (optimización de recursos) y **Resource** (análisis de recursos convencional), cuando realmente utilizan el mismo conjunto de ficheros. Sin embargo, esto se debe a que se ha desarrollado un *mecanismo* para poder ejecutar las simulaciones concurrentes del algoritmo de optimización de manera paralela de tal forma que el proceso se pueda detener de manera sencilla. Entre otras cosas esto requiere ejecutar Node.js desde un proceso del sistema operativo y hace que

los archivos Maude a importar se encuentren en otro *path* (desde aquel en el que se encuentra el fichero JavaScript que ejecuta el proceso). Se detallará un poco más adelante.

Posteriormente, se ejecuta el comando del análisis el cual se crea usando la función *getCommand()*. Esta función recibe por parámetro la información recolectada de los formularios creados en esta fase para ambos análisis (figuras 63 y 64), que son el número de simulaciones junto con el número de instancias y nombres para cada recurso.

Como se puede ver en la figura 85, el método construye una cadena en formato Maude para ejecutar el número dado de simulaciones concurrentes para los recursos y las instancias. Por ejemplo, para **mil simulaciones con diez drones y seis empleados** la cadena que genera el comando sería:

$$pretty - printing(initState(1000, (drone|- > 10, employee|- > 6))).$$

```
function getCommand(population,currentInstances,names){
    if(currentInstances.length != names.length) throw new Error('names.length != currentInstances.length')
    let res = 'rew pretty-printing(initState('+population+',(', aux=[]
    for(let i=0; i<names.length; i++) aux.push(names[i] + ' |-> ' + currentInstances[i])
    res += aux.join(', ')
    res += '))) .\n'
    return res
}
```

Figura 85: Función getCommand.

Finalmente, el comando se ejecuta y cuando se terminan las simulaciones se recibe la salida en el evento correspondiente, devolviendo la información a la función principal del servidor. Una vez explicado esto se pueden detallar las diferencias entre los análisis. Comenzamos primero por el análisis de recursos del requisito **RF - 10**, el cual recordemos que recibe el número de simulaciones concurrentes junto con el nombre de cada recurso y el número de instancias que tiene, para realizar las simulaciones y generar **gráficas** con la información obtenida. La generación de gráficas comienza en el servidor, donde se procesa la salida de Maude para transformar los datos de forma que se le envíe al cliente con el formato de entrada correcto para librería gráfica utilizada, **C3.js**. Esto se hace en un módulo separado, donde se procesa la salida Maude utilizando nuevamente *expresiones regulares* y guardando los campos necesarios utilizando groups, como se muestra en la figura 86.

```
export default async function createCharts(maudeOutput){
let regex = /resource\((?<resname>\w+),\s*(?<numInstances>\d+),
            \s*(?<numInstancesAvailable>\d+),\s*(?<execTime>\d+(\/\d+)?\s*),
            \s*(?<instancesInTime>(\(\s*\d\.\d+(e(\+|\-)\d+)?\s*,\s*\d+\s*\)\s*)*)\)/mg;
let result = [], match
while((match = regex.exec(maudeOutput)) !== null){
    var instancesInTime = getListFromString(match.groups.instancesInTime)
        result.push({res:match.groups.resname,instancesInTime:instancesInTime})
    }
    return result
}
```

Figura 86: Función createCharts.

Se han creado las gráficas utilizando el **tiempo de uso de cada instancia**, de manera que se puede observar de forma visual el número de instancias en uso que hay durante el transcurso de simulaciones. Gracias a esto, un diseñador podría, por ejemplo, detectar de manera rápida si se está usando un número de recursos demasiado grande o demasiado pequeño. Hay otras métricas extraídas que pueden ser objeto de otras gráficas, pero no se ha dispuesto del tiempo necesario para implementarlas y se han plasmado en las *líneas futuras*. Los parámetros utilizados para la generación de estas gráficas son el **nombre del recurso** (para indicar que recurso es objeto de cada gráfica) y la variable llamada **instancesInTime** que representa el conjunto de instancias en uso por instante de tiempo. Se obtiene de la función *getListFromString* que devuelve una dupla con dos listas, una con los valores temporales (eje X) y otra con los valores de las instancias (eje Y), formato de entrada que deben tener los datos para usarlos con C3.js.

Cabe destacar que si le pasamos la información tal cual se obtiene de Maude estaríamos generando una gráfica incorrecta. Por ejemplo, para el par de duplas (1,2) y (2,3), que representan que en el instante de tiempo uno había dos instancias en uso de un recurso y que en el instante dos pasaban a ser tres, La librería lo que hará será unirlos mediante una línea diagonal. Esto no es correcto ya que se estaría representando que en el instante 1.5 había un número de 2.5 instancias de un recurso, lo cual no tiene sentido debido a que las instancias tienen que ser números naturales pues representan entidades indivisibles.

Para solventar el problema, como se puede ver la figura 87, se genera entre cada par de puntos uno artificial. Este punto tendrá el mismo número de instancias que el primero, repre-

sentado con el mismo valor en el eje de ordenadas, y el mismo valor de tiempo restado por un valor muy pequeño en el eje de abscisas. Esto se hace también en getListFromString al mismo tiempo que se le dan a los datos el formato apropiado.

```
function getListFromString(str){
    let listPairs1 = [], listPairs2 = []
    let regex = /\(\s*(?<pair1>\d\.\d+(e\+\d+)?)\s*,\s*(?<pair2>\d+)\s*\)\s*/mg;
    let match = regex.exec(str);
    var pair1 = match.groups.pair1, pair2 = match.groups.pair2;
    listPairs1.push(pair1);
    listPairs2.push(pair2);
    while((match = regex.exec(str)) !== null){
        let nextPair1=Number(match.groups.pair1),nextPair2=match.groups.pair2;
        listPairs1.push((nextPair1-0.0001).toString())
        listPairs2.push(pair2)
        listPairs1.push((nextPair1).toString())
        listPairs2.push(nextPair2)
        pair1=nextPair1;pair2=nextPair2;
    }
    return [listPairs1,listPairs2]
}
```

Figura 87: Función getListFromString.

De esta forma, se consigue que la gráfica se adapte al comportamiento real, que debe ser el aumento en el segundo instante de dos a tres instancias de forma casi inmediata. En las figuras 87 y 88, se muestran respectivamente la gráfica original y la que tiene agregado este procesado. Nótese que las gráficas no son exactamente debido al no determinismo que existe en las simulaciones.

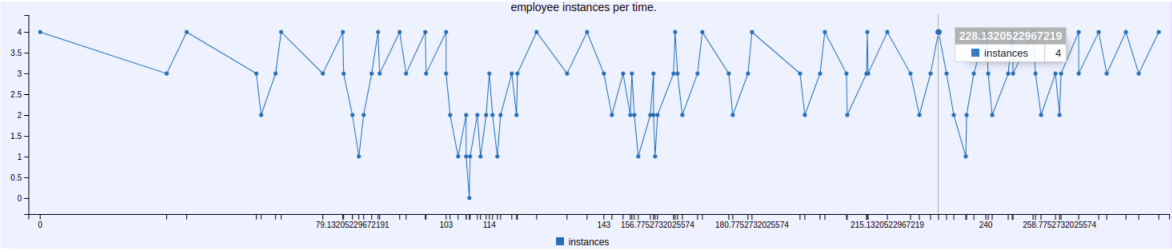


Figura 88: Gráfica original.

Figura 89: Gráfica procesada.

Finalmente, queda mostrar cómo se utiliza la librería C3.js para crear estas gráficas dinámicas. Decimos dinámicas porque la librería permite mostrar los puntos a medida que acercas a ellos el cursor y permite activar/desactivar los datos con los que se grafica, lo que es de gran utilidad si en una misma gráfica coexisten a la vez varios tipos de *datasets*. Inicialmente, se pensó usar esta función para con una sola gráfica mostrar conjuntamente la información de todos los recursos, pero quedó descartado para crear una gráfica por cada uno debido a que se obtiene una mayor legibilidad

La librería es bastante fácil de usar, como se comentó en la introducción. Simplemente se usa el método *c3.generate* y se le proporciona la información necesaria en cada parámetro. Su uso viene detallado en la documentación oficial y en este proyecto solo se usan los que se pueden ver en la figura 90. Con el campo **bindto**, hacemos que la gráfica se genere en el elemento HTML que tiene ese identificador. El campo **data**, es el que tiene mayor importancia y se le especifica mediante el parámetro *xs* el nombre con el que se referencian los datos de las coordenadas de los ejes X e Y. Mediante el parámetro *columns*, se le deben incluir los datos en forma de un array donde el primer elemento es el nombre (dado en xs) al que se refieren y el resto los datos. Para incluir más de un conjunto de datos en una misma gráfica simplemente se añaden tanto sus nombres como los datos al campo xs y columns. Por último, en el campo **title** se le especifica el nombre que se le da a la gráfica.

```
//Generate one chart per resource in diagram. Charts show the amount of resource instances per time.
function generateCharts(data){
  var resultContainer = document.getElementById('results-visualization');
  var resultDiv = document.createElement('div')
  resultDiv.id = 'result-div'
  resultContainer.appendChild(resultDiv)
  for(let resourceData of data){
    var div = document.createElement('div')
    var resname = resourceData.res
    div.setAttribute('id',resname)
    resultDiv.appendChild(div)
    var chart = c3.generate({
        bindto:'#' + resname, //Bind to HTML element with this id
        data: {
            //Y-axis label is set to 'instances' and X-axis one to 'time'
            //The commented code shows how can we integrate different functions together in only one chart, which will be
            //referenced by 'instances', 'y2' and 'y3' in this case
            //Remark: Only removing comments doesnt show all charts, also edit create-charts.js
            xs: {
                instances: "time",
                //y2: "x2",
                //y3: "x3",
            },
            columns: [
                ['instances'].concat(resourceData.instancesInTime[1]),
                ['time'].concat(resourceData.instancesInTime[0]),
                //['y2'].concat(responseData.data.nbavl[1]),
                //['x2'].concat(responseData.data.nbavl[0]),
                //['y3'].concat(responseData.data.queuesize[1]),
                //['x3'].concat(responseData.data.queuesize[0])
            ],
            type: 'line' //Line chart
        },
        title:{
            text: resname + ' instances per time.'
        }
        });
  }
}
```

Figura 90: Generación de gráficas.

El último análisis implementado coincide con el requisito **RF - 11** y trata de encontrar la combinación óptima *(localmente)* de recursos para un diagrama dados el rango de instancias a tener en cuenta y el coste por hora para cada recurso. Para ello, como ya se explicó, se pretende calcular el valor de coste para una combinación inicial dada. Después, se calculará el valor de todas las *combinaciones adyacentes*, llamadas de ahora en adelante **vecinos** por simplicidad, continuando con aquella que tenga un menor coste, para terminar el algoritmo en el momento en el que se obtenga un mínimo local.

Pese a ser un algoritmo ávido que reduce en gran medida el espacio de búsqueda, conlleva una carga computacional considerable. Debido a que para toda combinación de recursos se realizan un gran número de simulaciones concurrentes y su número de vecinos es exponencial (en el número de recursos), concretamente, $3^n - 1$ donde n el número recursos. Por ello, en el desarrollo se presta especial atención a la **eficiencia**, buscando minimizar el tiempo de ejecución.

Al comienzo del script que ejecuta el proceso se definen una serie de variables y estructuras de datos necesarias para su funcionamiento, las cuales se irán explicando conforme sean necesarias. Los parámetros que se reciben son el *nombre de cada recurso, el rango de instancias y su coste por unidad de tiempo*. Para utilizarlos, se decidió utilizar una lista con los nombres de los distintos recursos (llamada **names**), otra con el coste (**resourceCost**) y otra donde cada posición es un array de dos elementos donde la posición cero es el límite inferior de instancias y la posición uno el límite superior (**resourceIntervals**). Para todas ellas el elemento *i-ésimo* almacena la información del recurso **i**. Además, se usa un HashMap (**data**) para emparejar cada combinación de recursos con su valor de coste correspondiente y se utilizará una estructura de datos (**nCartesianProd**) que se crea antes de comenzar el análisis para hacer más eficiente y legible la generación de vecinos. Como hemos comentado, si *n* el número de recursos, el número de vecinos que va a tener una combinación de recursos será como máximo $3^n - 1$, ya que se tiene en cuenta cualquier combinación que sume o reste uno al número de instancias de al menos un recurso. Por ello, para al comienzo del algoritmo se guarda en nCartesianProd, el producto cartesiano del conjunto [1,0,-1] consigo mismo $n - 1$ veces. De esta forma, dada una combinación de recursos $[r_1, r_2, ..., r_n]$ con $r_i \in \mathbb{N}$, se puede generar de forma sencilla y eficiente el conjunto de recursos sumando $[r_1, r_2, ..., r_n]$ con cada elemento de **nCartesianProd**, descartando aquellos en los que los recursos queden fuera de su rango de instancias definido en **resourceIntervals**.

La figura 91 muestra las primeras lineas del flujo principal del algoritmo. Lo primero es definir la lista que se devuelve como solución al terminar el proceso. En cada posición, almacena un objeto de tipo **Node** con dos atributos, la combinación de recursos y su valor de coste. Esta lista va a contener las combinaciones de recursos que han sido seleccionadas para ser expandidos por el algoritmo y en su última posición se encontrará la combinación óptima (localmente). Después, se ejecuta la función **getFirstCombination()** que devuelve la combinación de recursos inicial de la que parte el algoritmo. Para ello, se pueden seguir varias estrategias ya que realmente los resultados dependen de los datos y del diagrama objeto del análisis. En nuestro caso la implementación de la función se puede ver en la figura 92.

```
//Creamos la lista que contendra cada nodo que es seleccionado para la busqueda
let list = []
//Creamos la ED vacia
data = new Map()
let found = false
//Obtenemos combinacion de recursos de partida
let currentInstances = getFirstCombination()
//getting aet from maude process for current resources combination
let currentAET = await execMaude(currentInstances)
console.log('first aet ' + currentAET)
// eval the current resource combination given its aet, cost, ...
let currentEval = evaluation(currentInstances,currentAET)
console.log('first eval is: ' + currentEval)
let node = new Object()
node.resourceCombination = currentInstances
node.eval = currentEval
list.push(node)
// store it in its correspondent position
data.set(JSON.stringify(currentInstances),currentAET)  //we  use  stringify  because  js
maps cant use arrays as keys
...
...
```

Figura 91: Función principal del algoritmo de optimización.

```
//La estrategia que seguiremos para escoger la primera combinacion es obtener el valor
//central de cada intervalo
function getFirstCombination(){
    return resourcesInterval.map((elem) => Math.floor((elem[0]+elem[1])/2))
}
```

Figura 92: Función getFirstCombination.

Como se puede ver, se devuelve en cada posición la **media aritmética** del intervalo de cada recurso de forma que el algoritmo partirá desde el punto central del *espacio de búsqueda*. Posteriormente, se ejecuta el análisis, que se basa en realizar tantas simulaciones concurrentes como haya especificado el usuario, con el coste asociado a cada recurso y con el número de instancias para cada recurso que ha devuelto getFirstCombination(). El análisis lo realiza la función **execMaude**, que se encarga de llamar al script explicado anteriormente para ejecutar Maude y, posteriormente, procesa el resultado con una expresión regular para obtener el tiempo de ejecución medio de todas las simulaciones (**AET**). Además, se utiliza un *await* para esperar a que el valor sea devuelto.

El siguiente paso consiste en calcular la **función de coste** para esa combinación de recursos dado su AET. Sea el conjunto $[r_1, r_2, ..., r_n]$ con $r_i \in \mathbb{N}$ la combinación de recursos,

el conjunto $[c_1, c_2, ..., c_n]$ con $c_i \in \mathbb{N}$ el conjunto de los costes por hora de cada recurso y *AET_weight y Cost_weight* dos constantes que modelan, respectivamente, la importancia del AET y del coste de los recursos, se define la función de coste como:

$$\sum_{i=1}^{n}[\frac{(AET_weight * AET_i) + (Cost_weight * AET_i * r_i * cost_i)}{3600}]$$

Nota: Se divide por 3600 debido a que el coste viene dado en euros por horas y el tiempo de ejecución medio viene dado en segundos.

Sin embargo, AET_weight, Cost_weight y AET_i son constantes para todos los recursos ($\forall i, AET_i = AET$), por lo que se puede simplificar la ecuación como sigue:

$$\frac{n * AET * AET_weight + Cost_weight * AET * \sum_{i=1}^{n}[r_i * cost_i]}{3600}$$

Estos cómputos los realiza la función **evaluation** y su código es el siguiente:

```
// aet is the average time in seconds, cost is the cost per hour and instances is
//the number of resource that are being used in the process
function getResourceCost(aet,cost,instances){
    return (aet / 3600) * cost * instances
}

//Given currentInstances (current combination of resources) and the aet of the process
//It returns the sum of (cost_weight * cost + aet_weight * aet) for each resource
//As aet_w * aet is constant, it is taken out of the sumatory by multiplying //currentinstances.lenght
//and adding it to the result of the computation
function evaluation(currentInstances,aet){
    let sum = 0
    for(let i=0;i<currentInstances.length;i++){
        sum += getResourceCost(aet,resourceCost[i],currentInstances[i])
    }
    return costWeight * sum + currentInstances.length * aetWeight * aet
}
```

Figura 93: Función evaluation.

Llegados a este punto se crea y se introduce en la lista solución el objeto Node, que como se comentó almacena el array con la combinación de instancias inicial junto al valor devuelto por evaluation. Dichos valores también se almacenan en la estructura de datos (**data**). Nótese el uso del método *JSON.stringify* para convertir el array de instancias en una cadena de texto JSON para poder usarla como *clave* dentro de la estructura.

A continuación, comienza el proceso iterativo que se repite hasta que se encuentra una solución, figura 94. En primer lugar, se genera el conjunto de vecinos llamando a la función **generateNeighbours()**, figura 95, que utiliza la estructura de datos explicada anteriormente

para generar el conjunto de vecinos (**nCartesianProd**), teniendo en cuenta que ninguno de los recursos tome un valor fuera de su rango definido y que la combinación resultante no tenga un valor definido en el mapa data, ya que esto implicaría que su valor ya ha sido calculado.

```
while(!found){
    //generamos vecinos del estado actual y nos quedamos con aquellos que no hayan
    //sido calculados
    let neighbours = generateNeighbours(currentInstances)
    //Hay vecinos?
    // no --> terminamos y devolvemos la lista
    if(neighbours.length == 0) found = true
    else{
        // si --> ejecutamos los vecinos y obtenemos el vecino con eval minimo
        let neighboursEvals = await execNeighbours(neighbours)
        // minNeighbour es un objeto con dos att: value, eval del minimo vecino e
        // index, el indice del vecino con ese eval
        let minNeighbour = getMinNeighbour(neighboursEvals)
        // el minimo vecino tiene una eval mas pequeña que la actual --> current es
        // igual al minimo y repetimos
        if(minNeighbour.value < currentEval){
            currentEval = minNeighbour.value
            currentInstances = neighbours[minNeighbour.index]
            let node = new Object()
            node.resourceCombination = currentInstances
            node.eval = currentEval
            list.push(node)
        //Guardamos cada vecino en la estructura de datos
        //esto se hace en este punto y no antes porque si algoritmo terminase en esta
        //iteracion (si se ejecuta el else, no es necesario almacenar en la ED los
        //vecinos)
            neighbours.forEach((elem,i) =>
                data.set(JSON.stringify(elem),neighboursEvals[i]))
        } else found = true //otherwise --> terminamos y devolvemos current
    }
}
```

Figura 94: Proceso iterativo del método principal.

```
function generateNeighbours(currentInstances){
    let res=[]
    for(let elem of nCartesianProd) {
        let neighbour = elem.map((e,i) => e + currentInstances[i])
        if(data.get(JSON.stringify(neighbour)) == undefined && inRange(neighbour))
            res.push(neighbour)
    }
    return res
}
```

Figura 95: Función generateNeighbours().

Si el número de vecinos es igual a cero, se termina la ejecución y se devuelve la lista actual. En otro caso se ejecutan el análisis Maude sobre todos los vecinos llamando a la función **execNeighbours()**, figura 96. Este punto es uno de los más importantes ya que representa el mayor esfuerzo computacional del proceso, el cual como comentamos requiere ser realizado de la forma más eficiente posible.

```
//Ejecutamos mode para cada proceso, al estar creando procesos diferentes para cada vecino lo estamos haciendo en paralelo y lo hacemos
//ejecutando promises.all, ojo,  Tiene fast fail behaviour
async function execNeighbours(neighbours){
    let res
    await Promise.all(neighbours.map(elem => maude.runMaude(undefined,
        maudeRepresentation, 'Allocation', elem, population, names)))
        .then((result) =>{
            res = result.map((elem,i) => evaluation(neighbours[i],
            parserMaudeResult(elem)))
            console.log(res)
        })
        .catch((err) => {
            console.log('Error en promiseAll ' + err)
            return Promise.reject(new Error(err))
        })
    return res
}
```

Figura 96: Función execNeighbours().

Para ello, se combina la ejecución de los análisis mediante procesos del sistema operativo (usando la librería **child_process** de Node.js) con el método **Promises.all** para lanzar en paralelo cada uno de los análisis y esperar a que todos terminen. El método Promises.all es en sí una promesa que recibe como parámetro un array donde cada elemento es otra promesa y termina revolviéndola y devolviendo el valor de cada una si todas se ejecutan correctamente o termina rechazando si una sola de ellas falla, lo que se conoce como *fast-fail behaviour*. Estas promesas se crean utilizando la función *map* sobre el array de vecinos, que recibe como parámetro un función que se aplica sobre cada elemento de la lista. En este caso la función es una lambda que toma "elem", el cual referencia a cada elemento de la lista, y ejecuta sobre él la función que ejecuta el análisis Maude dentro de una promesa. Finalmente, en el then, el cual se ejecuta cuando la promesas terminan correctamente, se llama a **evaluation** para cada vecino usando la función map de nuevo y se devuelve una lista donde el elemento *i-ésimo* contiene el valor de coste para el vecino *i*. Esta lista se devuelve al método principal del algoritmo y sobre ella se ejecuta la función **getMinNeighbour**, la cual recorre la lista y devuelve un objeto que contiene el índice del vecino con menor coste junto con su valor. Una vez se obtiene este objeto en el main, llamado minNeighbour, se comprueba si el nodo que se estaba procesando tiene un valor de coste menor o mayor que él. En el caso en el que el nodo actual tenga un coste menor, se termina el algoritmo y se devuelve la lista con la función, pues el nodo actual es un mínimo local. En caso contrario, el nodo actual se intercambia con minNeighbour y se repite el proceso.

De esta forma al terminar el proceso se obtiene la lista de nodos, que es devuelta como

resultado al módulo principal del servidor, el cual almacena la información en un JSON y la envía al cliente, que tendrá que procesarla para construir la salida **(RF - 18)**.

Antes de terminar es necesario explicar lo siguiente. Como se ha mencionado con el resto de análisis, existe un requisito del cual dependen todos los análisis. El requisito **RF - 22**, que especifica que se debe proporcionar una manera de parar los análisis. Hemos explicado que esto se realiza mapeando el proceso del sistema operativo que realiza análisis Maude con la *sesión* del cliente, para ejecutar un kill sobre dicho proceso en el momento en el que el usuario pulse el botón Stop Analysis del formulario de la figura 42. Sin embargo, como se ha explicado en este análisis se crean múltiples procesos que ejecutan Maude manera secuencial (para el primer nodo en expandir) y concurrente (para el resto), por lo que surge el problema de encontrar una forma eficiente y sencilla de detenerlos en el momento en el que el usuario pulse el botón.

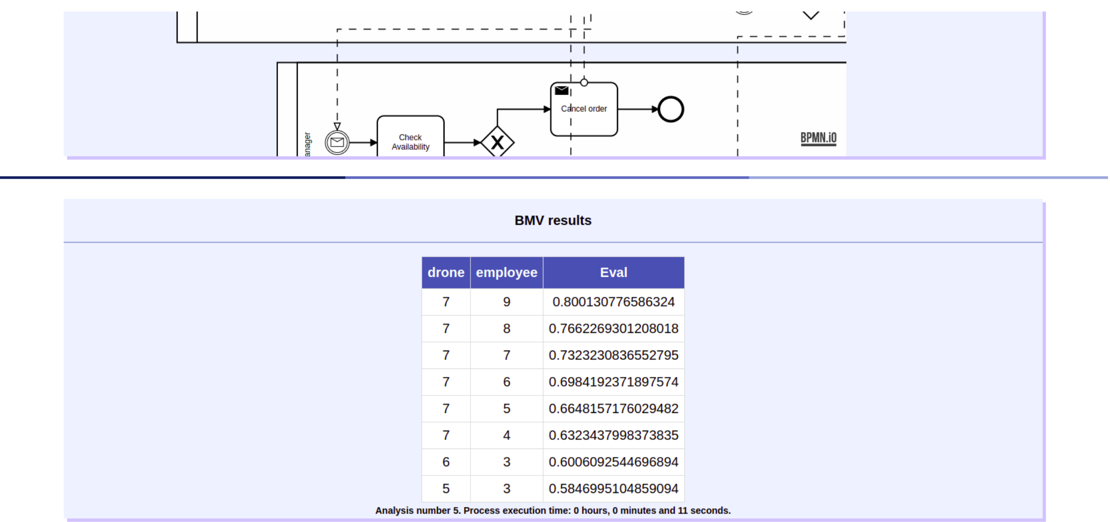
Para solventarlo, se podría almacenar cada proceso creado con la sesión del cliente, pero como se ha comentado el número de vecinos es exponencial en el número de recursos. Esto hizo que no se considerase esta solución debido a que el gran número de elementos a almacenar en la estructura de datos supondría un gran *overhead* de memoria que podría colapsar al servidor si entran en juego numerosos usuarios que realicen este tipo de análisis. Por tanto, se decidió crear otro módulo llamado **exec-algorithm**, el cual utilizando nuevamente la librería child_process de Node.js, crearía un proceso al comienzo del análisis que se encargaría de ejecutar el script que realiza el algoritmo de optimización. Este proceso es el único que se almacena junto con la sesión del cliente y será el encargado de crear el resto de procesos necesarios para el análisis. De esta manera, cuando el usuario pulse el botón, este proceso será el que reciba la señal, parando todos los subprocesos activos en ese momento. Esto se facilita usando el módulo de npm **tree-kill**, que mediante su método kill mata un proceso junto con todos sus hijos.

2.3.2.4. Resultados obtenidos

Finalmente, pasamos a mostrar los resultados que se consiguen de cada uno de los análisis implementados en esta segunda iteración. En la figura 97 se puede ver el resultado del análisis de recursos, requisito **RF-10**, sobre el diagrama de ejemplo (figura 14) para *300* simulaciones concurrentes con *4 drones* y *3 empleados.*

En la figura 98 se puede ver el resultado del análisis de optimización de recursos, requisito **RF-11**, sobre el mismo diagrama. Se han usado *100* simulaciones concurrentes, un rango de *tres a doce drones*, donde cada uno cuesta a *diez euros la hora* y un rango de **tres a quince** empleados donde cada uno cuesta *cinco euros la hora*. Como se puede ver, la combinación óptima es *5 drones* y *tres empleados* con un valor de evaluation de aproximadamente *0.585* unidades.

Figura 97: Resultado del análisis de recursos para el diagrama de ejemplo (figura 14).



drone	employee	Eval
7	9	0.800130776586324
7	8	0.7662269301208018
7	7	0.7323230836552795
7	6	0.6984192371897574
7	5	0.6648157176029482
7	4	0.6323437998373835
6	3	0.6006092544696894
5	3	0.5846995104859094

Analysis number 5. Process execution time: 0 hours, 0 minutes and 11 seconds.

Figura 98: Resultado del análisis de optimización para el diagrama de ejemplo (figura 14).

3
Conclusiones y Líneas Futuras

3.1. Conclusiones

A lo largo del proyecto se comenta varias veces la importancia que tienen los **procesos de negocio** y cómo su correcto funcionamiento es estrictamente necesario para que una empresa pueda conseguir sus objetivos. Es por esto que surgen áreas como el modelado de procesos de negocio y la necesidad de analizarlos formalmente para prevenir errores de diseño y para poder comparar unos con otros.

BPMN es la notación más utiliza hoy en día para modelar estos procesos, sin embargo, con la tecnología actual aún no es posible realizar análisis formalmente sobre todos sus elementos debido a la gran expresividad que lo caracteriza. Los análisis desarrollados por mi tutor y sus colaboradores, junto con el análisis de optimización proporcionado como parte de este proyecto, son una contribución en esa dirección, en la cual se espera seguir avanzando para poder llegar un día a realizar análisis de toda clase de propiedades formales pulsando solo un botón. Destacar que además de implementar un nuevo análisis, la principal aportación del proyecto es proponer una **interfaz unificada** en forma de aplicación web con la que se puedan crear diagramas y analizarlos, de manera que se pueda escalar en un futuro al mismo tiempo que se desarrollan nuevos tipos de análisis.

Por último, quiero dar mi agradecimiento a Francisco Durán Muñoz, tutor de este trabajo, por proponerme este interesante proyecto con el que he aprendido tantas cosas. En primer lugar, he podido poner en práctica todos los conocimientos necesarios para desarrollar desde cero una aplicación web, la cual se ha desarrollado en **Node.js**, un entorno de ejecución que permite utilizar JavaScript en el servidor. Presenta características diferentes a los len-

guajes y plataformas que he usado a lo largo de la carrera como, por ejemplo, ser orientado a eventos, asíncrono y no bloqueante, permitiéndome aprender nuevos estilos de programación. Finalmente, el proyecto también me ha permitido adentrarme en el mundo de la lógica de reescritura, de **Maude** y de los análisis formales de procesos de negocio, áreas que de otra forma hubiera sido difícil entender y conocer.

3.2. Líneas futuras

El sitio web presenta un buen número de funcionalidades, sin embargo, existen distintas características que hubieran aumentado en gran manera el valor aportado y que por falta de tiempo no pudieron ser llevadas a cabo. En esta sección se proponen una serie de estas características para trabajos y desarrollos futuros.

3.2.1. Puesta en producción de la aplicación

Se destacan los cambios necesarios para poner en producción esta aplicación. Entre estos destaca utilizar las configuraciones necesarias en Webpack para aplicar *minify and uglify* sobre el código fuente. Estas técnicas consisten en realizar distintas transformaciones sobre el mismo, como pueden ser la eliminación de espacios, variables, saltos de línea, etc. De tal manera que el código no sea legible, pero ocupe menos memoria y se renderice más rápido en el lado del cliente. La aplicación se alojaría en un servidor dedicado de la Universidad Málaga preparado con todas las configuraciones necesarias.

3.2.2. Procesamientos más refinados en las salidas

1. Mostrar las trazas de contraejemplo directamente sobre el diagrama:

   Como se menciona en las fases del trabajo, el model checker de Maude devuelve, cuando el diagrama no satisface la propiedad LTL introducida, un contraejemplo con la traza de la ejecución que conduce a ese resultado. Por ende, se estudió la posibilidad mostrar de forma visual dicha traza sobre el diagrama BPMN, ya que resultaría de gran utilidad para diagramas con una gran cantidad de elementos en los que pueda resultar realmente difícil para una persona encontrar estos errores de diseño.

Una de las posibilidades es utilizar una funcionalidad que ofrece el *modeler de Camunda*, la cual permite añadir color a un elemento. De esta forma, se podría interactuar con el contraejemplo dado por Maude para tratar de detectar esos elementos que conforman la traza y generar un nuevo XML que represente el diagrama donde estos elementos aparezcan con las propiedades necesarias para que el modeler los muestre de otro color.

2. Mostrar de manera dinámica cada nodo expandido en el algoritmo de optimización: Esta funcionalidad es de utilidad ya que este análisis puede llegar a tardar un tiempo considerable y que el usuario pueda ir viendo los resultados de manera dinámica tiene dos ventajas principales. La primera es que el usuario puede saber que no hay ningún error en el desarrollo del proceso y la segunda es que es posible que los resultados parciales le aporten al usuario el conocimiento necesario para volver a lanzar la ejecución con distintos parámetros o directamente terminar el análisis sin necesidad de que se ejecute por completo.

3.2.3. Nuevos algoritmos de optimización

El algoritmo de optimización de recursos es una de las funcionalidades más útiles de la aplicación, sin embargo, utiliza un enfoque ávido que se puede mejorar. Este se basa en el *descenso del gradiente*, buscando escoger en cada paso el mejor vecino, es decir, aquel cuyo coste sea mejor. Por ende, se propone realizar otros análisis de optimización más refinados.

1. Simulated annealing: El simulated annealing o recocido simulado es un algoritmo similar al descenso del gradiente, ya que trata de encontrar y escoger el mejor vecino, sin embargo, solo expande ese vecino con una cierta probabilidad, que se reduce con el transcurso del algoritmo. Con esto se pretende evitar que el algoritmo encuentre un mínimo local de manera prematura.

2. Múltiples búsquedas: Otra posibilidad es implementar un algoritmo que supone una carga computacional mayor, pero que aumenta mucho las posibilidades de encontrar una solución más óptima. En el algoritmo implementado como parte de este trabajo se comienza ejecutando la combinación de recursos que se encuentra en el centro del *espacio de búsqueda*, por ende, se propone lanzar en paralelo varias ejecuciones del algoritmo

desde diversos puntos de inicio diferentes, para finalmente devolver la mejor solución de todas.

3.2.4. Liveness

Como se ha explicado a lo largo del trabajo, la verificación de la propiedad **liveness** permite comprobar si un diagrama está *libre de deadlocks*. Sin embargo, hay una limitante en este análisis y es que, con la implementación actual, cuando termina sin encontrar un deadlock no se puede saber si es porque el diagrama termina correctamente o porque se ha consumido la memoria dedicada al análisis. Esto se debe a que los diagramas pueden tener bucles infinitos y la memoria se limita para garantizar que el análisis siempre termina. Por tanto, se propone para trabajos futuros implementar una función que permita distinguir entre estos dos casos, aportando más valor a los usuarios.

Referencias

[1] *About npm | npm Docs*. URL: https://docs.npmjs.com/about-npm.

[2] *bpmn-js walkthrough | Toolkits | bpmn.io*. URL: https://bpmn.io/toolkit/bpmn-js/walkthrough/.

[3] *bpmn-js: BPMN 2.0 rendering toolkit and web modeler | Toolkits | bpmn.io*. URL: https://bpmn.io/toolkit/bpmn-js/.

[4] *C3.js | D3-based reusable chart library*. URL: https://c3js.org/.

[5] *CommonJS: JavaScript Standard Library*. URL: https://www.commonjs.org/.

[6] *Documentación | Node.js*. URL: https://nodejs.org/es/docs/.

[7] Francisco Durán, Camilo Rocha y Gwen Salaün. "Analysis of Resource Allocation of BPMN Processes". En: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11895 LNCS (2019), págs. 452-457. ISSN: 16113349. DOI: 10.1007/978-3-030-33702-5_35/COVER/. URL: https://link.springer.com/chapter/10.1007/978-3-030-33702-5_35.

[8] Francisco Durán, Camilo Rocha y Gwen Salaün. "Analysis of the Runtime Resource Provisioning of BPMN Processes Using Maude". En: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12328 LNCS (2020), págs. 38-56. ISSN: 16113349. DOI: 10.1007/978-3-030-63595-4_3/COVER/. URL: https://link.springer.com/chapter/10.1007/978-3-030-63595-4_3.

[9] Francisco Durán, Camilo Rocha y Gwen Salaün. "Computing the parallelism degree of timed BPMN processes". En: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11176 LNCS (2018), págs. 320-335. ISSN: 16113349. DOI: 10.1007/978-3-030-04771-9_24/COVER/. URL: https://link.springer.com/chapter/10.1007/978-3-030-04771-9_24.

[10] Francisco Durán, Camilo Rocha y Gwen Salaün. "Stochastic analysis of BPMN with time in rewriting logic". En: *Science of Computer Programming* 168 (dic. de 2018), págs. 1-17. ISSN: 0167-6423. DOI: 10.1016/J.SCICO.2018.08.007.

[11] Francisco Durán y Gwen Salaün. "Verifying timed BPMN processes using Maude". En: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10319 LNCS (2017), págs. 219-236. ISSN: 16113349. DOI: 10.1007/978-3-319-59746-1_12/TABLES/1. URL: https://link.springer.com/chapter/10.1007/978-3-319-59746-1_12.

[12] Francisco Durán y col. "Rewriting Logic Approach to Resource Allocation Analysis in Business Process Models". En: *Science of Computer Programming* 183 (2019), págs. 1-32. DOI: 10.1016/j.scico.2019.102303ï. URL: https://hal.inria.fr/hal-02345895.

[13] Dr. Steven Eker, José Meseguer y Ambarish Sridharanarayanan. *The Maude LTL Model Checker*. 2002. URL: http://www.csl.sri.com/papers/1555/.

[14] *Express - Infraestructura de aplicaciones web Node.js*. URL: https://expressjs.com/es/.

[15] *Figma: the collaborative interface design tool*. URL: https://www.figma.com/.

[16] *Fundamentos | Node.js*. URL: https://bluuweb.github.io/node/01-fundamentos/.

[17] Mario Garcia. *V8, el motor de código desarrollado por Google*. 2021. URL: https://www.drauta.com/v8-el-motor-de-codigo-desarrollado-por-google.

[18] *Introducción a XML - XML: Extensible Markup Language | MDN*. 2021. URL: https://developer.mozilla.org/es/docs/Web/XML/XML_introduction.

[19] *ISO - ISO/IEC 19510:2013 - Information technology — Object Management Group Business Process Model and Notation*. 2013. URL: https://www.iso.org/standard/62652.html.

[20] *JavaScript | MDN*. 2022. URL: https://developer.mozilla.org/es/docs/Web/JavaScript.

[21] *jQuery*. URL: https://jquery.com/.

[22] *Manejos de archivos y asincronismo en NodeJS*. URL: https://utn-fullstack.github.io/clases/clase1.html.

[23] *Maude manual introduction*. URL: https://maude.lcc.uma.es/manual271/maude-manualch1.html.

[24] *Modules: ECMAScript modules | Node.js v18.4.0 Documentation.* URL: https://nodejs.org/api/esm.html#modules-ecmascript-modules.

[25] Alexander Nnakwue. *How to transpile ES modules with webpack and Node.js.* 2021. URL: https://blog.logrocket.com/transpile-es-modules-with-webpack-node-js/.

[26] *NVM.* URL: https://desarrolloweb.com/home/nvm.

[27] Debbie O'Brien y col. *Why webpack | webpack.* URL: https://webpack.js.org/concepts/why-webpack/.

[28] Nancy Piedad y Molina Montoya. "¿Qué es el estado del arte?" En: *Ciencia y Tecnología para la Salud Visual y Ocular* 3 (5 ene. de 2005), págs. 73-75. ISSN: 1692-8415. DOI: https://doi.org/10.19052/sv.1666. URL: https://ciencia.lasalle.edu.co/svo/vol3/iss5/10.

[29] PROcesses y Services Lab. *BProVe - PROS.* URL: https://pros.unicam.it/bprove/.

[30] *Promise - JavaScript | MDN.* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[31] *SAX.* URL: http://www.jtech.ua.es/j2ee/2003-2004/modulos/xml/sesion03-apuntes.htm.

[32] *Software Java | Oracle España.* URL: https://www.oracle.com/es/java/.

[33] Miguel Palomino Tarjuelo. *Reflexión, abstracción y simulación en la lógica de reescritura.* 2005. URL: https://dialnet.unirioja.es/servlet/tesis?codigo=19754.

[34] *The Maude System.* URL: http://maude.cs.illinois.edu/w/index.php/The_Maude_System.

[35] *Webpack.* URL: https://webpack.js.org/.

Apéndice A
Documento General de Requisitos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

Documento General de Requisitos

Realizado por
Pablo Espinosa Tarrío

Tutorizado por
Francisco Javier Durán Muñoz

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE
MÁLAGA

1. Introducción

El objetivo de este documento no es otro que analizar y definir las necesidades y funcionalidades del sistema que se quiere desarrollar. Para ello, se ha elaborado una listas con los requisitos funcionales que se implementan en el sistema y otra con los requisitos no funcionales a los que se adapta. Ambos se definen en una tabla de cuatro columnas, donde cada una se describe a continuación:

- Identificador: Identificador único que hace referencia al requisito en cuestión. El identificador de los requisitos funcionales y no funcionales se definen, respectivamente, mediante las letras RF y RNF, ambos acompañados de un valor numérico, según el orden de aparición en la tabla.

- Nombre: El nombre del requisito.

- Información: Campo de texto que contiene información de la funcionalidad que aporta (requisito funcional) o de las restricciones que impone (requisito no funcional).

- Dependencias: Referencia los requisitos de los cuales depende para poder ser llevado a cabo. Si un requisito no depende de ningún otro, entonces tendrá un guión (-) en este campo.

2. Requisitos funcionales

Identificador	Nombre	Información	Dependencias
RF - 01	Crear diagrama	Se permitirá la creación de diagramas BPMN directamente en la aplicación de forma gráfica, sencilla y eficiente.	RNF - 01
RF - 02	Cargar diagrama	Se permitirá la visualización de diagramas BPMN que cumplan con el estándar 2.0 seleccionando un archivo local con extensión .bpmn.	RNF - 01
RF - 03	Editar diagrama	Se permitirá la edición de diagramas BPMN creados o cargados en la aplicación.	RNF - 01
RF - 04	Descargar diagrama	Se facilitará la descarga de los diagramas creados, cargados o editados en la aplicación a través de un botón ubicado sobre el modeler.	RNF - 01
RF - 05	Analizar diagrama	Se podrán realizar análisis formales sobre cada uno de los diagramas.	RNF - 02

Identificador	Nombre	Información	Dependencias
RF - 06	Tiempo de ejecución	Se computará el tiempo de ejecución mínimo, máximo y medio.	RF - 05
RF - 07	Grado de paralelismo	Se calculará el grado de paralelismo. Esto es, el máximo número de tareas que se pueden estar ejecutando de manera simultánea.	RF - 05
RF - 08	Model Check	Permitirá comprobar propiedades de lógica temporal lineal siguiendo una semántica basada en tokens.	RF - 05 RNF - 03 RNF - 04
RF - 09	Liveness	Permite verificar si el diagrama está libre de deadlocks. Esto es, si termina para todas sus posibles ejecuciones.	RF - 05
RF - 10	Análisis de recursos	Se obtendrá diversa información sobre los recursos usados por el diagrama como el tiempo de uso, el tiempo medio de ejecución para un número de dado de simulaciones concurrentes, el tiempo de bloqueo en las actividades y en las puertas de eventos, etc. Esta información será procesada para crear distintas gráficas dinámicas que se mostrarán en la web.	RF - 05 RNF - 03 RNF - 04
RF - 11	Optimización de recursos	La información de los recursos discutida en el requisito de arriba (RF - 10) se utilizará para calcular una combinación de recursos óptima (localmente) en función de su coste y tiempo de ejecución.	RF - 05 RNF - 03 RNF - 04
RF - 12	Número de recursos	Se le solicitará al usuario el número de recursos que utiliza en su diagrama	RF - 09 RF - 10
RF - 13	Número de simulaciones	Se le solicitará al usuario el número de simulaciones concurrentes que quiere ejecutar para el análisis en cuestión.	RF - 09 RF - 10
RF - 14	Nombre de recursos	Se le solicitará al usuario el nombre de cada recurso.	RF - 09 RF - 10
RF - 15	Número de instancias	Se le solicitará al usuario el número de instancias de cada recurso	RF - 09
RF - 16	Rango de instancias	Se le solicitará al usuario el número mínimo y máximo de instancias de cada recurso que quiere que se tengan en cuenta en el análisis.	RF - 10

Identificador	Nombre	Información	Dependencias
RF - 17	Coste	Se le solicitará al usuario el coste por hora que le supone cada recurso.	RF - 10
RF - 18	Procesar salidas	Todos y cada uno de los análisis devuelven información no estructurada que requiere conocimiento experto para ser de utilidad. Por ende, se proporcionará un mecanismo para dar formato a las salidas y hacer que sean útiles para cualquier usuario.	RF - 05
RF - 19	Parser	Los diagramas BPMN están representados internamente por documentos XML, pero los análisis formales desarrollados en Maude necesitan de una especificación determinada de los procesos para funcionar. Por tanto se desarrollará un algoritmo que permitirá generar una representación Maude correcta a partir de todo diagrama BPMN válido.	-
RF - 20	Informar a los usuarios	Se implementarán diferentes botones de ayuda que mostrarán información a los usuarios de la web para explicarles cómo pueden realizar cada uno de los análisis de forma correcta.	RF - 05
RF - 21	Página de bienvenida	Se dedicará una espacio del sitio web para explicar de manera general que es y como se puede utilizar la web, proporcionando referencias a los papers publicados por mi tutor y su equipo que tienen relación con la página para que cualquier usuario interesado pueda tener más información sobre cada tipo de análisis.	-
RF - 22	Parar análisis	Se debe permitir que los usuarios puedan parar un análisis que se encuentre en ejecución de forma sencilla y rápida	RF - 05 RNF - 03 RNF - 04
RF - 23	Mostrar tiempo	Para los análisis no inmediatos (duración superior a 2s) se mostrará un cronómetro dinámico que le vaya indicando al usuario cuánto tiempo lleva realizándose el proceso.	RF - 05 RNF - 03 RNF - 04
RF - 24	Añadir información adicional	Se debe permitir que el usuario añada a los diagramas la información adicional necesaria para realizar los análisis de forma fácil e intuitiva.	-

3. Requisitos no funcionales

Identificador	Nombre	Información	Dependencias
RNF - 01	Cumplir con el estándar	Todos y cada uno de los diagramas BPMN usados en la aplicación deberán cumplir con el estándar 2.0 de BPMN.	RF - 01 RF - 02 RF - 03 RF - 04
RNF - 02	Diagramas válidos	Se deberá informar al usuario acerca de si su diagrama está correctamente diseñado antes de ser analizado.	RF-05
RNF - 03	Mover formularios	Se deben poder arrastrar los formularios para facilitar el escribir sobre ellos al mismo tiempo que se puede observar el diagrama, de forma que la utilización de la página sea más cómoda para el usuario.	RF - 08 RF - 09 RF - 10 RF - 21
RNF - 04	Cerrar formularios	Se deben poder cerrar todos los formularios pulsando un botón sin necesidad de realizar el análisis.	RF - 21

Apéndice B
Manual de Instalación

Para ejecutar, instalar y utilizar la aplicación a partir de los ficheros fuente es necesario cumplir los requisitos y seguir los pasos indicados en este manual.

B.1. Sistema operativo

El servidor debe ejecutar programas escritos en Maude para realizar los análisis. Maude solo puede ser ejecutado en **MacOS** o en ciertas distribuciones de **Linux**. Por ello, se recomienda utilizar uno de estos dos sistemas o una máquina virtual en su defecto. Para usuarios de *Windows* también existe la posibilidad de utilizar **Windows Subsystem for Linux (WSL)**, un sistema desarrollado por Microsoft que permite el uso de ejecutables Linux en su sistema operativo.

B.2. Versiones y las herramientas necesarias

La aplicación está desarrollada utilizando Node.js junto con un script escrito en Java que se ejecuta instalando un *jar* en el servidor. Por tanto, es necesario tener instalado **Node.js, Java** (jdk) y **npm**.

Es muy probable que la aplicación se pueda ejecutar utilizando las últimas versiones de estas herramientas. Sin embargo, para asegurar el correcto funcionamiento se recomienda instalar las versiones que se usaron durante el desarrollo de las mismas. Para facilitar la instalación de npm y Node.js se recomienda utilizar **nvm**, un software que permite instalar y mantener de manera sencilla distintas versiones de Node.js y npm en un mismo sistema. Las versiones utilizadas fueron:

- Node.js: v12.18.3
- npm: 6.14.6
- JDK (Java Development Kit): 17.0.3

B.3. Instalación de dependencias

El siguiente paso será descargar e instalar todas las dependencias necesarias para ejecutar el código. Para ello, se debe abrir un *terminal* en el directorio raíz del proyecto y ejecutar el

comando **npm install**. Al terminar de ejecutarse se habrán descargado e instalado correctamente todas las dependencias.

B.4. Despliegue del servidor

Para arrancar el servidor web, simplemente debemos ejecutar (también en el directorio raíz del proyecto) el comando **npm run reboot**, que accionará el script descrito en el fichero *package.json* que se explicó en las fases de trabajo. De esta forma tendremos un mensaje en consola que indicará que el servidor está esperando respuestas en el puerto 8080, por lo que simplemente se debe abrir un navegador web y acceder a la dirección: https://localhost:8080/ para poder utilizar la aplicación.

Apéndice C
Manual de Usuario

En este manual detalla el funcionamiento de la aplicación desde la perspectiva del usuario. El sitio web se llama **BPMN Verifier**, y su objetivo es permitir que los usuarios creen y analicen formalmente diagramas BPMN. Consta de dos páginas explicadas a continuación.

C.1. Landing Page

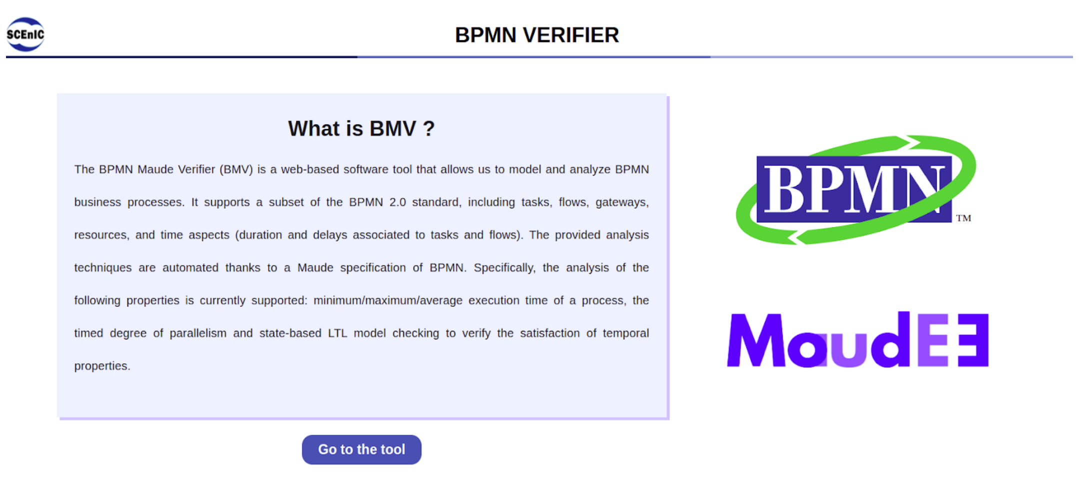


Figura 99: Landing Page, parte 1 / 2.

Figura 100: Landing Page, parte 2 / 2.

Landing Page es el nombre comúnmente usado en la ingeniería del software para referirse a la primera página que ven los usuarios de un sitio web, figuras 98 y 99. En este proyecto,

esta página está pensada para ofrecer información básica a los usuarios. En primer lugar, podemos hablar de la cabecera, que muestra el nombre de la aplicación junto con una imagen del logo del equipo de investigación de mi tutor y sus colaboradores, que al mismo tiempo es un hipervínculo a dicha página. El resto de la página se divide en tres secciones diferenciadas:

- Sección 1: muestra un resumen de las aportaciones de la herramienta y las técnicas que utiliza. Además muestra dos imágenes con las principales herramientas que usa la aplicación: BPMN y Maude. Por último, tiene un botón que dice "Go to the tool", el cual puede ser pulsado para acceder a la página principal.

- Sección 2: muestra un diagrama del flujo básico que debe seguir un usuario para utilizarla, el cual se detallará a continuación.

- Sección 3: muestra una sección con los enlaces a las publicaciones del tutor de este trabajo relacionadas con él, ya que los análisis que se realizan en la aplicación están detallados y explicados en ellos y se piensa que pueden ser útiles para aquellos usuarios interesados.

C.2. Página principal

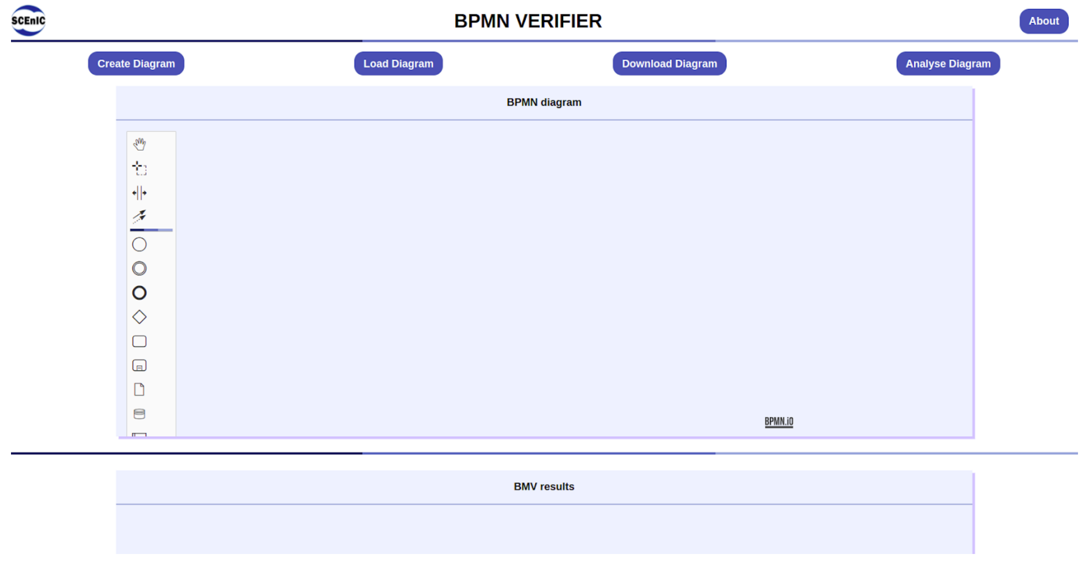


Figura 101: Página principal.

La figura 100 muestra la que, como su propio nombre indica, es la página más importante del sitio web, ya que es aquí donde los usuarios interactuarán con los diagramas BPMN a través del *modeler* y donde realizarán los análisis. El encabezado es idéntico al de la página anterior excepto porque tiene un botón llamado About que sirve de enlace para volver al Landing Page, por si el usuario quiere volver a consultar la información allí presente.

Lo siguiente que se puede ver son los botones que se encuentran encima del modeler:

- Create Diagram: Activa el modeler para que el usuario pueda crear un diagrama desde cero en la página.

- Load Diagram: Abre el sistema de archivos del ordenador del cliente y permite que este seleccione y añada un diagrama al modeler. Solo permite que se seleccione un archivo de extensión .bpmn, abortando el proceso y mostrando un error en caso contrario.

- Download Diagram: Descarga el diagrama que se encuentra en ese momento en el modeler.

- Analyse Diagram: Muestra el menú lateral que muestra un botón para cada análisis que se puede realizar. Para que despliegue el menú es necesario que haya un diagrama creado, en caso contrario se mostrará un error.

Debajo de estos botones se encuentra una parte fundamental de la aplicación: el **modeler**. Como hemos comentado, se integró con la aplicación el modeler de Camunda para permitir la creación de diagramas. En la parte izquierda podemos ver una barra que contiene los elementos que se pueden insertar y que son parte del estándar BPMN 2.0, tales como las *tareas, eventos o flujos*. En el centro se encuentra el espacio dedicado a la creación, visualización y edición de los diagramas. Por último, a la derecha se encuentra el panel que se añadió para especificar aquellas propiedades que son necesarias para el análisis y no forman parte del estándar BPMN 2.0. Las extensiones añadidas son:

- Tiempo: Se permite añadir el tiempo de ejecución de las tareas y flujos. Debe ser un valor numérico para los análisis de la iteración uno. Para los análisis de recursos (iteración dos) puede ser tanto un valor numérico como una de las distribuciones de probabilidad admitidas, de acuerdo a su respectiva sintaxis:

    - Distribución normal: Norm(x, y), donde "x" es un número real que representa la media de la distribución e "y" es un número real que representa su varianza.
    - Distribución uniforme: Unif(x,y) donde "x" e "y" son dos números reales que representan los límites del intervalo cerrado dentro del cual se generan valores.

- Recursos: Debe ser una cadena que identifique el número de instancias de los recursos que necesita la tarea para su ejecución, así como el número de instancias para cada uno. La sintaxis es la siguiente para una tarea que necesite instancias de *n* recursos:

  $$nombre_r_1 : instancias_r_1, ..., nombre_r_n : instancias_r_n$$

  Como se puede ver, la información de cada recurso se encuentra separada por comas, siendo esta información un par indicando el nombre y el número de instancias, separados por dos puntos.

- Probabilidad: Para los análisis de recursos se permite añadir la probabilidad de selección que tienen los flujos que salgan de compuertas exclusivas e inclusivas. Debe ser un número real en el intervalo [0,1] escrito en notación inglesa, es decir, utilizando el "." para separar la parte entera de la decimal.

La siguiente cuestión es una de las más importantes a la de hora de utilizar la aplicación: la diferencia entre los tipos de diagramas que cada análisis utiliza. Los análisis de procesos presentados en la iteración uno (tiempo, grado de paralelismo, model checking y liveness) deben ser utilizados con diagramas que solo tengan aquellas construcciones que se consideran en su implementación, que son los eventos de comienzo y fin, las puertas exclusivas, inclusivas y paralelas, las tareas y los flujos de secuencia (ambos con tiempo de ejecución determinista). Por ende, se **asume** que los diagramas a analizar cumplen este requisito.

Por su parte, los análisis de recursos (iteración dos) comprenden el subconjunto anterior además de tener mensajes, eventos, recursos y tiempos no deterministas asociados a las tareas y a los flujos de secuencia. Sin embargo, no se pueden utilizar los análisis anteriores con este tipo de diagramas.

Volviendo a la explicación de los elementos de la página, podemos ver el elemento lateral que se muestra a la izquierda una vez que el usuario pulsa el botón **Analyse Diagram**.

En este menú se muestran todos los análisis que se pueden realizar, los cuales se accionan con un click sobre su etiqueta. Además, en la esquina inferior izquierda, se puede ver un botón de ayuda que le explica al usuario como acceder a la información de cada análisis. Esta información, tal y cómo explica la ayuda, aparece al mantener el cursor sobre cada análisis por más de un segundo. Los análisis son los siguientes, separados por la iteración en los que se presentaron:

C.2.1. Análisis presentados en la primera iteración

- Análisis temporal: Calcula el tiempo de ejecución mínimo, máximo y medio del diagrama.
- Análisis del grado de paralelismo: Calcula el máximo número de nodos del proceso que se ejecutan en un mismo instante de tiempo.
- Model checking: Permite que el usuario pueda comprobar propiedades de lógica temporal lineal (LTL). Este tipo de análisis requiere que el usuario especifique dichas propiedades pues son dependientes del diagrama a analizar. Una vez el usuario pulse sobre el botón, se mostrará un formulario en el cual se debe introducir la propiedad. Además, se proporciona una ayuda que le indica al usuario cómo utilizar ciertas construcciones con las que puede, entre otras cosas, referenciar a los elementos del diagrama.
- Liveness: Permite comprobar que el diagrama está libre de deadlocks. En este contexto un deadlock es un estado no terminal del cual no se puede transitar a ningun otro estado. Por ende, si un diagrama cumple esta propiedad significa que termina o que contiene un bucle infinito. Ahora mismo el análisis está incompleto debido a que no se puede diferenciar los casos en los que el diagrama termina o los que contiene un bucle un infinito pero su distinción está propuesta para trabajos futuros.

C.2.2. Análisis presentados en la segunda iteración

Al igual que en el model checking, los dos análisis realizados en esta iteración requieren la especificación por parte del usuario de una serie de parámetros. Esto se realiza también por medio de dos formularios que se accionarán cuando el usuario pulse los botones en cuestión.

Estos formularios también presentan un botón de ayuda en la esquina inferior derecha que muestra información de ayuda para indicar qué información se debe aportar y cómo.

- Análisis de recursos: Genera una gráfica que muestra el número de instancias en uso en función del tiempo para cada uno de los recursos utilizados. Primero el usuario debe aportar el número de recursos a utilizar y el número de simulaciones concurrentes que quieren que se ejecuten. Tras esto, deberá especificar el nombre de cada recursos y el número de instancias a tener en cuenta por cada uno.

- Optimización de recursos: Emplea el algoritmo del descenso de gradiente para encontrar una combinación óptima de recursos. Igual que en el caso anterior, primero el usuario debe aportar el número de recursos a utilizar y el número de simulaciones concurrentes que quieren que se ejecuten. Tras esto, deberá especificar para cada recurso su nombre, el coste por hora, y el número mínimo y máximo de instancias a considerar. Estos límites definirán el intervalo de instancias para cada recurso que el algoritmo puede escoger como solución. Dicho intervalo comenzará en el límite inferior y aumentará de uno en uno hasta llegar al límite superior. Por ejemplo, para los límites 2 y 8, se considerará el intervalo: [2,3,4,5,5,6,7,8]. Se **asume** que el usuario no escogerá un límite inferior menor o igual que el superior.

El último elemento a considerar es el cuadro de salidas. Este se encuentra justo debajo del modeler y en él se mostrarán las gráficas y tablas con los resultados de cada análisis.

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga