Universidad de Cantabria

Facultad de Ciencias

GENERACIÓN AUTOMÁTICA DE VISUALIZACIONES PARA RESULTADOS DE ANÁLISIS DE TIEMPO REAL MAST

Automatic generation of results visualizations for MAST real time analysis

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Arturo Rodríguez Benito

Director: Alfonso de la Vega Ruiz

Co-Director: Juan Maria Rivas Concepcion

Julio - 2025

Agradecimientos

Me gustaría agradecer a Michael González Harbour, Alfonso de la Vega Ruiz y Juan Maria Rivas Concepcion por guiarme y ayudarme durante la realización de este proyecto. También quiero agradecer a mis padres por su apoyo.

Resumen

Las herramientas de análisis de tiempo real MAST generan sus resultados en un documento xmi que es poco legible al ojo humano. La herramienta MAST ofrece tablas para visualizar los resultados pero se encuentran en ventanas dispares y dependiendo del caso, no es la visualización óptima para interpretar los resultados.

En este proyecto implementamos una serie de visualizaciones a partir de los resultados MAST con el objetivo de mejorar la legibilidad de los datos. Para mejorar la legibilidad haremos uso de gráficas para presentar los datos al usuario de forma eficiente.

Aprovechando que los resultados MAST están contenidos en un modelo MAST-2 usamos Ingeniería dirigida por modelos, basada en Eclipse Modeling Framework (EMF), para generar las visualizaciones aplicando transformaciones al modelo de resultados. Las transformaciones que usamos hacen uso de lenguajes del framework Epsilon.

Las visualizaciones se muestran dentro del IDE Eclipse gracias al uso del Plugin Picto que nos permite mostrar las visualizaciones en una ventana de navegador dentro del entorno de Eclipse.

Palabras clave

Tiempo real, MAST, Ingeniería Dirigida por Modelos, Visualización, Generación de Código, Picto, Epsilon.

Abstract

MAST real time analysis suite generate their results in an xmi document that not easily readable for an user. The MAST suite offers tables to visualize the results but these tables are scattered across different windows and depending on the use case tables might not be the optimal visualization.

In this project we implement a series of visualizations from MAST results with the objective of improving data readability. In order to improve readability we use plots in order to efficiently provide the user with data.

Taking advantage of the fact that MAST results are included in a MAST-2 model we use Model-Driven Engineering based on Eclipse Modeling Framework (EMF) in order to generate the visualizations applying transformations to the results model. These transformations make use of languages from Epsilon framework.

The visualizations are shown inside Eclipse IDE thanks to the use of Picto's plugin which allows us to show the visualizations on a browser window inside Eclipse's enviroment.

Keywords

Real time, MAST, Model-Driven Engineering, Visualization, Code Generation, Picto, Epsilon.

Índice

Índice de figuras

1. Introducción

En esta sección se explica el objetivo del trabajo y varios conceptos necesarios para comprender el desarrollo y funcionamiento de la aplicación.

1.1. Sistemas de tiempo real

Un sistema de tiempo real es un sistema informático que ha de cumplir con requisitos temporales. Un sistema de tiempo real debe procesar y responder a solicitudes dentro de un tiempo preestablecido. Algunos sistemas de tiempo real tienen requisitos estrictos y otros no. También los hay con una mezcla de requisitos estrictos y no estrictos. Los requisitos estrictos deben cumplirse siempre, mientras que para los no estrictos se admite una cierta probabilidad de incumplimiento.

Las características de este tipo de sistemas les permiten interactuar con entornos físicos y poseen mecanismos que garantizan el cumplimiento de los requisitos de tiempo tales como el manejo de interrupciones, planificación en tiempo real, la sincronización de procesos y el manejo de tiempo de ejecución.

Normalmente un sistema de tiempo real está compuesto de una serie de hilos que se planifican independientemente llamados tareas. Las tareas pueden identificarse como periódicas, esporádicas o aperiódicas, en la Figura 1 se muestra un ejemplo de tarea periódica. Las tareas periódicas se ejecutan en intervalos regulares de tiempo. Las tareas esporádicas se activan a intervalos irregulares de tiempo pero hay un intervalo de tiempo mínimo entre activaciones. Las tareas aperiódicas se activan a intervalos irregulares de tiempo y no están limitadas en la frecuencia de activación.

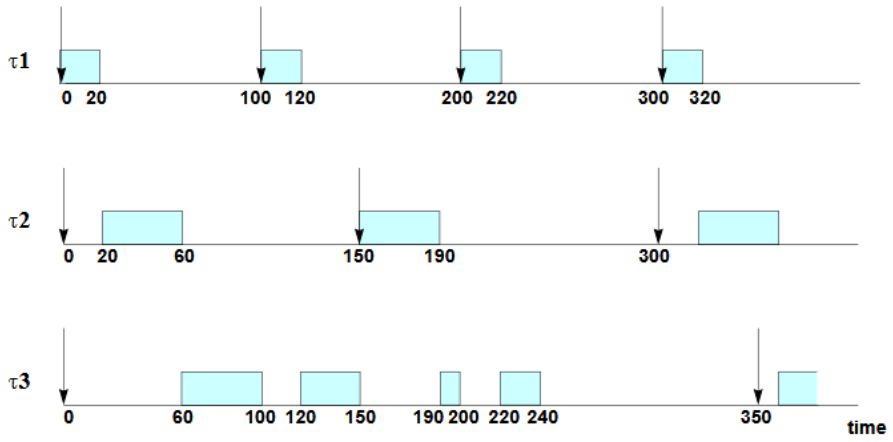

Figura 1: Representación de tres tareas periódicas, con periodos de 100, 150 y 3590 respectivamente [1]. La flecha indica el evento y las barras la ejecución

Cada tarea puede tener una prioridad que determina qué tarea accede a la CPU cuando más de una trata de ganar acceso a la vez, y también posee un tiempo de ejecución de peor caso representado por la letra C, que representa el intervalo máximo de tiempo (o una cota superior) que requiere la ejecución de la tarea cuando está sola en el sistema.

Una tarea puede depender de la finalización de otra para comenzar a ejecutarse. Una secuencia de tareas con relaciones de principio-fin forman un flujo.

Un flujo puede identificarse como lineal si no tiene bifurcaciones a dos o más tareas o no lineal si posee bifurcaciones. Los flujos poseen un plazo en el que su ejecución ha de completarse esto es, el tiempo máximo permitido entre que se activa la primera tarea del flujo y finaliza la ejecución de la ultima tarea. El tiempo de respuesta de una tarea se define como el tiempo máximo que trascurre desde que se activa la tarea hasta que finaliza su ejecución. Este tiempo incluye su tiempo de ejecución de peor caso (C), tiempo de interferencia, y el tiempo de bloqueo. El tiempo de respuesta de un flujo es por lo tanto el tiempo total entre la activación de la primera tarea, y la finalización de la última tarea en el flujo.

El tiempo de bloqueo de un flujo se representa con la letra B e incluye el tiempo que las tareas del flujo pasan esperando a ganar acceso a la CPU ya que una tarea de menor prioridad impide su ejecución, normalmente esto ocurre debido a recursos compartidos u otros mecanismos de sincronización.

El tiempo de interferencia es representado por la letra I y representa el tiempo que las tareas del flujo pasan esperando a tomar control de la CPU debido a que una tarea de mayor prioridad la tiene ocupada.

Otros conceptos de sistemas de tiempo real que conviene conocer son la holgura, también conocido como Slack, y el jitter. La holgura es el margen en que los tiempos de ejecución de peor caso (C) pueden ser incrementados antes de que los plazos dejen de cumplirse. El jitter o fluctuación en castellano es la variabilidad de los tiempos de respuesta cuando se ejecutan las tareas.

1.2. Herramientas MAST

En esta sección se introducen las herramientas MAST [2], su funcionamiento y su relación con el proyecto. También se detallan los modelos usados por MAST.

MAST (Modeling and Analysis Suite for Real-Time Applications) es un suite de herramientas de código abierto desarrollado por el grupo de Ingeniería de Software y Tiempo Real de la universidad de Cantabria. MAST permite modelar un sistema de tiempo real incluyendo una variedad de requisitos temporales y analizar los tiempos de respuesta de peor caso para verificar si el sistema cumple dichos requisitos y cómo de lejos o cerca está de cumplirlos.

Entre las características de MAST se incluye la descripción basada en modelos, siendo capaz de usar modelos como entrada y salida, de acuerdo a la metodología de desarrollo basado en modelos. Estos modelos son conformes a los respectivos metamodelos de entrada y salida de MAST.

En el modelo de entrada se describe la arquitectura del sistema indicando el número de CPUs y los mecanismos de arbitraje de los mismos. También posee las tareas y flujos que han de ser ejecutados en el sistema junto a sus especificaciones temporales, prioridades de ejecución y recursos a compartir. Las tareas se describen junto a sus tiempos de ejecución de peor caso y los flujos se definen por su periodo, plazo, y las tareas que lo componen. Aquí falta definir el término plazo (de tarea y de flujo) en la sección 1.1

En el modelo de salida se recogen los resultados del análisis temporal de MAST incluyendo las distintas métricas de cada tarea y flujo así como la utilización y holgura del sistema. Las principales métricas que se recogen de cada flujo son el tiempo de bloqueo, tiempo de respuesta de peor caso y el Jitter

Para visualizar mejor el funcionamiento de la herramienta presentamos un ejemplo. En el siguiente sistema de tiempo real tenemos dos flujos, op y oc, que comparten un recurso (Figura 2).

Ambos tienen un periodo de diez unidades y un plazo de 20.

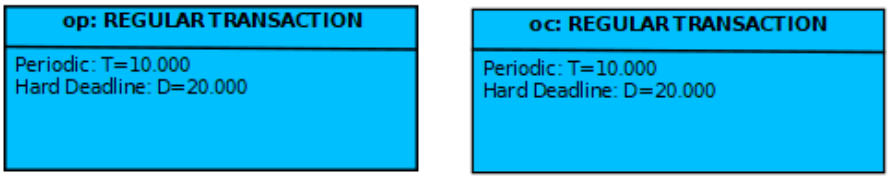

Figura 2: Flujos en el sistema de ejemplo

En el flujo op las dos operaciones, una operación es un elemento del modelo que especifica el C de las tareas, tienen un tiempo de peor caso 3 y 2 unidades respectivamente. En oc los tiempos de peor caso son 1,5 y 3. En ambos flujos la primera operación toma el recurso compartido durante su ejecución y lo libera al finalizar (Figura 3).

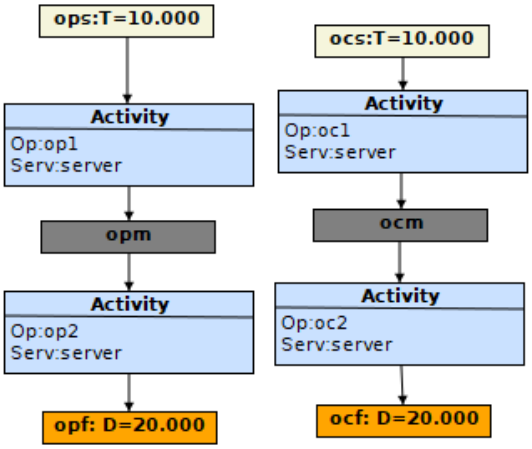

Figura 3: Las operaciones del sistema de ejemplo, op izquierda y oc derecha

A continuación la herramienta MAST realiza un análisis de la planificabilidad de este sistema de ejemplo dando los resultados mostrados en las siguientes figuras.

Primero se nos presenta con la planificabilidad general del sistema resultando en un sistema planificable (Figura 4) con una utilización del 95 % dejando una holgura del 5,32 %.

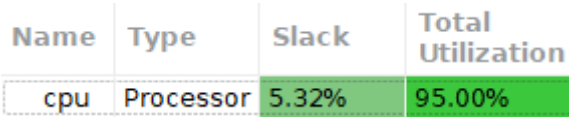

Figura 4: Planificabilidad general del ejemplo

En la siguiente ventana podemos encontrar la holgura de cada flujo siendo 9,36 % para op y 10,55 % para oc (Figura 5).

Pulsar la figura de una carpeta a la derecha de los resultados abre una nueva ventana que proveerá información específica de cada flujo (Figura 6). Aquí podemos ver los tiempos de peor caso del flujo completo junto al plazo del flujo. El flujo oc tiene tiempo de respuesta de peor caso

Name	Type	Slack	Timing Results
op	Regular_Transaction	9.38%	
oc	Regular_Transaction	10.55%	

Figura 5: Holgura de cada flujo de ejemplo

de 9,5 unidades mientras que el flujo op posee un tiempo de respuesta de peor caso de 6.5 unidades y un tiempo de bloqueo de 1.5 unidades.

Transaction <View All> View All

Global Response Times | Output Jitters | Blocking Times | Local Response Times | Suspensions | Local M

Transaction	Event	Referenced Event	Best Response	Worst Response	Hard Deadline
op	opf	ops	0.000	6.500	20.000
oc	ocf	ocs	0.000	9.500	20.000

Figura 6: Métricas de los flujos de ejemplo

En las figuras de ejemplo se puede ver la interfaz con la que MAST presenta los datos que consiste en varias tablas con los valores de las métricas separadas en varias pestañas y ventanas.

En este proyecto proponemos crear una serie de nuevas vistas, que faciliten la visualización y comparación de los resultados. Las tablas de la herramienta MAST que hemos mostrado están basadas en GTK [3]. En vez de crear una herramienta adhoc, nuestras vistas están integradas en un entorno de ingeniería dirigida por modelos, como un plugin del IDE de Eclipse.

1.3. Ingeniería dirigida por modelos

La ingeniería dirigida por modelos(Model-Driven Engineering, o MDE por sus siglas en ingles) es un paradigma de desarrollo de software que trata de aumentar la abstracción del desarrollo haciendo uso de modelos.

En vez de desarrollar un programa de forma directa para resolver un problema se crea uno o varios modelos del problema, según las necesidades. A partir de estos modelos se pueden generar programas, pero también se pueden transformar estos modelos a otros modelos o representaciones. Trabajar con un modelo mejora la comprensión del desarrollo por parte del desarrollador, facilita la manipulación de sistemas complejos y da más opciones para automatismos.

Los modelos utilizados deben ser conformes a un metamodelo que describe el lenguaje que usa el modelo para representar su información, conociendo este metamodelo se puede manipular el modelo de varias formas, por ejemplo se puede transformar a otro modelo conforme a un metamodelo distinto o generar código u otra información a partir del mismo.

MAST utiliza dos metamodelos [4], un metamodelo que permite modelar los sistemas de tiempo real y un metamodelo que permite representar los resultados de los análisis. MAST tiene actualmente dos versiones, MAST-1, que hemos explicado en la sección 1.2, es la versión que la herramienta usa en la actualidad y que será remplazada en un futuro por MAST-2, una nueva versión de MAST que se integrará dentro de un entorno basado en modelos utilizando la tecnología de Eclipse EMF. MAST-2 se alinea con la nomenclatura de MARTE UML [5] y añade nuevos elementos al modelo tales como swicthes de red y planificación por particionado temporal.

Para manipular un modelo se usa una transformación que tiene como entrada un modelo conforme a un metamodelo y puede tener como salida otro modelo conforme a un metamodelo, código u otra información. Las transformaciones se implementan usando un lenguaje de transformación que indica qué elementos del modelo se traducen y cómo se realiza la traducción.

Durante el trabajo se usarán los metamodelos de MAST-2 que permitirán generar vistas a partir de los modelos de sistemas y resultados generados por MAST.

1.4. Motivación

El principal problema que nos a impulsado a realizar este proyecto es la falta de legibilidad en los archivos de resultados MAST-2 generados por MAST.

Los resultados son generados en ficheros XMI. Para ilustrar la legibilidad de estos tenemos el siguiente ejemplo de un modelo que usamos para gráficas de ejemplo más adelante:

Listing 1: Ejemplo de archivo de resultados MAST-2

```
<?xml version="1.0" encoding="UTF-8"?>
<mast2_res:Real_Time_Situation_Results
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mast2="http://mast.unican.es/ecoremast/Mast2"
    xmlns:mast2_res="http://mast.unican.es/ecoremast/Mast2_Results"
    xsi:schemaLocation="http://mast.unican.es/ecoremast/Mast2
       /mastresults/Mast2.ecore
       http://mast.unican.es/ecoremast/Mast2_Results
       /mastresults/Mast2_Results.ecore"
    Model_File="caseva_example2.xmi"
    Model_Date="2000-01-01T00:00:00"
    Generator_Tool="MAST Schedulability Analysis, version 1.5.2.0"
    Generation_Profile="mast_analysis default -v -c -p -d
       /home/michael/tmp/caseva_example2.xmi -s
       /home/michael/tmp/caseva_example2.txt
       /home/michael/tmp/caseva_example2.out.xmi"
    Generation_Date="2023-10-10T11:34:48"
    Slack="101.56">
  <Mast_Model
    href="caseva_example2.xmi#caseva"/>
  <Element_List
      xsi:type="mast2_res:End_To_End_Flow_Result"
      Slack="221.48">
    <Model_Elem
      xsi:type="mast2:Regular_End_To_End_Flow"
      href="caseva_example2.xmi#e2e_servo_control"/>
    <Timing_Result
```

```
        Event_Name="o1"
        Num_Of_Suspensions="0"
        Worst_Blocking_Time="135.000">
      <Worst_Global_Response_Time
          Referenced_Event="e1"
          Value="1420.00"/>
      <Best_Global_Response_Time
          Referenced_Event="e1"
          Value="0.000"/>
      <Worst_Output_Jitter
          Referenced_Event="e1"
          Value="1420.00"/>
    </Timing_Result>
  </Element_List>
</mast2_res:Real_Time_Situation_Results>
```

En toda esta sección solo se representa un único flujo. La primera mitad es en su mayoría metadatos sobre el formato y creación del archivo y en la segunda mitad que contiene la información del flujo extraer los datos a simple vista es posible pero requiere esfuerzo localizar la información relevante.

Debido a esta falta de legibilidad creemos que proporcionar visualizaciones a los usuarios puede facilitar el trabajar con los resultados de MAST.

1.5. Objetivos

El principal objetivo del proyecto es suministrar al usuario una serie de vistas generadas usando ingeniería dirigida por modelos. Estas vistas deberían facilitar la lectura y comprensión de los resultados de análisis de sistemas de tiempo real generados por la herramienta MAST.

La herramienta MAST genera dos modelos MAST-2, uno de entrada que describe el sistema a analizar y uno de salida con el resultado del análisis. Se puede aplicar una transformación sobre estos modelos para obtener las vistas.

Las vistas generadas deberán transmitir la información de los resultados con eficacia y la generación de las mismas tendrá que realizarse de forma rápida y cómoda para el usuario.

2. Tecnologías

En esta sección se presentan las tecnologías usadas dentro del desarrollo de la aplicación.

2.1. Eclipse modeling framework

Eclipse Modeling Framework (EMF) [6] es un framework de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo estructurado de datos.

EMF provee una serie de herramientas para producir clases Java a partir de un modelo especificado en XMI así como una serie de clases adaptadoras que permiten visualizar y editar, usando comandos, un modelo.

Las piezas básicas de EMF son las siguientes:

1. EMF.core incluye el meta-metamodelo Ecore que permite describir metamodelos. EMF core también da soporte en tiempo de ejecución, la persistencia de los modelos se consigue con serialización XMI y también provee una API para manipular objetos EMF.

2. EMF.edit incluye clases genéricas para construir editores para modelos EMF, estas clases incluyen clases proveedoras de contenido y etiquetas que permiten a los modelos EMF ser mostradas en visualizaciones y tablas de propiedades, también incluye un framework de comandos genéricos para construir editores que soporten hacer y deshacer.

3. EMF.codegen permite generar todo lo necesario para construir un editor de modelos EMF, incluye una GUI desde la que indicar opciones de generación e invocar generadores.

El uso de EMF es necesario para manejar los modelos de MAST ya que estos se construyen usando EMF, por lo que es necesario tanto para su visualización como para la generación y transformación de los mismos.

2.2. MAST-2

MAST-2 [4] es una evolución del modelo MAST basada en EMF. MAST-2 se alinea con la nomenclatura de MARTE UML [5] y añade nuevos elementos al modelo como switches de red y planificación por particionado temporal.

En el momento en el que se realizó el proyecto, MAST-2 todavía no se había implementado en las herramientas MAST, pero nos decantamos por esta versión para que esté preparado para versiones futuras. Para obtener los modelos MAST-2 se han transformado resultados en modelos MAST con una herramienta externa.

MAST-2 define dos metamodelos. Uno permite describir modelos de entrada que definen un sistema de tiempo real a analizar y el otro permite describir modelos de resultados obtenidos tras un análisis.

Puesto que nuestras visualizaciones solo representan los flujos, no necesitamos hacer uso de todos los objetos del modelo. A continuación describimos los elementos relevantes del metamodelo para la realización del proyecto.

La clase que representa los flujos se llama “End_To_End_Flow” y está representada tanto en el modelo de MAST2 base como en el modelo de resultados. Ambas versiones contienen distinta

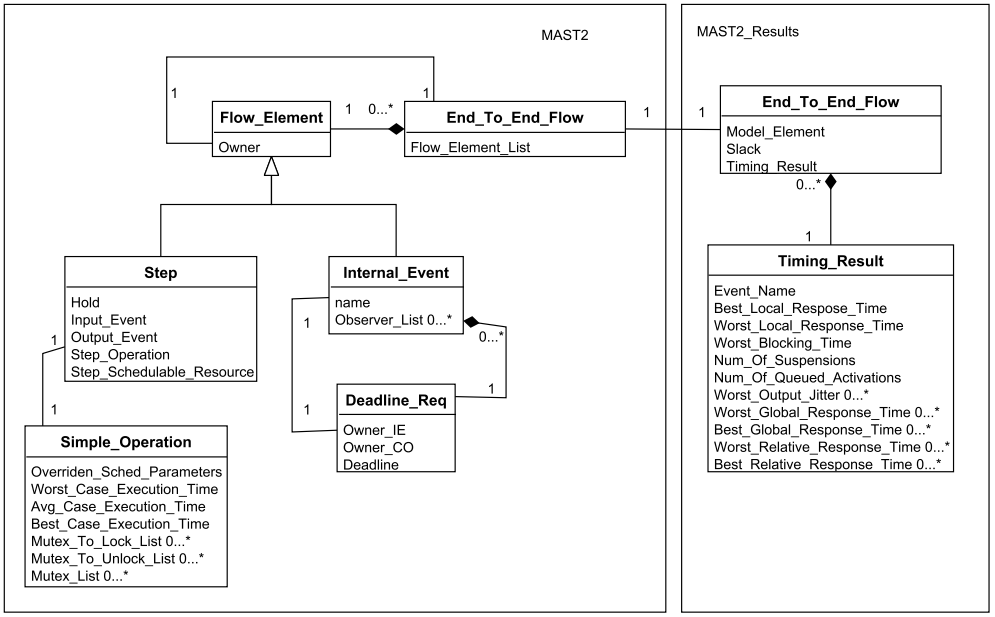

Figura 7: Diagrama de clases de los metamodelos MAST2

información que queremos representar en las visualizaciones. La clase "End_To_End_Flow" en el modelo de resultados posee una referencia a su contraparte del modelo base lo que nos facilita acceder a la información.

La estructura general de las clases con se muestra en el diagrama de la Figura 7. El diagrama solo muestra las clases relevantes para los datos que queremos extraer de los modelos.

En el modelo base de MAST2 necesitamos obtener los plazos del flujo. Estos se encuentran en la clase "Deadline_Req" que se recogen en una lista en la clase de eventos internos del flujo "Internal_Event". También es necesario obtener el tiempo de ejecución de peor caso (C) que se encuentra en la operación "Simple_Operation" que a su vez es contenida en la clase "Step". Tanto "Internal_Event" como "Step" heredan de la clase "Flow_Element" y todos los elementos del flujo son almacenados en la lista "Flow_Element_List".

El resto de datos se encuentran en el modelo de resultados. La clase "End_To_End_Flow" en este modelo contiene la hogura y una lista de "Timing_Result", sin embargo como estamos trabajando con flujos simples esta lista solamente contendrá un solo objeto. En la clase "Timing_Result" encontramos el resto de la información relevante, incluyendo el tiempo de respuesta de peor caso, el tiempo de bloqueo y el jitter.

2.3. Epsilon

Epsilon [7] es una familia de lenguajes de scripting y herramientas para la automatización de tareas de ingeniería de software dirigida por modelos incluyendo generación de código, transformación entre modelos, validación de modelos y visualización de modelos. Epsilon funciona de forma directa con EMF, UML, SimuLink y XML entre otros tipos de modelos. Epsilon provee soporte especialmente fuerte para Eclipse y EMF pero también funciona con otros editores y modelos.

Todos los lenguajes de Epsilon están construidos a partir de un lenguaje de expresión común que permite reusar código entre distintos lenguajes, denominado Epsilon Object Language.

A continuación se detallan los principales lenguajes de Epsilon que se han utilizado durante la realización del proyecto.

2.3.1. Epsilon Object Language

El lenguaje base de Epsilon es EOL (Epsilon Object Language), es un lenguaje de estilo imperativo similar a Java/JavaScript e incluye capacidades de query de modelos como las de OCL (Object Constraint Launguage), pero también incluye estructuras propias de lenguajes imperativos como variables, condicionales y bucles. A continuación un ejemplo de EOL sacado del playground de Epsilon [8]:

Listing 2: Muestra de código EOL

```
// For every task in the model
for (t in Task.all) {
        // Print the title and the total person-months of the task
        (t.title + ": " + t.getTotalEffort()).println();
}

// Count the tasks that are undertaken by a single person
Task.all.select(t|t.effort.size() = 1).size().
        println("One-person tasks: ");

// Returns the total person-months for a task
operation Task getTotalEffort() {
        return self.effort.
                collect(e|self.duration*e.percentage/100.0).sum();
}
```

Todos los lenguajes Epsilon funcionan en conjunto con una capa de conectividad de modelos que les permite interactuar con los modelos sin importar la tecnología de modelado que utilicen.

Aparte de EOL Epsilon provee una serie de lenguajes más específicos dependiendo de la tarea a realizar.

2.3.2. Epsilon Generation Language

EGL (Epsilon Generation Lenguaje), es un lenguaje de transformación de modelo a texto, incluyendo código. EGL es un lenguaje basado en plantillas por lo que los programas EGL se asemejan al texto que generan.

El código de un programa EGL se divide en secciones estáticas que no cambian al ejecutarse el programa y secciones dinámicas que son remplazadas por el texto generado al ejecutarse el programa, las secciones dinámicas son indicadas usando el par de etiquetas [% %] dentro de las cuales se usa código EGL basado en EOL. Ejemplo de EGL sacado del playground de Epsilon [9]:

Listing 3: Muestra de código EGL

```
[*For every person*]
[%for (p in Person.all){%]

[*Generate a <h1> with the name of the person*]
<h1>[%=p.name%]'s Tasks</h1>

[*Generate a table for the person's tasks*]
<table>
[*For every task*]
[%for (t in p.getTasks()){%]
        [*Generate a row with the title of the task*]
    <tr>
        <td>[%=t.title%]</td>
    </tr>
[%}%]
</table>

[%}%]
```

2.3.3. EGL Co-Ordination Language

EGX (EGL Co-Ordination Language) es un lenguaje basado en reglas que permite automatizar la ejecución parametrizada de transformaciones de modelo a texto aunque fue construida principalmente para funcionar con EGL otros lenguajes basados en plantillas también funcionan con EGX.

El principal uso de EGX es invocar la misma plantilla múltiples veces, con distintos parámetros de entrada, siguiendo ciertas reglas de ejecución o especificando el objetivo de la generación según el tipo o el elemento. Esto permite, por ejemplo, usar una plantilla sobre una serie de elementos de cierto tipo dentro de un modelo para generar una serie de documentos de texto.

Un módulo EGX se compone de una serie de reglas con nombre que pueden tener bloques pre y post que permiten usar código para realizar tareas antes y después de ejecutar una regla. Un módulo EGX ejecuta todas sus reglas en el orden en el que están definidas.

Cada regla puede tener una serie de bloques de componentes que son opcionales, los más comunes son: "template" que indica la plantilla a usar, "transform" que indica uno o más elementos sobre los que aplicar la regla y "parameters" que indica los nombre de variables y sus valores para ser pasados a la plantilla para su ejecución. Ejemplo de EGX sacado del playground de Epsilon [10]:

2.3.4. Pinset

El lenguaje Pinset [11] ofrece una sintaxis especifica que extrae datasets con forma de tabla de modelos, Pinset se utiliza principalmente para extraer datos para data mining o machine learning.

Listing 4: Muestra de código EGX

```
// For every person in the model
rule Person2TaskList
    transform p : Person {

    // run the EGL template below
    template: "template.egl"

    // and generate a HTML page
    // containing its output
    target: "gen/" + p.name + ".html"
}
```

En un documento Pinset se indica el nombre del dataset y los elementos a extraer del modelo y a continuación se indican una serie de columnas con el nombre de las mismas y el código que extra dato en cada. Ejemplo de Pinset sacado del playground de Epsilon [12]:

Listing 5: Muestra de código Pinset

```
dataset tasks over t : Task {
    properties [title, start, duration]

    // Assuming each person provides at most one effort to a task
    column num_participants : t.effort.size()
    column total_effort : t.effort.collect(e | e.percentage).sum()
}
```

2.4. Picto

Picto [13] es una vista de Eclipse para visualizar modelos gracias a la transformación modelo a texto, en este caso a SVG o HTML. La visualización en Picto ocurre en un navegador embebido en Eclipse y por lo tanto se puede hacer uso de cualquier tecnología web para las visualizaciones. Las visualizaciones de Picto son de solo lectura y no permiten editar los modelos.

Para usar Picto es necesario tener un modelo que visualizar y una transformación modelo a texto, a continuación se debe conectar ambas mediante un documento Picto con el nombre del modelo dentro del cual se indicara la transformación a ejecutar y su formato.

Se proporcionaran ejemplos de Picto en las siguientes secciones junto con más detalles.

2.5. Plotly

Plotly [14] es una librería declarativa para trazado, principalmente de gráficas. Plotly incluye más de 40 tipos distintos de gráficas y funciona sobre JavaScript.

Para crear una gráfica con Plotly solo es necesario llamar a la función “newPlot” y pasarle como argumentos el nombre de la gráfica, los datos, la disposición de los elementos y de manera opcional una configuración.

Los datos están formados por varias trazas conteniendo los valores a representar y opcionalmente los nombres y tipo de datos, la disposición indica el tipo de gráfico a construir y de manera opcional se puede indicar el titulo con tipo de letra, margenes e incluso dibujar formas encima del gráfico.

En la configuración se indican otro tipo de opciones con efectos varios, la que se usa durante el proyecto es "responsive" ya que al activarla obliga al gráfico a adaptarse al tamaño de ventana del navegador lo que permite a la gráfica ajustarse al tamaño de la vista.

A modo de ejemplo Plotly proporciona en su web varias visualizaciones con las que mostrar qué clase de implementaciones se pueden realizar con la librería [15].

3. Análisis de requisitos

En esta sección se especifican los requisitos de la aplicación junto al proceso a través del cual se han desarrollado dichos requisitos.

El principal objetivo de la aplicación es generar una serie de vistas a partir de los resultados MAST2 que se mostrarán dentro del IDE de Eclipse usando Picto.

Cada modelo de resultados MAST está formado por varios flujos de eventos que incluyen métricas temporales a partir de las cuales se desarrollaron tres visualizaciones distintas: una gráfica general que muestre todos los flujos, una gráfica por cada flujo que permita centrarse en el flujo específico y una tabla que muestre las medidas de todos los flujos.

En las vistas se representan flujos de resultados lineales, los flujos no lineales son más complejos de representar y por lo tanto no se han considerado dentro de los objetivos de este proyecto. Cabe destacar que dentro de los resultados MAST no se especifican las unidades de tiempo que usan los resultados y por lo tanto las gráficas y métricas de estas vistas no tienen unidades.

Inicialmente los requisitos fueron especificados de manera conjunta por los participantes de este proyecto. Más adelante en el desarrollo de la aplicación se realizó una reunión con expertos del grupo de sistemas de tiempo real para enseñarles la aplicación con el objetivo tanto de validar las primeras versiones como de captar requisitos extra.

3.1. Gráfica resultados

A continuación usamos la Figura 8 como ejemplo para explicar el diseño de la gráfica de resultados, este ejemplo representa un sistema CASEVA, un robot para soldado de piezas, con cinco flujos: message_logger, reporter, light_manager, trajectory_planning y servo_control. El sistema no es planificable ya que el flujo message_logger tiene un tiempo de interferencia tan grande que se considera infinito.

En esta vista se muestran los resultados MAST divididos en varios flujos, cada flujo es representado por una barra que indica la duración completa del flujo de inicio a fin. La barra se divide en secciones de distintos colores para indicar los diferentes tipos de tiempos que influyen en el tiempo de respuesta de principio a fin. Estas secciones son: C (Tiempo de ejecución de peor caso) en color azul es el tiempo que el flujo realiza trabajo útil tomando como referencia el peor caso, B (Tiempo de bloqueo) en naranja es el tiempo que el flujo pasa esperando a tener acceso a la CPU debido a la ejecución de tareas de prioridad inferior y I (Tiempo de interferencia) en verde es el tiempo que el flujo pasa sin realizar trabajo útil debido a la interferencia de otros flujos de prioridad superior, para más detalle explicación en 1.2.

La gráfica se centrará en el flujo de mayor duración y si uno o más flujos tiene una duración infinita, lo cual indica que no es planificable, se indicará con una flecha que su duración se sale de los límites de la gráfica y se centrará en el siguiente flujo planificable de mayor duración. Este es el caso de message_logger en el ejemplo con la gráfica, ya que su duración es infinita se indica con la flecha y la gráfica se centra en reporter.

En esta gráfica cada sección de tiempo deberá representarse con un color que permita distinguirlas entre sí de forma clara, el nombre de cada flujo se indicará en el eje y mientras que la duración se representará en el eje x, también debe ser posible hacer zoom dentro de la gráfica para poder visualizar flujos de magnitud menor y se deberá poder mostrarla métrica especifica

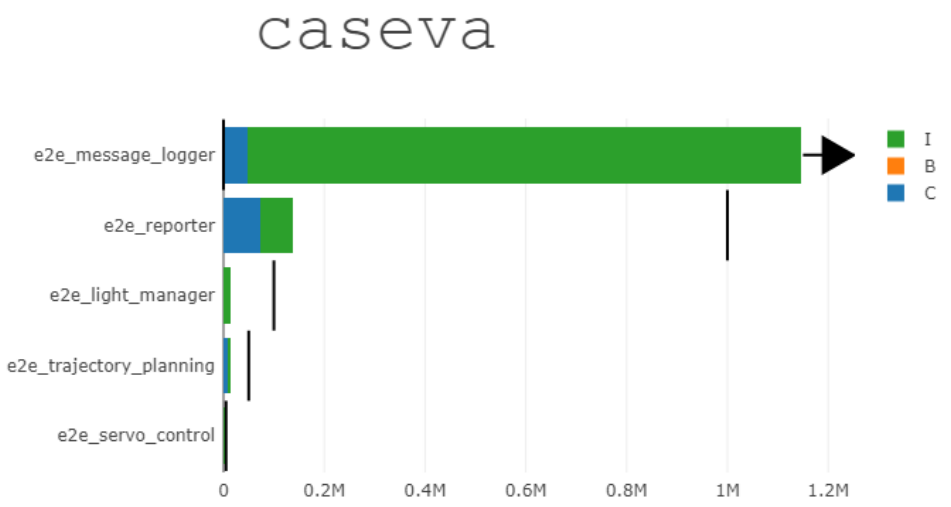

Figura 8: Ejemplo gráfica resultados

que representa cada sección de una barra al mantener el cursor sobre la misma. Tras la reunión con expertos se decidió representar los plazos de cada flujo como una linea vertical en la posición adecuada, si la barra de un flujo sobrepasa la linea esto indica que no está cumpliendo el plazo como es el caso de message_logger en el ejemplo.

3.2. Gráficas Flujos

La diferencia de magnitud entre los flujos puede dificultar la visualización de flujos cortos y, aunque se puede hacer zoom, puede hacer difícil distinguir los detalles del mismo. Para remediar esto generamos una vista específica a cada flujo cuyo zoom este ajustado al plazo del mismo.

A continuación en la Figura 9 se muestra la siguiente gráfica de ejemplo, también del mismo modelo que el caso anterior, en este caso está centrada en el flujo light_manager, los flujos message_logger y reporter duran más que el plazo de light_manager por lo que parte de su tiempo de interferencia queda fuera de la gráfica.

Estas vistas son casi idénticas a la vista de resultados con la diferencia que el zoom de la gráfica se ajusta al plazo del flujo, con el plazo permaneciendo dentro de la vista y las barras de flujos más largas extendiéndose más allá del límite de la vista sin mostrarse las flechas que indican los flujos no planificables, esto permitirá visualizar mejor la diferencia en magnitud entre flujos.

3.3. Tabla resultados

En la Figura 10 se muestra la tabla generada a partir del modelo caseva de los ejemplos anteriores.

Obtener los valores exactos de cada métrica a partir de las gráficas puede ser complicado, por ello se incluirá una última vista de tabla en la que se incluirán los valores numéricos de cada métrica

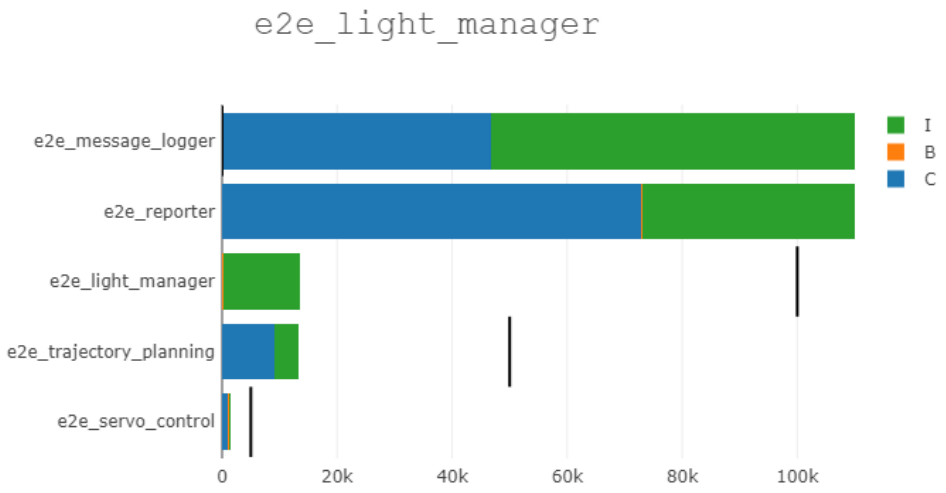

Figura 9: Ejemplo gráfica flujos

event	slack	jitter	C	B	I	deadline
e2e_servo_control	221.48	1420.0	1080.0	135.0	205.0	5000.0
e2e_trajectory_planning	264.84	13240.0	9045.0	135.0	4060.0	50000.0
e2e_light_manager	4602.3	13564.0	119.0	135.0	13310.0	100000.0
e2e_reporter	656.64	137614.0	72952.0	79.0	64583.0	1000000.0
e2e_message_logger	35216.4	1.0E100	46820.0	0.0	1.0E100	

Figura 10: Ejemplo tabla de resultados

que aparece en las gráficas (C,B,I y plazo) así como otras métricas mas difíciles de representar en una gráfica tales como la holgura y el jitter de cada flujo.

Cabe destacar que en la tabla de ejemplo message_logger aparece sin un plazo, esto representa un plazo infinito.

4. Metodología

En esta sección se profundiza en los métodos y pasos seguidos durante el desarrollo de la aplicación exponiendo el razonamiento detrás de las decisiones tomadas. El proyecto contó con una fase inicial de análisis y selección de librerías gráficas, sobre las que posteriormente se desarrollaron las visualizaciones. A continuación se describe cómo se realizaron ambas fases.

4.1. Análisis de librerías gráficas

El objetivo del análisis fue encontrar una o varias librerías gráficas que facilitasen visualizar gráficos de barras para uso en las vistas. Estas librerías tenían que cumplir una serie de requisitos que están descritos en la secciones 3.1 y 3.2. Los principales son que se muestre información al mantener el cursor sobre los elementos, ser capaz de hacer zoom en tareas que puedan tener un tiempo menor y permitir configurar las leyendas y colores.

Al principio del desarrollo se consideraron dos librerías gráficas para la realización de la aplicación, Vega [16] y Chart.js [17]. Más adelante, durante el desarrollo encontramos la librería Plotly que se ajustaba mejor a nuestros requisitos. Pueden verse descripciones más detalladas en sección 2.5.

Vega es un lenguaje declarativo que permite la creación de visualizaciones cuya apariencia e interactividad se describen en formato JSON [18]. Esta librería permite el uso de una gran variedad de visualizaciones y también permite la implementación de interactividad dentro de las visualizaciones.

La librería Vega fue considerada debido a la gran variedad de características que ofrecía, sin embargo fue descartada antes de empezar el desarrollo en favor de Chart.js debido a la complejidad que introducía su uso. Esta complejidad se debe a la gran variedad de visualizaciones y características que soporta Vega, la mayoría de las cuales son innecesarias para el desarrollo de la aplicación.

Chart.js es una librería gráfica que se centra en visualizaciones basadas en gráficas, ofreciendo los tipos de gráficas mas comúnmente usados. Esta librería es bastante sencilla y fácil de usar en comparación con Vega e incluye todas las funcionalidades necesarias para el desarrollo de la aplicación, algunas como el zoom mediante plugins, incluyendo soporte limitado de interactividad aunque la implementación de interactividad añade bastante complejidad al código.

La librería plotly.js es similar a Chart.js pero ofrece más funcionalidades manteniendo la simplicidad de uso. La funcionalidad que consideramos más atractiva de plotly.js es que integra interactividad limitada en las visualizaciones sin necesidad de manejar plugins o implementaciones complejas. Usando esta librería se puede cambiar la configuración de una visualización para permitir ajustar su tamaño y hacer zoom, así como segregar datos. Se puede hacer click en la leyenda de los elementos para activar y desactivar su visualización en la gráfica y mostrar información adicional al mantener el cursor sobre un elemento.

Al final decidimos hacer uso de plotly.js debido a que ofrece todas las funcionalidades necesarias para el desarrollo de la aplicación manteniendo al mismo tiempo facilidad de uso.

4.2. Proceso de desarrollo de visualizaciones

Durante el desarrollo de las visualizaciones de Picto seguimos un proceso incremental debido a la complejidad accidental que supone probar las visualizaciones en Picto ya que se trata de un plugin de eclipse. Este proceso incremental se desarrolló en las siguientes fases.

4.2.1. Fase 1: visualizaciones

Las visualizaciones comenzaron su desarrollo dentro de un documento html de prueba usando unos datos predeterminados. Se realizó la visualización de una gráfica de resultados para comprobar cómo funcionaban las librerías que se estaban considerando para el desarrollo.

Este documento de prueba consiste de un fichero html sencillo que contiene un script en javascript [19] que describe los datos a partir de los cuales se genera la gráfica, la configuración de la misma y la llamada a la librería que la genera.

Una vez elegida la librería se continuó usando el documento html de prueba para seguir el desarrollo de las visualizaciones, implementando las gráficas de resultados de forma manual usando datos estáticos y cumpliendo con los requisitos descritos en las secciones 3.1 y 3.2.

4.2.2. Fase 2: generación de código

Una vez completadas las visualizaciones en el documento html de prueba, este se usa como base para el desarrollo de las transformaciones que han de generar las vistas en la versión final de la aplicación.

Estas transformaciones hacen uso del lenguaje de lenguaje de generación de código de Epsilon EGL [20] para generar código html a partir de un modelo de resultados.

Para desarrollar la transformación primero se identifican las secciones dinámicas de la visualización que varían entre modelos, estas secciones se reemplazan por codigo EGL que carga información a partir del modelo. Estas transformaciones pueden ejecutarse pasándolas un modelo como entrada para generar un documento html que puede ser evaluado, para ello es necesario un fichero EGX que se encarga de coordinar la ejecución de las transformaciones.

En esta fase también se desarrolló la vista de tabla usando el lenguaje de extracción de datos Pinset [21], esta vista es bastante sencilla por lo que simplemente se implementó un archivo Pinset que se coordina mediante el mismo fichero EGX que las gráficas.

Durante el desarrollo de esta fase y anteriores hicimos uso de la librería chart.js para las gráficas. En esta fase decidimos usar la librería Plotly y se repitió la fase 1 con Plotly. Por suerte la forma en la que lo datos son proporcionados es bastante similar en ambas librerías lo que facilitó la transición.

4.2.3. Fase 3: integración con Picto

Con las transformaciones terminadas se procedió a integrarlas con Picto, haciendo uso de un fichero Picto mediante el que se puede conectar un modelo con el fichero coordinador EGX de tal manera que al abrir la ventana de Picto dentro de Eclipse se generaran las vistas de forma dinámica. Mientras se tenga el modelo en la pestaña activa mostrarán las visualizaciones. En secciones posteriores se explicaran más detalles sobre Picto.

5. Diseño

En esta sección se describe el diseño general de la aplicación incluyendo esquemas y/o diagramas de su funcionamiento.

La aplicación estará integrada como un plugin de eclipse, la gráficas generadas serán mostradas dentro de una ventana de Picto, ejemplo en Figura 11.

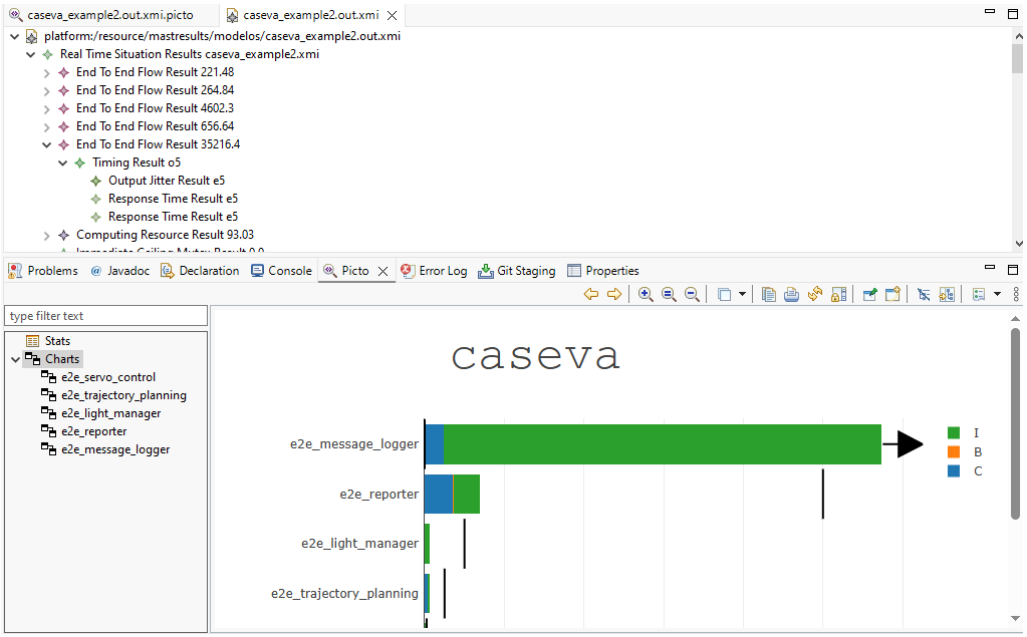

Figura 11: Ejemplo de aplicación en ventana Picto

En la mitad superior está abierto el modelo de resultados a partir del cual se generan las vistas, abrir un modelo diferente generara vistas para el mismo de forma automática. En la mitad inferior esta la ventana de Picto donde se muestran las visualizaciones.

En el lado izquierdo de la ventana de Picto se encuentra el indice de vistas, esta aplicación tendrá una vista de tabla nombrada en el ejemplo como “Stats” y corresponde al ejemplo de la sección 3.3. Una gráfica de resultados nombrada como “Charts” en el ejemplo y corresponde al ejemplo de la sección 3.1. Dentro del indice de la gráfica de resultados se anidaran las gráficas de flujos cada una con el nombre de su flujo en el ejemplo el flujo light_manager se corresponde con la gráfica mostrada en la sección 3.2.

El lado derecho de la ventana funciona como una navegador web en el que se mostraran las vistas, en el ejemplo de arriba se muestra una gráfica de resultados.

Para poder mostrar las visualizaciones en la ventana de Picto es necesario conectar el modelo a visualizar con las transformaciones encargadas de generar la visualización. La Figura 12 muestra un diagrama sencillo de cómo se enlazan las transformaciones con el modelo usando el modelo de ejemplo que hemos usado en las secciones anteriores.

El modelo se conecta con un la transformación mediante un fichero que usa un lenguaje especifico del dominio basado en EMF. Este fichero debe tener el mismo nombre que el modelo más la extensión “.picto”, en el fichero se indica la transformación a ejecutar junto a su formato. También

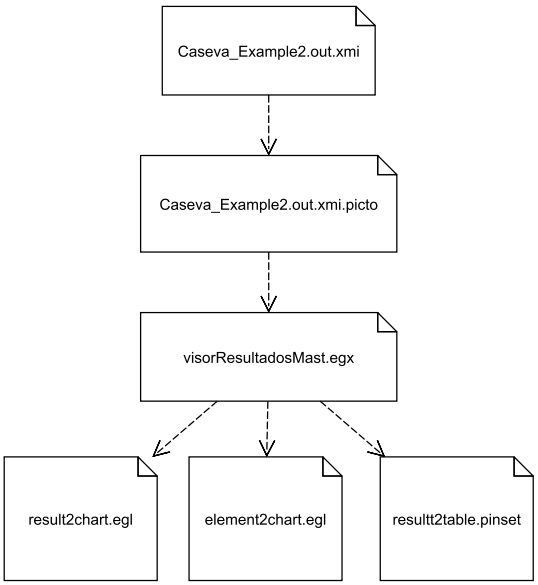

Figura 12: Diagrama de flujo entre archivos en una visualización usando Picto

se pueden pasar parámetros a la transformación pero, como muestra el ejemplo a continuación, decidimos manejar los parámetros dentro de la transformación.

Listing 6: Ejemplo archivo de conexión Picto

```
<?nsuri picto?>
<picto format="egx"
    transformation="platform:/resource/mastresults/picto/visorResultadosMast.egx">
</picto>
```

Para usar más de una transformación hacemos uso de un fichero director EGX (listing 4) que nos permite aplicar varias plantillas que en este caso se corresponden con la gráfica de resultados, las gráficas de flujos y la tabla de resultados.

6. Implementación

En esta sección de profundiza en la implementación de la aplicación explicando las distintas partes que la componen y sus funciones. El código fuente de este proyecto se encuentra publicado en un repositorio público con una licencia de código abierto: https://github.com/Arturo-96/MastVisor

6.1. Transformaciones

En esta sección vamos a explicar las transformaciones usadas en la aplicación, para empezar las transformaciones están gestionadas por el fichero "visorResultadosMast.egx" en el cual cada una de las tres transformaciones se describe en una regla.

Antes de declarar las reglas es necesario cargar los metamodelos, esto lo conseguimos con una sección "pre" la cual ha de ser ejecutada antes de las reglas. Dentro de esta sección usamos una herramienta de EMF para obtener todos los modelos cargados en el proyecto y de estos seleccionamos los metamodelos por su extensión .ecore.

Dentro de las reglas se definen los parámetros de la transformación y el fichero que se usara como plantilla para la transformación. En los parámetros indicamos el camino que tendrá la visualización en el indice de visualizaciones, un icono en caso de que queramos añadir uno y el formato del resultado de la transformación. Una regla de ejemplo:

Listing 7: Ejemplo de regla EGX

```
rule Results2Chart {
    parameters : Map {
        "path" = Sequence{"Charts"},
        "format" = "html"
    }
    template : "results2chart.egl"
}
```

Esta regla genera el gráfico de resultados. Se le pasan como parámetros el nombre que mostrara en el indice de Picto y el formato del archivo generado junto a la plantilla con la cual generar código.

La primera regla se ocupa de generar la tabla de datos, el fichero plantilla es "results2table.pinset" y el formato de salida es csv. Se usa el icono de tabla y la visualización estará en el primer nivel del indice como "Stats".

El resultado de esta regla es un archivo csv que forma una tabla con los datos de los flujos del resultado.

La segunda regla se encarga de generar la gráfica de resultados, el fichero plantilla es "results2chart.egl" y el formato de salida es html. No se especifica icono y la visualización estará en el primer nivel del indice con el nombre "Charts".

El resultado de esta regla es un archivo html que mostrara la gráfica de resultados.

La tercera regla genera las gráficas de flujos, el fichero de plantilla es "element2charts.egl". Esta regla es parecida a la anterior, sin usar icono y devolviendo formato html pero hay que indicar que la transformación tiene que ser ejecutada por cada flujo del modelo de resultados, en el indice las visualizaciones estarán anidadas bajo "Charts" y tendrán el nombre de su respectivo flujo.

El resultado de esta regla es un archivo html por cada flujo de los resultados, cada uno mostrando su propia gráfica.

6.2. Gráfica resultados

La transformación que genera la gráfica de resultados se implementa en fichero "results2chart.egl" el cual contiene código html en el cual se inyecta código generado usando el lenguaje EGL, el código encapsulado usando las combinaciones de caracteres "[%" y "%]" se ejecuta al usar la transformación y en caso de usar "[%=" al principio del encapsulado el resultado de la operación encapsulada sera escrito en el resultado generado.

A continuación un ejemplo sencillo de cómo funciona el lenguaje:

Listing 8: Ejemplo de generación de código con EGL

```
y: [
    [%var i = 0;%]
    [%for( result in results){%]
            [%i++;%]
            '[%=result.Model_Elem.name%]'[%if(i<count)%][%=',%]
    [%}%]
],
```

"y" es el array de html que contiene las etiquetas del eje y de la gráfica, el código EGL inyecta los nombres de los flujos encapsulados en apostrofes y separados por comas.

Al comienzo del fichero se instancian una serie de variables que se usaran en la generación de código, las variables "results" y "timing" almacenaran los flujos del resultado MAST conteniendo results información del modelo de resultados y timing del modelo base, la variable "inf" contiene el valor numérico máximo que se representara en la gráfica, la variable "top" contendrá el máximo valor que se representara en la gráfica.

"C", "B" e "I" son listas para almacenar los tiempos a representar y la lista "arrows" indica que flujos se salen del límite de la gráfica para añadir una flecha que lo indique.

La siguiente sección del fichero contiene los cálculos de los tiempos a representar.

Primero se calcula el límite del gráfico a partir del cual no se representaran los valores, para ello recorremos la lista de flujos "results" de la que extraemos los plazos de cada flujo y guardamos el plazo menor en la variable "top"de forma temporal. En caso de un sistema no planificable, el peor tiempo de respuesta puede ser mayor que el plazo, en caso que el tiempo de peor caso sea mayor que el plazo pero menor que el valor máximo que podemos representar "inf" este sustituirá al plazo para los cálculos.

A continuación se calculas los tiempos de ejecución de los flujos, estos resultados se guardan en la lista "C". Para ello recorremos los flujos y dentro de cada flujo recorremos las operaciones que los componen guardando el sumatorio de los tiempos de ejecución de peor caso de cada operación de un flujo en la lista.

En el siguiente paso se calcula el tiempo de bloqueo que es almacenado en la lista "B", en este caso solo es necesario recorrer "timing" y guardar las sumas de los peores tiempos de bloqueo de las operaciones de cada flujo.

Por último se calcula el tiempo de interferencia en la lista “I”, este se calcula a partir de los otros resultados, primero obtenemos el tiempo de respuesta global de cada flujo y a este se le sustrae los tiempos de ejecución y bloqueo.

Una vez definidas la variables y calculados los datos el resto del fichero se dedica al código html a generar, este consiste unicamente en un script Javascript en el cuerpo del documento conteniendo la declaración de la gráfica en Plotly.

Dentro del script hay que definir una serie de trazas como variables que contendrán los valores a representar de cada sección de una barra, por lo que definimos tres trazas; cada traza necesita un nombre, un tipo que para todas es barra “bar”, una orientación que para todas es horizontal “h”, una lista de valores en el eje Y que para todas serán los nombres de los flujos a representar y una lista de valores del eje X que serán los datos calculados en “C”, “B” e “I” para cada traza respectivamente.

Para inyectar los valores recorremos un bucle dentro de las listas de valores escribiendo los datos necesarios, los nombres para Y y los datos calculados en las listas para X, junto a una coma para todos menos el último ciclo del bucle.

La traza 3 tiene un inyección algo distinta puesto que se encarga de la sección derecha de la gráfica donde necesitamos representar si un valor tiende al infinito, para ello la inyección se divide con un if, si el valor a representar es menor que el máximo representable ”top.$^{\text{en}}$tonces se inyecta el valor normal y se añade un falso a la lista de flechas “arrows”, en caso de que sea mayor se inyecta un valor un 10 % más alto que “top” y se añade un valor verdadero a la lista “arrows”.

Las tres trazas son añadidas a una lista de datos y a continuación se define un layout para la gráfica que contendrá, el titulo con la fuente de texto, un offset y el nombre del modelo MAST inyectado, el modo de representación de las barras que en este caso es apiladas “stack”, las medidas de los margenes de la gráfica y una lista de formas a representar encima de la gráfica. Es dentro de esta lista donde se inyectaran las flechas que indican que los datos tienden la infinito y una barra indicando el plazo de cada flujo.

En esta sección primero se recorre la lista de flechas “arrows” si el valor es verdadero entonces se inyecta el código necesario para dibujar la flecha definiendola como una forma de tipo “path” y describiendo el path como una serie de lineas que forman una flecha, el siguiente paso es rellenar la flecha de color negro e indicar que las lineas que componerla flecha también son de color negro. A continuación se definen las barras de los plazos, para ello se recorren los flujos obteniendo los plazos y por cada una se dibuja una linea en la posición del plazo, las lineas se definen usando dos coordenadas en la gráfica, cada barra es una unidad de ancha por lo que la coordenada y de ambos puntos se define como la posición y de la barra actual +- 0.5 unidades, en ambas coordenadas la posición x es el valor del plazo.

Para terminar el script se crea una configuración para la gráfica donde solo incluimos “responsive=true” para que la gráfica se adapte al tamaño de la ventana de navegador donde se muestre y finalmente se crea la gráfica usando los datos, layout y configuración definidas con anterioridad.

6.3. Gráficas Flows

La transformación que genera la gráfica de flujos esta descrita en el fichero “element2chart.egl” y es muy similar al fichero “results2chart.egl” compartiendo la totalidad del código html no generado.

En la primera sección de código se declaran las mismas variables y se realizan los mismos cálculos que en "results2chart.egl" declarando dos variables adicionales, "element" y "time" que contienen los datos de ambos modelos del elemento a representar que es indicado por el fichero egx en el parámetro "e".

La generación de los datos para el script de la gráfica es idéntica al fichero "results2chart.egl", con las principales diferencias en el layout.

Primero utilizamos el nombre del flujo en vez del nombre del modelo, la generación de las flechas y plazos no cambia respecto a "results2chart.egl". La principal diferencia es la adición del campo "xaxis" que permite acotar la porción del gráfico mostrado en el eje x entre dos valores, en este caso queremos que se ajuste al flujo en el que queremos centrarnos por lo que el primer valor siempre sera cero y solo es necesario calcular el segundo valor.

Vamos a usar el tiempo de interferencia como referencia ya que determina la dimensión de la parte derecha de la barra, para calcular el tiempo de interferencia primero necesitamos calcular los valores de tiempos de ejecución y bloqueo con el mismo método que al principio del fichero. Estos cálculos ya han sido realizados pero, ya que el flujo a mostrar se nos ha sido pasado como parámetro, no conocemos su posición en la lista de datos por lo que no somos capaces de identificarlo. También obtenemos el plazo del flujo.

En caso de que el tiempo de interferencia sea mayor que el valor máximo representable "inf" usaremos el valor máximo a representar de la variable "top" más una holgura del 30 % para dejar espacio para una flecha, en caso contrario si el tiempo de interferencia es mayor al plazo pondremos el valor en el tiempo de interferencia más una holgura del 10 % y si no usaremos el valor del plazo más, de nuevo, una holgura del 10 %

6.4. Tabla resultados

En esta sección se detalla el código usado para la visualización de la tabla de resultados.

La tabla de resultados está definida en el fichero "results2table.pinset", en este fichero se declara un set de datos que sera extraído recorriendo los flujos del modelo. Para ello se definen una serie de columnas donde volcar los datos extraídos así como el método para extraer el dato asociado.

Listing 9: Ejemplo de extracción de tabla con pinset

```
dataset resultsStats over result : End_To_End_Flow_Result {
    column flow : result.Model_Elem.name
    column slack : result.Slack
}
```

Como ejemplo este código generara una tabla con los nombres y holguras de los flujos.

Las columnas representadas son nombre del flujo, holgura, jitter, tiempo de ejecución, tiempo de bloqueo, tiempo de interferencia y plazo. Los tiempos de ejecución, bloqueo e interferencia son calculados del mismo modo que en apartados anteriores, el resto de datos solo es necesario extraerlos del modelo.

6.5. Gestión de configuración

En esta sección se explica cómo está configurado el entorno de trabajo así como la aplicación para su correcto funcionamiento.

El proyecto se ha desarrollado en un entorno Eclipse al cual hemos añadido varios plugins necesarios para el desarrollo y funcionamiento del proyecto.

Para poder usar el paradigma de ingeniería dirigida por modelos ha sido necesario instalar plugins de EMF incluyendo EMF Base Runtime, EMF Core Runtime Developer y EMF SDK así como Flexmi [22] incluyendo Flexmi y Flexmi Development Tools para poder usar modelos y transformaciones.

Los lenguajes Epsilon usados para las transformaciones requieren los plugins de Epsilon relevantes que son Epsilon Core, Epsilon Development Tools, Epsilon Development Tools for EMF y Epsilon EMF Integration.

Para hacer uso de Picto es necesario instalar su plugin en eclipse. Cabe destacar que actualmente para hacer uso de las visualizaciones es necesario tener el proyecto importado en Eclipse y crear un fichero .picto que enlace el modelo a visualizar con la transformación. En un trabajo futuro se podría generar un plugin que implemente la interfaz PictoSource. Con esta interfaz se puede indicar una extensión sobre la que Picto ha de actuar, si todos los modelos de resultados MAST en el entorno de desarrollo terminan en una extensión, como por ejemplo .mast, Picto puede aplicar las transformaciones cada vez que uno de estos modelos se guarde.

Para ejecutar los test se ha usado Junit utilizando Maven [23] para ejecutarlos como un programa Java stadalone en vez de a través de Eclipse. Maven también se encarga de las dependencias necesarias para generar los test.

También se ha hecho uso de un repositorio de GitHub. El proyecto esta ordenado en cinco directorios, como se muestra en la figura 13: metamodelos contiene los metamodelos de MAST2, modelos contiene varios modelos de ejemplo con los que probar las vistas, picto contiene las transformaciones, src contiene el el código de los test y tests contiene las visualizaciones correctas con las que comparar los resultados de los test Junit.

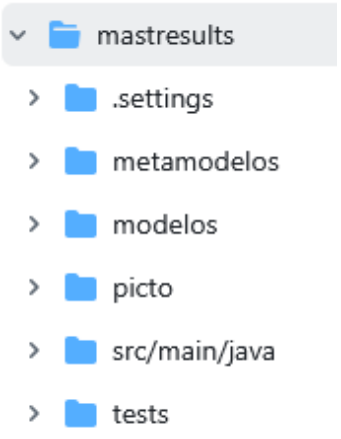

Figura 13: Estructura de los directorios

7. Evaluación

En esta sección se describen los pasos llevados a cabo para evaluar el funcionamiento de la aplicación.

7.1. Pruebas Unitarias

Con el objetivo de validar las gráficas generadas por la aplicación se realizaron unas pruebas unitarias de Junit.

Queremos comprobar que el código html generado por las transformaciones es correcto y se corresponde con una visualización adecuada. Para comprobar esto primero se elaboran una serie de archivos de forma manual a partir de un modelo estático, en este caso "caseva_example2.xmi". Estos archivos se almacenan en la carpeta test para compararlos con los generados por las transformaciones.

El fichero "Tests.java" registra los metamodelos MAST y el modelo a evaluar. A continuación genera las gráficas usando el archivo "visorResultadosMast_test.egx" que funciona igual que "visorResultadosMast.egx" pero genera las gráficas en la carpeta temporal gen.

Para terminar se generan de forma dinámica tantas pruebas Junit como gráficas se hayan generado, estas pruebas comparan el archivo html de una gráfica con el archivo correcto en el directorio tests.

Usando este método no es posible validar las tablas de resultados generadas con pinset ya que los datos con extraídos de forma dinámica por lo que el archivo csv generado por "test.java" contiene las llamadas a métodos para extraer información del modelo en vez de los datos.

7.2. Pruebas de Aceptación

Una vez la aplicación fue capaz de generar gráficas correctas se propuso una reunión con varios expertos del departamento de ingeniería software y tiempo real para evaluar la utilidad de las vistas generadas y recibir sugerencias sobre cómo mejorar las mismas.

Durante la reunión se lanzaron varias sugerencias de las cuales se implementaron dos principales.

Las gráficas originales no representaban los plazos de los flujos por lo que recibimos varias sugerencias para añadirlos a las gráficas.

Varios expertos se quejaron de la dificultas de distinguir los flujos con duración infinita ya que las barras que los representaban se cortan para mejorar la visualización de los otros flujos. Recibimos varias sugerencias sobre como representar los flujos infinitos de las cuales la más fácil de implementar fue mediante una flecha al final de la barra tal y como se describe en la sección 3.1.

8. Conclusiones y trabajo futuro

En esta sección se exponen las conclusiones llegadas durante el trabajo y cambios que se pueden realizar en el futuro para mejorar y expandir la aplicación.

8.1. Conclusiones

En esta sección valoramos como de efectivas son as tecnologías usadas en la aplicación así como la usabilidad de la misma y su efectividad.

Las tecnologías usadas durante la realización del proyecto han resultado ser efectivas a la hora de generar las visualizaciones. Las transformaciones permiten generar el código para las visualizaciones de forma eficaz y ofrecen gran flexibilidad para expandir y crear nuevas visualizaciones en el futuro.

En cuanto a las visualizaciones, cumplen con el objetivo de mostrar los datos de los flujos de forma eficiente y legible. Las gráficas ofrecen una forma rápida de comparar las diferencias de tiempo entre flujos y comprobar si cumplen sus plazos. La tabla ofrece más métricas que las gráficas y es parecida a las que ofrece la herramienta MAST pero centralizado en una sola visualización.

Una vez configurado el entorno de trabajo generar las visualizaciones es tan fácil como abrir el modelo de resultados a visualizar en el IDE. Las visualizaciones aparecerán de forma automática en la ventana de Picto. Inicializar el entorno de trabajo no es trivial, solucionar este problema se deja a trabajo futuro.

8.2. Trabajo futuro

En esta sección exploramos que pasos pueden tomarse para expandir y mejorar la aplicación, incluyendo vistas adicionales y otras características.

Para empezar se podría implementar la interfaz PictoSource como describimos en la sección 6.5, de esta manera la aplicación se instalaría como un plugin de Eclipse y las visualizaciones se generarían automáticamente para los modelos de resultados en el entorno.

Durante la realización de este proyecto solo se consideraron flujos lineales. En un futuro se podría modificar la aplicación para poder generar visualizaciones de sistemas con flujos no lineales. La lógica para lograr esto sería mucho más compleja pero sería posible.

A medida que el plugin de Picto sea utilizado, se estudiara realizar cambios adicionales o añadir visualizaciones atendiendo a las posibles sugerencias de sus usuarios.

Referencias

[1] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and H. P. Tijero, "Curso sistemas de tiempo real," 2012. [Online]. Available: https://ocw.unican.es/course/view.php?id=255

[2] S. engineering and real-time group Universidad de Cantabria, "Mast: Modeling and analysis suite for real-time applications." [Online]. Available: https://mast.unican.es/

[3] "Gtk." [Online]. Available: https://www.gtk.org/

[4] C. C. Cuesta, J. M. Drake, M. G. Harbour, J. J. Gutiérrez, P. L. Martínez, J. L. Medina, and J. C. Palencia, "Mast metamodel." [Online]. Available: https://mast.unican.es/simmast/MAST_2_0_Metamodel.pdf

[5] "Uml profile for marte," 8 2024. [Online]. Available: https://www.omg.org/spec/MARTE/

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: clipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2009.

[7] "Eclipse epsilon." [Online]. Available: https://eclipse.dev/epsilon/

[8] "Epsilon playground ejemplo eol." [Online]. Available: https://eclipse.dev/epsilon/playground/?eol

[9] "Epsilon playground ejemplo egl." [Online]. Available: https://eclipse.dev/epsilon/playground/?egl

[10] "Epsilon playground ejemplo egx." [Online]. Available: https://eclipse.dev/epsilon/playground/?egx

[11] "Dataset extraction (pinset)." [Online]. Available: https://eclipse.dev/epsilon/doc/pinset/

[12] "Epsilon playground ejemplo pinset." [Online]. Available: https://eclipse.dev/epsilon/playground/?psl2csv

[13] D. Kolovos, A. de la Vega, and J. Cooper, "Efficient generation of graphical model views via lazy model-to-text transformation," 2020. [Online]. Available: https://eprints.whiterose.ac.uk/id/eprint/164209/1/models2020_picto.pdf

[14] "Plotly." [Online]. Available: https://plotly.com/

[15] "Plotly: Data visualization & dashboards." [Online]. Available: https://plotly.com/examples/dashboards/

[16] "Vega projects visualization grammars." [Online]. Available: https://vega.github.io/

[17] "Chart.js." [Online]. Available: https://www.chartjs.org/

[18] "Json (javascript object notation)." [Online]. Available: https://www.json.org/

[19] "Javascript." [Online]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript

[20] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack, "The epsilon generation language," in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, ser. Lecture Notes in Computer Science, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 1–16. [Online]. Available: https://doi.org/10.1007/978-3-540-69100-6_1

[21] A. de la Vega, P. Sánchez, and D. S. Kolovos, "Pinset: A DSL for extracting datasets from models for data mining-based quality analysis," in *11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018, Coimbra, Portugal, September 4-7, 2018*, A. Bertolino, V. Amaral, P. Rupino, and M. Vieira, Eds. IEEE Computer Society, 2018, pp. 83–91. [Online]. Available: https://doi.org/10.1109/QUATIC.2018.00021

[22] D. S. Kolovos and A. de la Vega, "Flexmi: a generic and modular textual syntax for domain-specific modelling," *Softw. Syst. Model.*, vol. 22, no. 4, pp. 1197–1215, 2023. [Online]. Available: https://doi.org/10.1007/s10270-022-01064-3

[23] "Maven." [Online]. Available: https://maven.apache.org/