PROYECTO DE SISTEMAS INFORMÁTICOS
2005/2006

Facultad de Informática
Universidad Complutense de Madrid

EDITOR GRÁFICO PARA MODELADO Y SIMULACIÓN CON EL FORMALISMO *DEVS*

Autores
Carlos García Arano
Samer Hassan Collado
María Teresa Murguialday Barrio

Profesores Directores
Jesús Manuel de la Cruz
José Luís Risco Martín

Resumen:

DEVS[1] es un formalismo matemático de modelado y simulación basado en conceptos de sistemas dinámicos genéricos con conceptos bien definidos de acoplamiento de componentes, jerarquía, construcción modular de modelos. Nuestro proyecto se basa en la construcción de un editor gráfico de modelos que siguen el formalismo DEVS, y en su integración con una herramienta externa: el simulador de esos mismos modelos.

Además el editor es un generador de código que trabaja con modelos almacenados en archivos XML y crea archivos de este tipo una vez se ha generado un modelo gráfico, sin tener que escribir código para generar un modelo.

Abstract:

DEVS is a mathematical formalism of modelling and simulation based on concepts of generic dynamic systems with well defined concepts of components coupling, hierarchy and modular models building. The aim of project is developing a graphical models editor, using DEVS, integrated with models simulator (an external tool).

Besides, the editor is a code generator which works with XML models and makes files in this format from the graphical model created, without need to write code.

Palabras clave (Keywords):

DEVS, JGraph, XML, XML-Schema, Editor gráfico, simulador.

[1] Discrete Event System Specification

1. INTRODUCCIÓN

1.1 Objetivos del proyecto

Este proyecto se centra en la construcción de un editor gráfico de modelos que siguen la formalismo DEVS (***D****iscrete* ***EV****ent* ***S****ystem Specification)*. Mediante un entorno gráfico el usuario podrá generar su modelo jerárquico a partir de modelos DEVS más sencillos, ya construidos o incluso creados por el propio usuario. La gestión (creación, modificación y almacenamiento) de dichos modelos se llevará a cabo mediante archivos *XML*, respetando una estructura definida en *XM- Schema*.

Uno de nuestros objetivos principales era desarrollar un proyecto en código abierto, y esto se ha respetado tanto en el código generado como en las librerías utilizadas.

Otra de las características prioritarias ha sido la creación de un diseño modular, claro, y fácilmente ampliable, que a su vez brindara una utilización sencilla y una interfaz amigable de cara al usuario, sin renunciar a toda la potencia que una herramienta de estas características puede proporcionar. Con este propósito se han utilizado distintos patrones de diseño a lo largo del desarrollo, y el código se ha realizado en Java, con el fin de obtener una aplicación multiplataforma. Se han utilizado también las estructuras de datos que proporciona la API de Java, para obtener un diseño más robusto en cuanto al soporte de errores y un código más eficiente, ya que dichas estructuras han sido sometidas a numerosas pruebas.

Para realizar todos estos objetivos, previo al comienzo del desarrollo de la aplicación, se ha realizado una labor de investigación de distintos entornos similares al que se pretendía generar, y se ha invertido mucho tiempo en la creación de un diseño robusto que respetara todas las propiedades ya comentadas.

Finalmente cabe destacar que, debido a las características de este proyecto, va a haber un componente muy fuerte de investigación y auto-formación, ya que a lo largo de la carrera no se realiza nada parecido a lo que presentaremos a continuación.

1.2 Resumen global[2]

Como ya hemos comentado, el proyecto se centra en la construcción de un editor gráfico de modelos que siguen la formalismo DEVS [3](***D**iscrete **EV**ent **S**ystem Specification)*. DEVS es un estándar que proporciona una manera de definir sistemas. Podríamos separar el proceso de definición/simulación de un modelo DEVS en dos módulos bien definidos: el entorno de edición de modelos (Etapa 1) y el simulador (Etapa 2). La aplicación que presentamos se corresponde con la Etapa 1. Pero proporcionaremos ejemplos de creación y posterior simulación de modelos, para lo que se ha utilizado como simulador el que nos ha proporcionado el grupo de investigación ISCAR[4]. El entorno es un conjunto de herramientas software que corren en cualquier sistema operativo que disponga de máquina virtual Java. A continuación mostramos un diagrama de bloques para una mayor comprensión de la diferenciación de etapas:

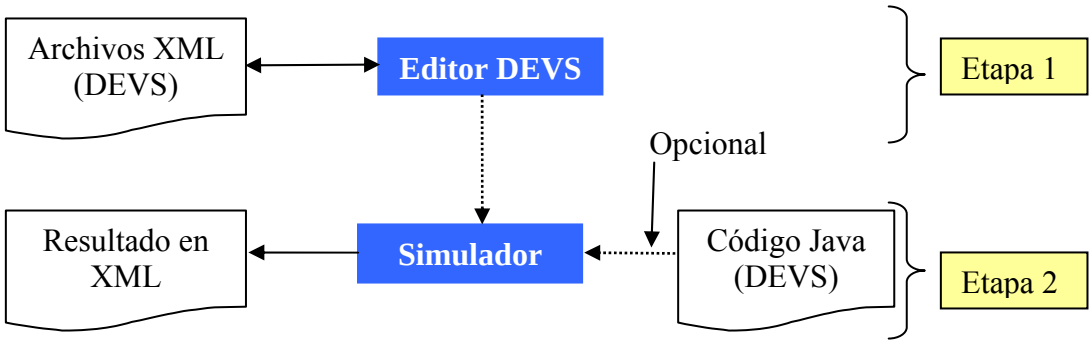

Figura 1.2.1: Cómo definir y simular un modelo que cumple el formalismo DEVS:

Desde el editor se pueden construir archivos XML que contienen modelos que cumplen el formalismo DEVS. Además en el editor se pueden cargar y modificar archivos XML con estos modelos. Trabajamos con archivos XML que cumplen un XML-Schema del que se hablará en puntos posteriores.

En el simulador se pueden cargar archivos XML externos, siempre y cuando cumplan dicho XML-Schema, o los que se han creado a partir del editor. También

[2] Si desea conocer con mayor detalle las características que ofrece nuestro entorno, acuda al apartado de funcionalidades.
[3] El formalismo DEVS se describe con detalle en el apartado 2 de este documento.
[4] Ingeniería de sistemas, control, automatización y robótica.

puede recibir la entrada en un archivo Java, pero los archivos XML proporcionan una gran potencia, al ser este el estándar para el intercambio de información estructurada entre diferentes plataformas.

Los componentes del editor son:

- *Lienzo*: donde el usuario puede editar modelos DEVS especificando el acoplamiento de manera gráfica. Pueden abrirse simultáneamente varias pestañas para trabajar en paralelo sobre varios modelos.
- *Inspector de objetos*: muestra propiedades gráficas y atributos de un submodelo seleccionado en el lienzo.
- *Barra de herramientas, menú superior y menús emergentes con botón derecho del ratón*: con las acciones que se permiten realizar sobre los componentes del modelo o sobre el propio modelo (añadir puertos a un submodelo, salvar la estructura del modelo en un archivo XML, abrir una nueva pestaña…).
- *librería (biblioteca)*: con los submodelos de los que dispone el usuario para generar su modelo actual.

En el lienzo del editor cada componente (submodelo) del modelo que se está creando se representa como una caja negra, con una etiqueta que indica su nombre, un icono gráfico y un conjunto de puertos de entrada/salida. Si se selecciona en el lienzo un submodelo se presentan, en el inspector de objetos del editor, tanto las propiedades gráficas de la caja como los atributos del modelo, sea este atómico o complejo. Se pueden conectar puertos de salida de los submodelos con puertos de entrada de otros, representándose en el lienzo gráficamente dicha unión por medio de una flecha etiquetable. También se pueden camuflar en el lienzo bajo una caja negra, un conjunto de los submodelos, pudiéndose desacoplar si fuera necesario posteriormente.

Para diseñar toda esta representación, la herramienta de Matlab Simulink resultó ser una fuente de inspiración esencial: para la visualización de los grafos, su gestión y hasta para las opciones disponibles en la interfaz.

Finalmente podemos destacar una de las características más potentes de nuestro proyecto: el usuario podrá almacenar el modelo que ha creado gráficamente en un

archivo XML. Además se permite cargar en el editor estos mismos documentos XML u otros que hayan sido generados en formato XML mediante otras herramientas o a mano. Este formato ha sido elegido por tratarse del estándar para el intercambio de información estructurada entre diferentes plataformas, aumentando el potencial del editor extraordinariamente. Con esto se permite gestionar modelos que cumplen el formalismo DEVS sin necesidad de que el usuario genere una sola línea de código. Es más, esto permitiría realizar un traductor que desde cualquier herramienta que siga el formalismo DEVS permitiera exportar modelos que fueran utilizados en nuestro editor.

1.3 Estructura de la memoria

En esta memoria se explican en profundidad, principalmente, los conceptos DEVS, las características de nuestro editor, las distintas etapas de su desarrollo, y su uso con ejemplos. A continuación se procede a detallar concretamente la estructura de los diversos puntos de que consta.

En el punto 1 se pretende dar una visión global del proyecto, introduciendo al lector en los fines que se persiguen con el desarrollo. En particular, en el subpunto 1.1 se detallan los objetivos generales y prioridades que se han tenido en cuenta. En el apartado 1.2 se introducen los principales conceptos con una visión integral del sistema y un paso por sus características más sobresalientes. El punto 1.3, este mismo, detalla la estructura de este informe, mientras que el 1.4 compara nuestro proyecto con las aplicaciones existentes previamente, analizando las razones de su originalidad.

El punto 2 persigue aclarar los conceptos del formalismo DEVS a quien no esté familiarizado con ellos. El 2.1 realiza un recorrido por los pasos incrementales del diseño DEVS: modelos atómicos, con la explicación de su estructura básica (principal sección de este punto); modelos acoplados, con sus características propias; y la construcción jerárquica de ellos. Además, el subpunto 2.2 detalla algunos ejemplos para su mejor comprensión.

El punto 3 especifica el alcance de la funcionalidad de nuestro editor, definiendo la lista de características más en profundidad que en el punto 1. Este punto, junto con el primero, proporciona la información básica para los lectores que no deseen entrar en los pormenores de la implementación. Así, pueden observarse de un vistazo las principales funcionalidades que ofrece nuestro editor en el subpunto 3.1. Y esas mismas características en detalle pueden leerse en el 3.2.

En el punto 4 se exponen las distintas etapas del proceso de desarrollo, dividiéndose en subpuntos con sus distintas fases. El 4.1 otorga una visión amplia de lo que significó el desarrollo, describiendo además las herramientas utilizadas. Le suceden

tres subpuntos de investigación. El 4.2 detalla cuáles cuestiones concretas tuvieron que ser más investigadas del formalismo DEVS como paso previo a cualquier otra fase. El 4.3 explica las ventajas que nos llevaron a utilizar XML y XML-Schema, una explicación de su uso y el XML-Schema que diseñamos para nuestros modelos DEVS. El 4.4 desarrolla el debate que nos supuso la elección de una librería gráfica, y las razones de decidir Jgraph.

El 4.5 es el principal subpunto para comprender el diseño implementado y sus pormenores. Se divide en amplios apartados que detallan: la evolución del diseño UML; la estructura de la librería Jgraph; la estructura del modelo y la vista de nuestro proyecto. El 4.6 otorga nuevos detalles en torno a la implementación, y en particular centrándose en la aplicación de dos patrones de diseño en dos capas distintas: la correspondiente a los grafos de Jgraph y la de la interfaz gráfica de usuario.

El punto 5 cumple la función de cierre en lo que al desarrollo respecta. En el subpunto 5.1 se detalla la metodología formal que se ha cumplido estrictamente a lo largo del proyecto. El 5.2 proporciona un ejemplo de uso detallado, mientras que el 5.3 expone las posibles vías de evolución futuras que este editor podría tener.

Como últimos puntos de esta memoria se ha incluido: un práctico manual de usuario en el punto 6, a modo de las aplicaciones comerciales; unas conclusiones generales desde la óptica de sus desarrolladores en el punto 7; una bibliografía (punto 8), que recopila las múltiples referencias encontradas a lo largo del texto; y un glosario que define las principales abreviaturas del texto. También se ha incluido una autorización para la Universidad Complutense de Madrid en la que se le permite su uso para fines académicos. Al ser un proyecto de código abierto, se adjunta su código en la versión digital.

1.4 Antecedentes

Existen ya entornos para generar modelos DEVS, como por ejemplo DEVSJAVA o JDEVS. Se ha realizado un trabajo de investigación sobre estos entornos, para descubrir que aspectos innovadores podíamos aportar, llegando a la conclusión de que era mejorable tanto la potencia que se puede ofrecer como la claridad y facilidad de uso.

La principal ventaja que observamos en JDEVS es la facilidad de su uso, pero en contraposición nos resultó pobre la potencia que ofrece, y no es excesivamente riguroso en cuanto a respetar el formalismo DEVS, aspectos que hemos mejorado notablemente con nuestra aplicación.

Comparando nuestro trabajo con DEVSJAVA, hemos intentado simplificar el diseño, ya que desde nuestro punto de vista, su uso y estructura eran algo complicados. En su favor podemos destacar por un lado la potencia que proporciona y por otro el rigor con el que sigue el formalismo DEVS. Otra innovación de nuestro proyecto con respecto a DEVSJAVA es la gestión de los modelos DEVS mediante archivos XML.

2. DEVS

En esta sección vamos a realizar una introducción sucinta del formalismo DEVS, centrándonos en la estructura de los modelos más que en su comportamiento. Se explicará de manera informal la construcción jerárquica y modular de los sistemas de eventos discretos. [5]

2.1. Introducción básica al Formalismo DEVS

El formalismo DEVS (***D**iscrete **EV**ent **S**ystem Specification*) fue formulado por Bernard P. Ziegler, y proporciona una manera de especificar sistemas. Básicamente, un sistema está compuesto por una base temporal, entradas, salidas, estados y funciones para determinar los siguientes estados y salidas según las entradas y los estados actuales. En los sistemas de eventos discretos, las entradas llegan en momentos arbitrarios de tiempo, a diferencia de los sistemas continuos.

En la tarea de modelado y simulación intervienen tres objetos básicos, como se aprecia en la ilustración 2.1.1.

- **Modelo**: Se define como un conjunto de instrucciones que generan datos comparables a los datos observados en el sistema real. La estructura del modelo es su conjunto de instrucciones. El comportamiento del modelo queda definido como todos los posibles datos que pueden ser generados con éxito ejecutando las instrucciones del modelo.
- **Simulador**: Utiliza las instrucciones del modelo para generar su comportamiento, que es el conjunto de todos los posibles datos.
- **Marco Experimental**: Especifica las condiciones bajo las cuales el modelo será observado.

[5] Para obtener información más formal del concepto teórico, consultar la obra de Bernard P. Zeigler y Hessam S. Sarjoughian “Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models”.

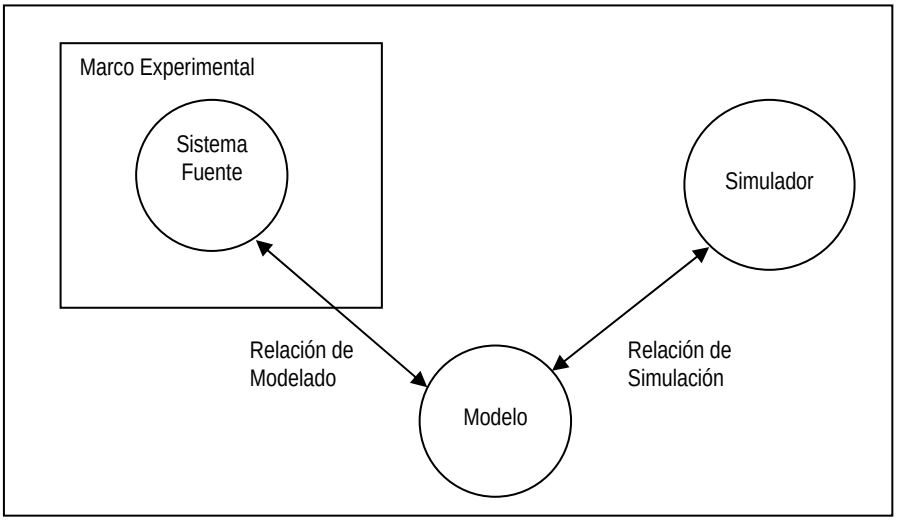

Figura 2.1.1 – Entidades básicas y relaciones

La *relación de modelado* define como el modelo representa el sistema o la entidad a modelar, mientras que la *relación de simulación* muestra como el simulador lleva a cabo las instrucciones del modelo.

En este caso se va a tratar solamente de la representación de los modelos, ya que el objetivo del proyecto es sintetizar código XML que los represente mediante una herramienta gráfica. El simulador proporcionado por el grupo de investigación ISCAR[6] se trata como una herramienta externa, aunque esté integrada en la aplicación.

2.1.1. Modelos Básicos

En el formalismo DEVS se deben especificar modelos básicos a partir de los cuales se construirán otros más complejos, y las conexiones entre ellos de manera jerárquica.

Un modelo básico contiene la siguiente información:

- Un conjunto de **puertos de entrada,** por los que se reciben los eventos externos.

[6] Ingeniería de sistemas, control, automatización y robótica.

- Un conjunto de **puertos de salida**, desde los que se envían los eventos externos
- Un conjunto de **variables de estado** y **parámetros**: dos de las variables de estado que aparecen normalmente son "phase" y "sigma". En ausencia de eventos externos el sistema permanece en la "phase" (estado) actual durante el tiempo indicado por el valor de "sigma".
- Una **función de avance del tiempo**, que controla las transiciones internas. Cuando la variable de estado "sigma" está presente, esta función simplemente devuelve el valor de "sigma".
- Una **función de transición interna**, que especifica el siguiente estado al que el sistema debe pasar al expirar el tiempo dado por la función de avance del tiempo
- Una **función de transición externa**, que especifica el cambio a otro estado cuando se recibe una entrada. Depende del estado actual, el puerto de entrada y el valor del evento externo, y el tiempo transcurrido en el estado actual
- Una **función de confluencia**, que se aplica cuando sucede un evento externo y al mismo tiempo se ha de producir una transición interna. Por defecto se realiza primero la transición interna.
- Una **función de salida,** que genera una salida justo antes de que tenga lugar una transición interna.

Formalmente un modelo DEVS se define con la siguiente estructura:

$$M=<X,S,Y, \delta_{int}, \delta_{ext}, \lambda, ta>$$

donde,

X es el conjunto de **entradas**

S es el conjunto de **estados**

Y es el conjunto de **salidas**

δ_{int} :S→S es la **función de transición interna**

δ_{ext} :Q × X→S es la **función de transición externa**, donde

$Q=\{(s,e)|s \in S, 0 \leq e \leq ta(s)\}$ es el **conjunto total de estados**

e es el **tiempo transcurrido** desde la última transición

λ :S→Y es la **función de salida**

ta :S→$R^{+}_{0,\infty}$ es la **función de avance de tiempo**

A continuación mostramos un diagrama con el que se pretende aclarar el funcionamiento de los modelos básicos de DEVS, y que explicamos a continuación.

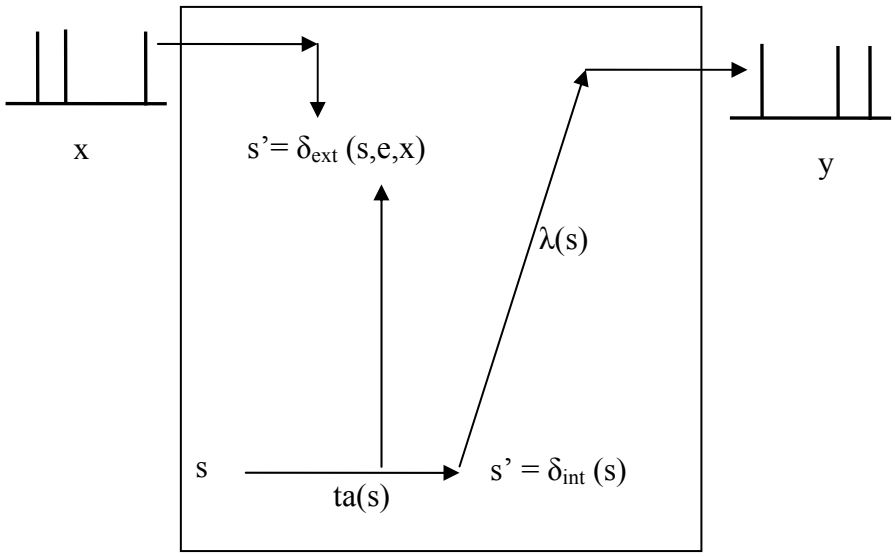

Figura 2.1.1.1: Estructura de un átomo de DEVS

En cualquier momento, el sistema está en cualquier estado S. Si no sucede ningún evento externo, el sistema permanecerá en el estado S durante un tiempo ta(s). Hay que hacer notar que ta(s) puede ser tanto un número real positivo como 0 o infinito. En el caso de que sea cero, la estancia en el estado S es tan corta que no puede suceder evento externo alguno: S es un estado de transición. En el caso de que sea infinito, el sistema permanecerá en S hasta que algún evento externo interrumpa: S es un estado pasivo.

Cuando el tiempo de espera termina, es decir, cuando el tiempo transcurrido e = ta(s), el sistema devuelve un valor, λ(s), y cambia al estado δ_{int}(s). Es importante el hecho de que la salida sólo es posible justo antes de las transiciones internas.

2.1.2. Modelos Acoplados

Los modelos complejos o acoplados especifican la manera en que deben ser conectados los modelos que lo componen. Se construyen a partir de modelos DEVS, que pueden ser, a su vez simples o acoplados. Con todo ello se consigue ir construyendo, de manera modular y jerárquica, modelos más complejos a partir de componentes más sencillos, como explicaremos en el siguiente apartado.

Un modelo acoplado contiene la siguiente información:

- Un conjunto de componentes.
- Un conjunto de puertos de entrada desde donde recibe los eventos externos.
- Un conjunto de puertos de salida por los que envía los eventos externos.

La manera de realizar las conexiones es bastante intuitiva y consiste en:

- Los puertos de entrada del modelo acoplado se conectan a los uno o más puertos de entrada de los submodelos.
- Los puertos de salida de los submodelos se conectan a los puertos de salida del modelo acoplado.
- Las conexiones internas se realizan entre submodelos. Así, un submodelo puede generar una entrada en un submodelo, además de propagarlo a un puerto de salida del modelo acoplado.

2.1.3. Construcción Jerárquica de Modelos

Un modelo acoplado puede ser expresado como un modelo básico. De esta manera se permite emplear modelos acoplados como modelos básicos y construir cada vez modelos más complejos y sofisticados.

2.2. Usos de DEVS

El poder expresivo de DEVS es asombroso. En la obra de Zeigler[7] se llega a simular el funcionamiento de un computador. Es mas, al poder representar el comportamiento completo de una máquina de Turing, su capacidad se amplía a todo el rango de algoritmos conocidos posible. Así, es capaz de expresar todo el rango de modelos de eventos discretos, y no se limita a un dominio específico en su aplicación. Presentamos a continuación una serie de ejemplos que demuestran esta potencia expresiva y los distintos campos de aplicación.

Router Simula el comportamiento de un dispositivo de encaminamiento en redes de computadores. Implementa varios algoritmos de encaminamiento (routing) en una topología fija.

http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain2.htm#routing

Incendios Simula la propagación e intensidad de un incendio. Existen otras simulaciones que incluyen la presencia de lluvia o bomberos para determinar su influencia.

http://www.sce.carleton.ca/faculty/wainer/research/fire.htm

Ciudad Simula el tráfico de una sección de Buenos Aires. Controla las posibles colisiones entre vehículos.

http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain1.htm#CarsOD

Procesador Simula el comportamiento de un procesador con fines educacionales.

http://www.sce.carleton.ca/faculty/wainer/usenix/

Colisión de Partículas Simula el comportamiento de diferentes gases en un entorno cerrado.

http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain2.htm#particle

Arroz Simula la polución provocada por una planta química en un campo de arroz por medio del agua.

http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain2.htm#rice

[7] Bernard P. Zeigler y Hessam S. Sarjoughian “Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models”.

3. Funcionalidad[8]

Lista de funcionalidades

- Interfaz amigable con acciones replicadas e imágenes identificativas
- Textos informativos dinámicos
- Opciones habituales de selección
- Opciones habituales de edición
- Menús emergentes múltiples
- Librería de modelos predefinidos
- Personalización de la librería
- Trabajo simultáneo sobre distintos modelos.
- Iconos y etiquetas identificativos para los modelos
- Potencia en la gestión de modelos acoplados
- Definición de nuevas conexiones entre modelos
- Visualización inmediata de atributos y propiedades gráficas
- Edición gráfica de atributos
- Atributos internos de los modelos atómicos editables
- Ocultación y visualización de puertos
- Diferenciación entre puertos de entrada y de salida
- Cargar un modelo XML
- Salvar un modelo XML
- Simulación integrada

[8] En la descripción global del proyecto ya se comentaron algunas de ellas. Aquí mostramos una enumeración y una breve explicación de todas las funcionalidades que ofrece el editor.

Características en detalle.

En este apartado enumeraremos y explicaremos las características que ofrece el editor de nuestro proyecto. Para obtener una interfaz más amigable de cara al usuario, algunas funciones pueden realizarse de varios modos distintos (por ejemplo con una opción de menú y con un botón, ambos con el mismo icono…).

La siguiente figura muestra un ejemplo del funcionamiento de nuestro editor. La utilizaremos para definir cada una de las partes del mismo, con el fin de proporcionar una mejor comprensión de las funcionalidades que presenta.

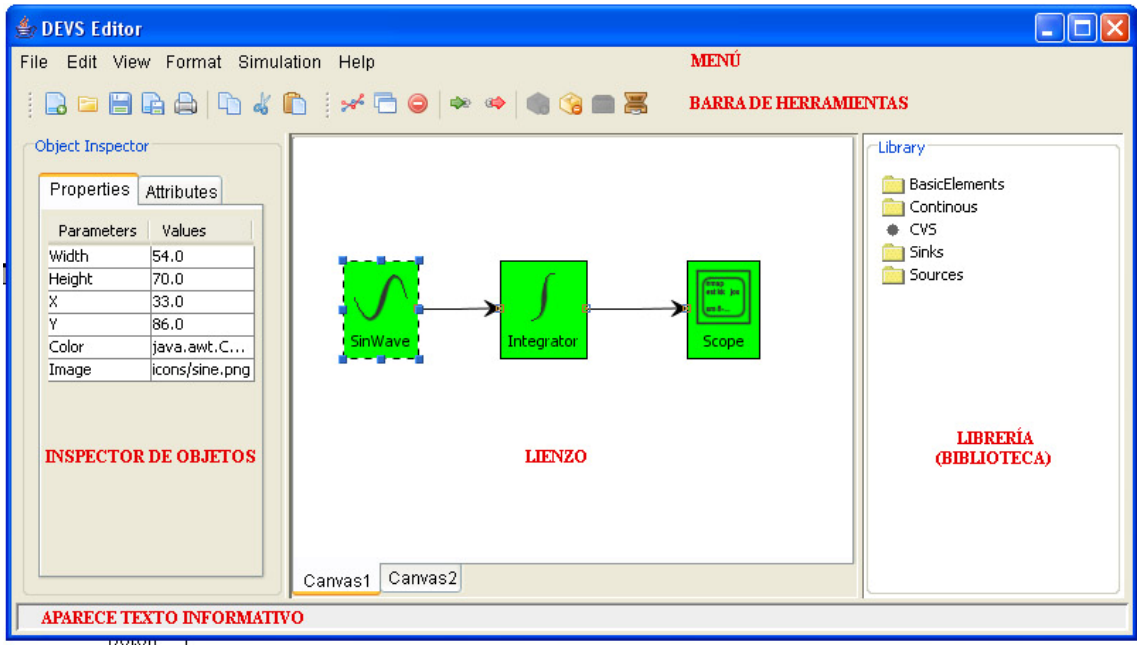

Figura 3.2.1: Ejemplo de funcionamiento del editor de modelos.

- La parte central del editor es lo que denominamos *lienzo*. El usuario puede crear sobre él gráficamente su modelo, que podrá almacenar posteriormente como un archivo XML. En el lienzo los submodelos que componen el modelo acoplado del usuario se representan como cajas negras con un icono gráfico y un nombre. Estas cajas pueden conectarse unas con otras ya que disponen de puertos visibles de entrada y de salida. Además se pueden añadir dinámicamente puertos sobre cada uno de los submodelos con las opciones del menú o de la barra de herramientas que se detallarán posteriormente. También se pueden acoplar bajo

una caja negra un conjunto de submodelos del lienzo, y desacoplarlos posteriormente. En la Figura 3.2.1 aparecen dos pestañas con distintos lienzos. Se pueden abrir simultáneamente distintos lienzos para trabajar en paralelo con varios modelos.

- En el *inspector de objetos* se muestran las propiedades gráficas (medidas, posición, archivo del icono…) de la caja seleccionada (en caso de haber alguna seleccionada) y los atributos del modelo que dicha caja representa. Si se trata de un modelo atómico cada uno de sus atributos cuenta con un nombre, tipo, documentación explicativa y valor por defecto (excepto en el caso del tipo *List* que no tiene valor por defecto). Tanto en los modelos atómicos como en los acoplados aparecerán también como atributos los puertos de entrada y los de salida. Las propiedades y atributos que se muestran se actualizan dinámicamente según se modifique la celda seleccionada.
- En la parte superior se encuentra el *menú* con 5 opciones principales:
  - *File*: Submenú con todas las opciones permitidas para la gestión de los archivos (Abrir un nuevo lienzo, abrir un archivo XML que contenga un modelo (definido con cualquier herramienta) y cargarlo en el editor, cerrar el lienzo actualmente seleccionado, salvar, salvar como, exportar como librería, imprimir y salir de la aplicación).

  - *Edit*: Con las principales opciones de edición como son seleccionar todo, cortar, copiar, pegar, abrir la celda seleccionada en un nuevo lienzo y borrar los elementos seleccionados. Además desde este submenú se pueden asociar nuevos atributos a un elemento atómico: si seleccionamos "Edit → Edit Atomic" aparece un formulario como el que se muestra en la figura 3.2.2:

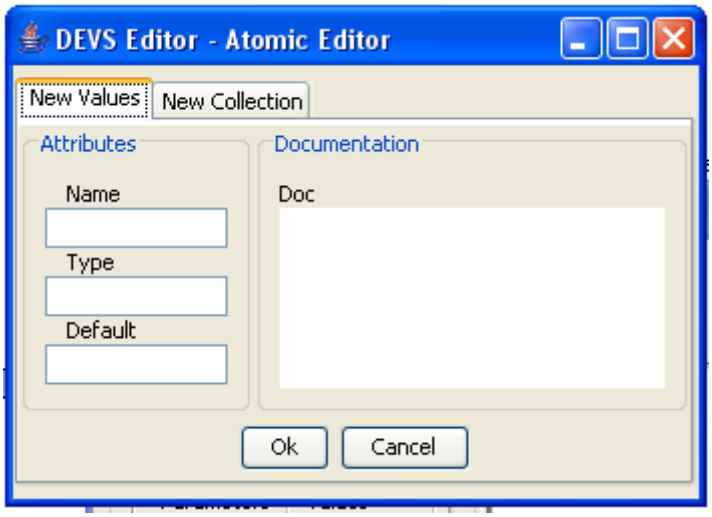

Figura 3.2.2: formulario para asignar propiedades a un modelo atómico

Como podemos ver en la figura, el nuevo atributo tendrá un nombre, un tipo, un valor por defecto y una documentación explicativa. Dicho atributo también podrá ser una lista (en la pestaña no seleccionada), en cuyo caso tendrá nombre, tipo, y documentación explicativa.

- *Simulation*: Con la opción de lanzar el simulador, que ejecutará el XML correspondiente.

- Justo bajo el menú encontramos la barra de herramientas. En el primer y segundo segmentos del lado izquierdo de la misma se duplican ciertas funciones de los menús *File* y *Edit* respectivamente.

  En el tercer segmento aparece un botón para borrar el elemento seleccionado (función que ya se ofrecía en el submenú *Edit*), otro botón para abrir la celda seleccionada en un nuevo lienzo Y un botón *connect*. Este último sirve para mostrar los puertos de los submodelos, habilitando la opción para que se pueda conectar un puerto de salida de un submodelo del lienzo actual de trabajo, con un puerto de entrada de otro submodelo del lienzo. Si se vuelve a pulsar el botón se deshabilita dicha posibilidad.

  En el cuarto segmento de la barra encontramos dos botones para añadir puertos de entrada o de salida al submodelo actual seleccionado. Cuando se añade un puerto, si se selecciona el modelo, se puede ver en el inspector de objetos cómo su número de puertos ha aumentado.

En el quinto y último segmento se muestran las herramientas para agrupar/desagrupar un conjunto de submodelos. También aparecen en él herramientas para acoplar (collapse) un conjunto de submodelos bajo una caja negra y poder desacoplarlos (expand) luego si fuera necesario.

- *Librería (Biblioteca)*: Muestra el contenido del subdirectorio *library*, dividido en los cuatro tipos principales de modelos que existen, guardados en formato XML en los distintos directorios. El usuario podrá personalizar esta librería añadiendo nuevos directorios con los modelos atómicos o acoplados que defina.

- *Menú emergente con el botón derecho del ratón*: Para mayor comodidad del usuario, según se seleccione un modelo u otro, con el botón derecho del ratón aparece un menú emergente con distintas opciones, todas las cuales están replicadas bien en el menú superior, o bien en la barra de herramientas (y por tanto ya han sido comentadas en este apartado). Un ejemplo del menú emergente puede verse en la figura 3.2.3:

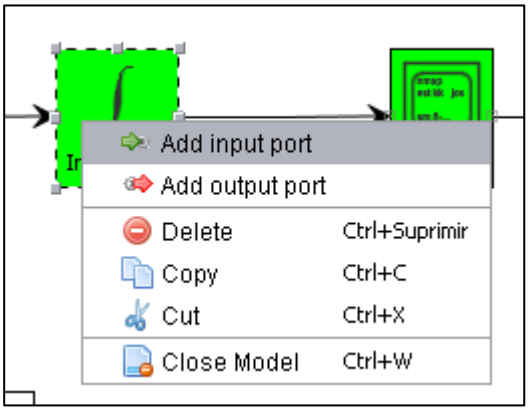

Figura 3.2.3 – Detalle de menú emergente contextual

4. ETAPAS DEL PROCESO DE DESARROLLO

Introducción y herramientas utilizadas

Como ya hemos comentado en otros apartados, este proyecto está muy centrado en la investigación de nuevas tecnologías, lo que ha conllevado una profunda labor de investigación y formación para nosotros. Entre otros campos, podemos destacar los patrones de diseño, el formalismo DEVS, archivos XML y las librerías gráficas. Los patrones de diseño no han sido estudiados en profundidad en las asignaturas de la titulación, y con respecto a las librerías gráficas, los archivos XML y el formalismo DEVS eran temas completamente nuevos para nosotros. Se ha seguido un desarrollo incremental con etapas bien definidas, y a lo largo del apartado 4 expondremos en orden cronológico estas etapas. Podríamos clasificarlas en tres grandes grupos: investigación y formación, diseño e implementación. A continuación adjuntamos un mapa cronológico con todas las tareas que se han realizado y que serán explicadas en detalle en los subapartados posteriores:

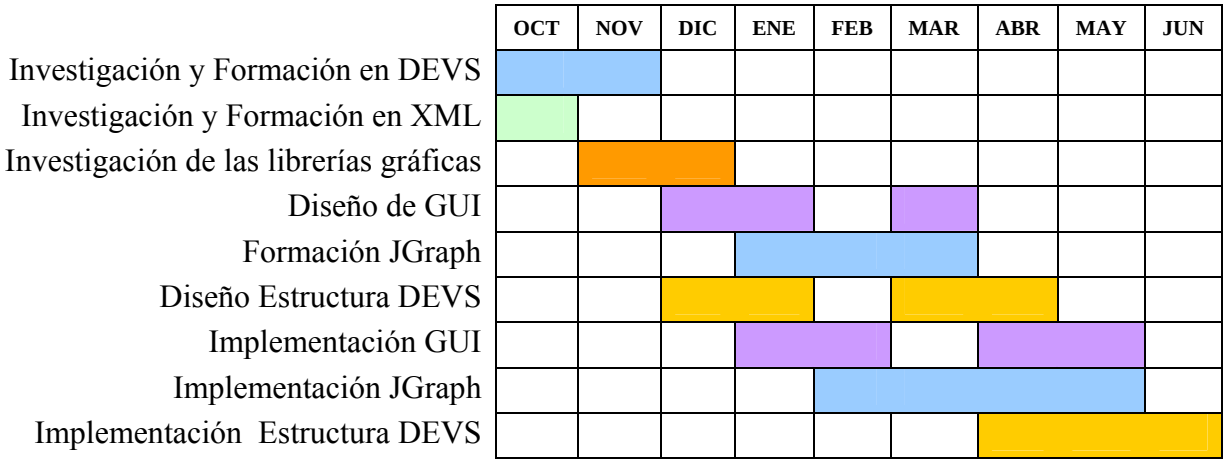

Tabla 4.1.1 - Diagrama de Gantt con las etapas de desarrollo

En la Tabla se resumen las etapas del desarrollo, especificando la tarea realizada, el tiempo consumido y el solapamiento entre las tareas. Los huecos existentes en las diferentes etapas vienen dados por la necesidad de replantear ciertas cuestiones de diseño, como se explica en las secciones “Evolución del diseño” y “Interfaz gráfica de usuario”. Las etapas que están íntimamente relacionadas vienen representadas con el mismo color.

Como decisiones de implementación señalaremos aquí las más destacadas:

En primer lugar el proyecto se ha desarrollado en JAVA, consiguiendo de este modo una aplicación multiplataforma. Además se ha utilizado la última versión disponible del SDK (5.0), aprovechando sus últimas novedades. También se han utilizado las estructuras de datos que proporciona la API de JAVA, para obtener un diseño más robusto en cuanto al soporte de errores y un código más eficiente, ya que dichas estructuras han sido sometidas a numerosas pruebas.

Como herramienta para el desarrollo de código se ha utilizado Eclipse, que es actualmente el IDE más potente del mercado, superando incluso al clásico JBuilder. Se trata de una herramienta de código abierto, por lo que existen disponibles para ella muchos plug-ins. De entre los plug-ins disponibles se han utilizado:

- *Visual Editor*: que tiene auto-generación de código a partir de la inserción de componentes gráficos (similar a los entornos "Builder"). Sin embargo, al restringirse a las clases por defecto de Swing y SWT, no nos otorga la potencia que necesitamos para la generación de los modelos DEVS. Por ello, y como será explicado, también se ha utilizado la librería gráfica Jgraph.

- *Omondo UML*: este potente plug-in es uno de los diversos programas de diseño UML disponibles actualmente (junto con, por ejemplo, el Borland Together y el Rational Rose). La alta integración con Eclipse y sus herramientas nos hicieron inclinarnos por este para componer nuestros diagramas UML.

A continuación procedemos a explicar cada una de las etapas del desarrollo en detalle.

Investigación: Especificación DEVS

La primera tarea que tuvimos que llevar a cabo, fue la de conocer y comprender el formalismo DEVS. Para ello, se siguió el texto de Zeigler[9], concretamente los capítulos 1 y 2.

Aunque el alcance de nuestro proyecto no requería más que el conocimiento de la estructura de los modelos, y a pesar de que no teníamos experiencia con el modelado ni la simulación de sistemas, también consultamos capítulos más avanzados, para tener una visión más completa, aunque general, del formalismo DEVS.

Una vez terminada esta etapa, la siguiente labor consistió en transformar la estructura de los modelos en esquemas XML, lo cual marcaría las pautas a seguir a la hora de diseñar e implementar la herramienta.

Investigación: XML y XML-Schema

La siguiente fase de investigación abarcó el lenguaje de marcado XML (e**X**tensible **M**arkup **L**anguage), cuya sintaxis y estructura no estudiamos en las asignaturas de la facultad. Este paso previo al diseño es fundamental para estructurar la ya estudiada especificación DEVS.

XML es un lenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C), que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil. Por estas razones fue el formato elegido en nuestro proyecto, atendiendo a las prioridades definidas.

[9] Bernard P. Zeigler y Hessam S. Sarjoughian "Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models"

Ya que una de las características principales de nuestro editor será la posibilidad de salvar y cargar modelos DEVS especificados en XML, debimos definir completamente la estructura de dichos diseños XML. Para ello, nada mejor que utilizar un XML-Schema.

Los XML-Schema permiten definir las restricciones y condiciones de un archivo XML, permitiendo comprobar la integridad de los datos en cualquier momento. Así conseguimos obtener información de características generales de un documento mediante la caracterización de pequeños detalles. XML Schema sobrepone muchas de las limitaciones y debilidades de los DTDs, utilizados también para la definición estructural de documentos SGML (de los que XML es un ejemplo concreto).

Para comprender mejor la relación entre XML y XML-Schema se muestra un pequeño ejemplo, para después saltar a nuestro propio XML-Schema del formalismo DEVS.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Estudiante">
            <xsd:complexType>
                    <xsd:sequence>
                            <xsd:element name="Sexo" type="xs:boolean" default="true"/>
                            <xsd:element name="Edad" type="xsd:integer" use="optional"/>
                    </xsd:sequence>
                    <xsd:attribute name=" Nombre " type="xsd:string"/>
            </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Estudiante Nombre="Carlos">
        <Sexo>false</Sexo>
        <Edad>24</Edad>
</Estudiante>
```

Figura 4.3.1: Instancia de documento XML estructurado según el XML-Schema previo

A continuación se muestra el XML-Schema final que se utilizó en nuestro proyecto. Los modelos que se construyen son guardados en documentos XML que siguen este esquema. Además, cualquier XML que exprese un modelo DEVS, y que se desee que pueda ser visualizado en nuestro editor, deberá también cumplir esta estructura. Esto permitiría realizar un traductor que, desde cualquier herramienta que siga el formalismo DEVS, permitiera exportar modelos que fueran utilizados en nuestro editor.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- definition of complex types -->

<!-- Item -->
<xs:complexType name="Item">
        <xs:attribute name="Name" type= "xs:string"/>
        <xs:attribute name="Class" type= "xs:string"/>
        <xs:attribute name="Constructor" type= "xs:string"/>
</xs:complexType>

<!-- ListElement -->
<xs:complexType name="ListElement">
        <xs:sequence>
                <xs:element name= "Item" type="Item" minOccurs= "1"/>
        </xs:sequence>
        <xs:attribute name="Name" type= "xs:string"/>
        <xs:attribute name="Description" type= "xs:string" use="optional"/>
</xs:complexType>

<!-- List -->
<xs:element name="List">
    <xs:complexType>
        <xs:sequence>
```

```
                    <xs:element name= "ListElement" type="ListElement"
minOccurs= "1"/>
            </xs:sequence>
    </xs:complexType>
</xs:element>

<!-- Parameter -->
<xs:element name="Parameter">
    <xs:complexType>
            <xs:attribute name="Name" type= "xs:string"/>
            <xs:attribute name="Class" type= "xs:string"/>
            <xs:attribute name="Constructor" type= "xs:string"/>
            <xs:attribute name="Description" type= "xs:string"/>
    </xs:complexType>
</xs:element>


<!-- Parameters -->
<xs:element name="Parameters">
  <xs:complexType>
    <xs:sequence>
            <xs:element name="Parameter" ref= "Parameter"/>
            <xs:element name="List" ref= "List"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>


<!-- InternalConnection -->
<xs:element name="InternalConnection">
    <xs:complexType>
            <xs:attribute name="ComponentFrom" type= "xs:integer"/>
            <xs:attribute name="PortFrom" type= "xs:integer"/>
            <xs:attribute name="ComponentTo" type= "xs:integer"/>
```

```
            <xs:attribute name="PortTo" type= "xs:integer"/>
        </xs:complexType>
    </xs:element>


    <!-- DEVS Atomic Model -->
    <xs:element name="Atomic">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Parameters" ref= "Parameters"/>

            </xs:sequence>
            <xs:attribute name="Id" type= "xs:string"/>
            <xs:attribute name="Name" type= "xs:string"/>
            <xs:attribute name="InPuts" type= "xs:string"/>
            <xs:attribute name="OutPuts" type= "xs:string"/>
            <xs:attribute name="Class" type= "xs:string"/>
            <xs:attribute name="Description" type= "xs:string"/>

        </xs:complexType>
    </xs:element>

    <!-- DEVS Coupled Model -->
    <xs:element name="Coupled">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Atomics">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="Atomic" ref=
"Atomic"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
```

```
                    <xs:element name="InternalConnections">
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element name="InternalConnection"
    ref= "InternalConnection"/>
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="Name" type= "xs:string"/>
                <xs:attribute name="InPorts" type= "xs:string"/>
                <xs:attribute name="OutPorts" type= "xs:string"/>
            </xs:complexType>
        </xs:element>
        </xs:schema>
```

Con fines ilustrativos, para mayor comprensión del XML-Schema expuesto, y de forma equivalente al ejemplo previo, se incluye a continuación un ejemplo de un modelo acoplado en XML. Este ejemplo puede copiarse a un archivo XML independiente y ser cargado sin problema por nuestro editor gráfico:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Coupled Id="0" Name="root" InPorts="0" OutPorts="0">
        <Description>
        Documentación relativa al modelo acoplado raíz.
        </Description>
        <Atomic Id="0" Name="fxu" InPuts="2" OutPuts="1"
Class="base.atomics.math.Function">
                <Description>
                </Description>
                <Parameters>
                        <List Name="FuntionArray" Description="">
                        <Item Name="x0Dot" Class="java.lang.String"
Constructor="x_1"/>
                        <Item Name="x1Dot" Class="java.lang.String" Constructor="-
0.1*x_1"/>
                        <Item Name="x2Dot" Class="java.lang.String"
Constructor="x_3"/>
```

```
                        <Item Name="x3Dot" Class="java.lang.String" Constructor="-
9.8-0.1*x_3-(if(floor(10-x_0)>x_2,1,0))*(30*x_3+100000*(x_2-floor(10-x_0)))"/>
                        </List>
                </Parameters>
        </Atomic>
        <Coupled Id="1" Name="Integrador y Gxu" InPorts="1" OutPorts="2">
                <Atomic Id="0" Name="integrator" InPuts="1" OutPuts="1"
Class="base.atomics.continuous.Integrator">
                        <Description>
                        </Description>
                        <Parameters>
                        <List Name="InitialState" Description="">
                                <Item Name="x00" Class="java.lang.Double"
Constructor="0.575"/>
                                <Item Name="x01" Class="java.lang.Double"
Constructor="0.5"/>
                                <Item Name="x02" Class="java.lang.Double"
Constructor="10.5"/>
                                <Item Name="x03" Class="java.lang.Double"
Constructor="0"/>
                                </List>
                                <Parameter Name="sampleTime"
Class="java.lang.Double" Constructor="0.0001" Description=""/>
                        </Parameters>
                </Atomic>
                <Atomic Id="1" Name="gxu" InPuts="2" OutPuts="1"
Class="base.atomics.math.Function">
                        <Description>
                        </Description>
                        <Parameters>
                                <List Name="FuntionArray" Description="">
                                <Item Name="y" Class="java.lang.String"
Constructor="x_2"/>
                                </List>
                        </Parameters>
                </Atomic>
                <InternalConnection ComponentFrom="0" PortFrom="0"
ComponentTo="1" PortTo="0"/>
                <ExternalInputConnection ComponentFrom="-1" PortFrom="0"
ComponentTo="0" PortTo="0"/>
                <ExternalOutputConnection ComponentFrom="1" PortFrom="0"
ComponentTo="-1" PortTo="0"/>
                <ExternalOutputConnection ComponentFrom="0" PortFrom="0"
ComponentTo="-1" PortTo="1"/>
        </Coupled>
        <Atomic Id="2" Name="scope" InPuts="1" OutPuts="1"
Class="base.atomics.sinks.Scope">
                <Description>
                </Description>
                <Parameters>
```

```
                        <Parameter Name="topTitle" Class="java.lang.String"
Constructor="Título" Description=""/>
                        <Parameter Name="title" Class="java.lang.String"
Constructor="Pelota" Description=""/>
                        <Parameter Name="xLabel" Class="java.lang.String"
Constructor="Time (s)" Description=""/>
                        <Parameter Name="yLabel" Class="java.lang.String"
Constructor="Estado" Description=""/>
                        <List Name="Series" Description="">
                        <Item Name="serie0" Class="java.lang.String"
Constructor="PosiciónY"/>
                        </List>
                </Parameters>
        </Atomic>
        <InternalConnection ComponentFrom="0" PortFrom="0" ComponentTo="1"
PortTo="0"/>
        <InternalConnection ComponentFrom="1" PortFrom="0" ComponentTo="2"
PortTo="0"/>
        <InternalConnection ComponentFrom="1" PortFrom="1" ComponentTo="0"
PortTo="0"/>
</Coupled>
```

Investigación: Librerías gráficas. Elección de *Jgraph*

Para la implementación del editor la necesidad de unas librerías gráficas de soporte era evidente. Por tanto, la siguiente fase consistió en investigar múltiples librerías gráficas disponibles para Java, cuyos usos y funcionalidades difieren notablemente. Al no existir un estándar comúnmente aceptado para el desarrollo de entornos gráficos de alto nivel, fue muy difícil ir tomando distintas decisiones para descartar librerías.

La librería (biblioteca) gráfica de Sun nativa en Java ***AWT*** dejó hace mucho de ser el referente al ser sustituida por la mucho más potente ***Swing,*** también desarrollada por Sun. Ésta también está dejando de ser el estándar ampliamente utilizado debido a sus múltiples deficiencias en lo q a eficiencia se refiere.

Por ello, han surgido algunas alternativas, como la propia ***SWT*** de Sun, que persigue una mayor rapidez sin pasar por la máquina virtual. Son unas librerías de código abierto, potenciadas por el proyecto Eclipse. Las librerías SWT (a diferencia de Swing y las antiguas AWT) no son portables, y necesitan ser compiladas para cada sistema (MacOS/X, Linux, win32, ...). Actualmente existen ya compilaciones nativas para la gran mayoría de los sistemas, pero para permitir un amplio espectro de ámbitos donde ejecutar la aplicación, dudamos si elegirlas o no. Además, el hecho de estar mucho más familiarizados con Swing, empujaba a su uso por la experiencia acumulada. Por todo ello, preferimos descartar SWT.

Además, dado el bajo nivel de Swing y SWT, existen múltiples librerías gráficas que se apoyan en alguna de estas para dotar de mayores funcionalidades, y de un marco de trabajo al diseño de interfaces gráficas de usuario de alto nivel. Las facilidades que implementan estas librerías hacen muy recomendable su uso. No obstante, la elección de limitarnos a herramientas de código abierto restringió considerablemente las posibilidades.

Dentro de las librerías con licencia comercial descartadas encontramos SWT/Swing Designer, Jvider, Advanced SWT designer, y la famosa Jigloo. Jgraph resultó ser la más potente de las opciones de código abierto, pero en principio fue aparcada al preferir un marco de trabajo (*framework*) completo, cuyo uso simplifica enormemente la creación de un editor gráfico.

El principal framework investigado fue el plug-in para Eclipse ***GEF: Graphical Editing Framework***, que permite al programador tomar un diseño y un modelo, y dotarle rápidamente de un rico editor gráfico. Se apoya en Draw2D (que a su vez se basa en SWT), parte fundamental del framework que permite dibujar los modelos y dibujar las conexiones sin tener que preocuparnos de su implementación. Aunque Draw2D era exactamente lo que necesitábamos, la otra parte de GEF, el framework que sigue el patrón MVC, sólo permite construir aplicaciones integradas en el IDE de Eclipse. Esto sería por un lado negativo, porque requeriría de Eclipse para funcionar (lo que se estaría haciendo sería un plugin de Eclipse que realizaría modelos DEVS). Pensamos entonces en usar sólo la parte de GEF del Draw2D, pero entonces no tendríamos la gran cantidad de facilidades que proporciona un framework (y seguiríamos requiriendo las SWT). Por todo ello se descartó el uso de de GEF.

Otro plug-in muy útil de Eclipse que sí que se ha utilizado para el desarrollo de la interfaz gráfica, ha sido Visual Editor, que tiene auto-generación de código a partir de la inserción de componentes gráficos (similar a los entornos “Builder”). Sin embargo, al restringirse a las clases por defecto de Swing y SWT, no nos otorga la potencia que necesitamos. Por ello, también sigue siendo necesario el uso de una librería gráfica compleja.

Una vez descartados los frameworks, se optó por retomar el uso de la librería gráfica Jgraph. Ésta es la más potente y amigable de las librerías analizadas para Java, que cumplen todos los estándares de código abierto. Está basada en Swing, e integrada completamente; es compatible con Java 5.0, e implementa el patrón MVC: Model-View-Controller.

Además, ofrece las siguientes características (para familiarizarnos con su terminología, se utilizará Edge para nombrar a las conexiones y Cell para los nodos):

- Edición de Edges
- Mover/cambiar de tamaño
- Selección múltiple
- Zoom
- Estructura por capas
- Agrupamiento de hijos, usando interfaz arbórea
- "Grid" para dibujar
- Edición de texto empotrada
- Visualización de atributos
- Puertos (flotantes)
- Layout para los gráficos
- Manejadores flexibles para las modificaciones de las celdas
- Arrastrar y soltar
- Portapapeles
- Deshacer/Rehacer
- Look-and-Feel
- Routing
- Soporte de visibilidad
- Gestión de complejidad en varios niveles
- Coordenadas de precisión y doble precisión
- Etiquetas
- Curvas splines y bezier con n puntos de control
- Comportamiento personalizable de los atributos mapeados

Evolución del diseño

En este apartado presentaremos un estudio de las fases por las que ha pasado el diseño de nuestro proyecto. En el primer subapartado, utilizaremos diagramas UML para facilitar la comprensión, presentando tanto el diseño del modelo como el de la interfaz por medio de estos diagramas. En el segundo subapartado haremos mención a las principales clases de la librería JGraph que hemos utilizado. Finalmente presentamos un estudio detallado de las fases de diseño por las que ha pasado la interfaz gráfica de usuario (GUI) de nuestro editor.

Diseño UML

En este punto detallaremos cómo ha ido evolucionando el diseño del editor a medida que se adquiría un mayor conocimiento de la potencia que nos ofrecían las bibliotecas gráficas utilizadas.

Diseño preliminar

Una vez superadas las fases anteriores procedimos a realizar el diseño UML de lo que sería nuestra aplicación. Dado que dicho diseño sufrió diversos cambios a medida que avanzó el proyecto, se ha decidido explicar aquí el primer (y más claro) de los diseños UML para a continuación aumentar la complejidad fusionando este diseño con las estructuras de la librería gráfica utilizada.

El paso fundamental consistía en codificar el formalismo DEVS en forma de clases con funcionalidades acotadas. El haber estructurado el XML-Schema previamente facilitó enormemente la tarea, pues las clases se correspondieron a los elementos allí definidos. Aún así, se requirió un esfuerzo de diseño para la conexión adecuada entre las clases (asociación y herencia) y para definir nuevas clases necesarias. Los diseños UML correspondientes son:

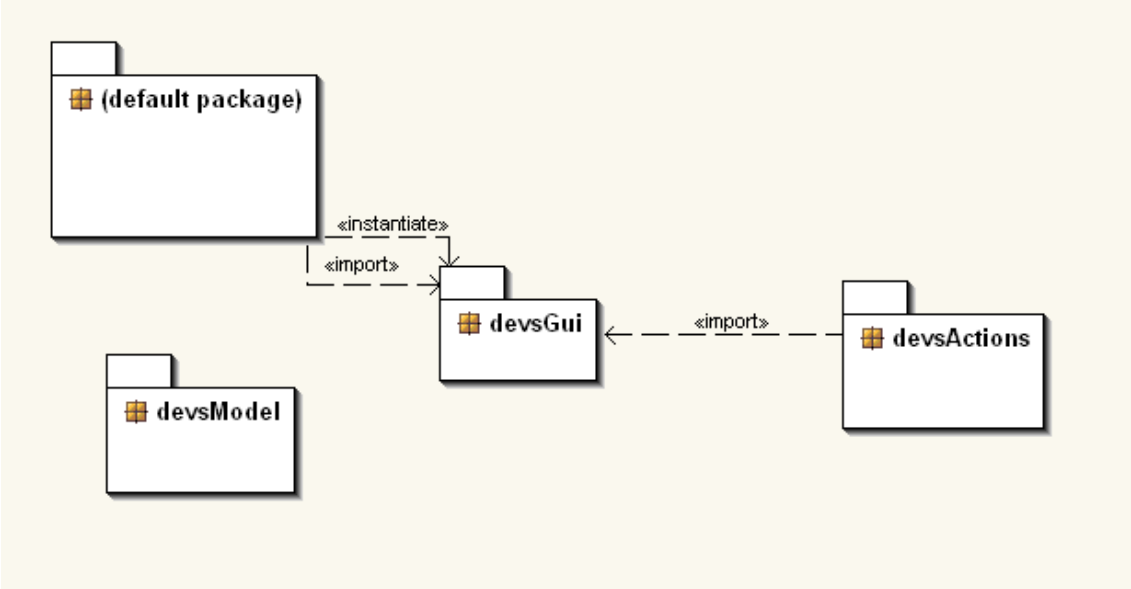

Figura 4.5.1.1.1 Diagrama de paquetes preliminar

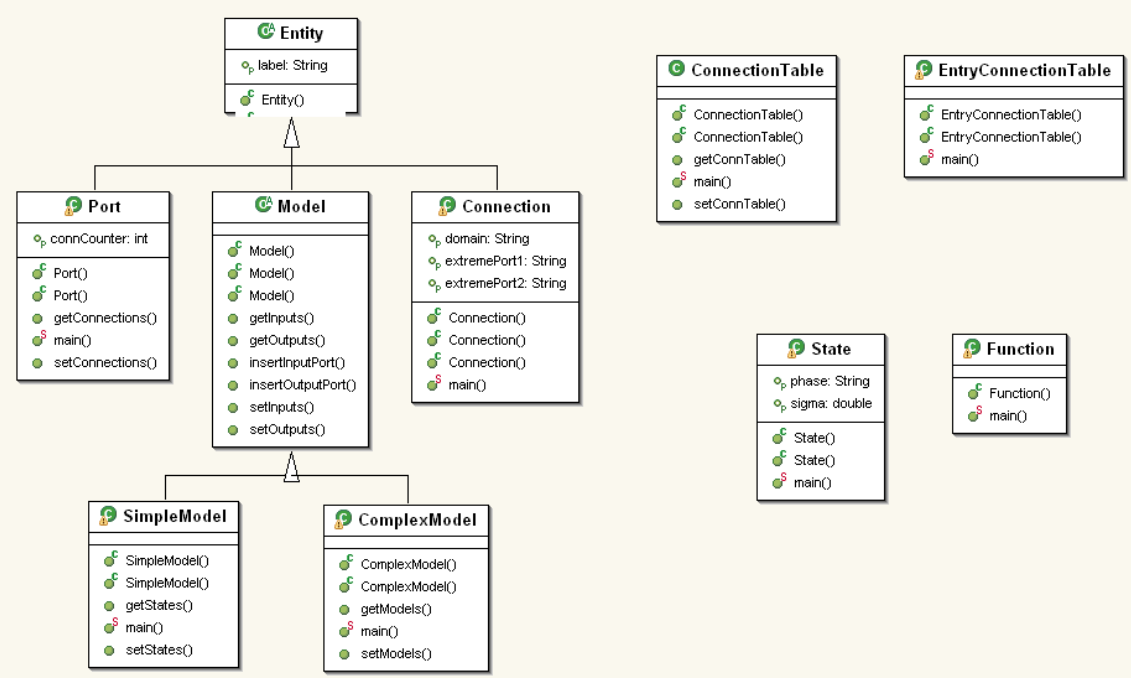

Figura 4.5.1.1.2 Diagrama de herencia del modelo, preliminar

Este diagrama recoge la estructura que surge de forma natural del formalismo DEVS. Así, existen modelos simples o atómicos (SimpleModel) y complejos (ComplexModel) con varios simples en su interior. Ambos tienen un comportamiento común que se condensa en la clase abstracta Model, de la que heredan. Existen conexiones (Connection) que conectan los distintos modelos simples o complejos, además de puertos (Port) que las agrupan. Además, todo elemento con propiedades

gráficas tendrá elementos comunes, recogidos en la clase abstracta Entity ("son entidades"). Asimismo, existen estados (State) y funciones (Function) dentro de los modelos simples, pero al no tener apariencia gráfica no son "entidades". El resto de clases se explican a continuación.

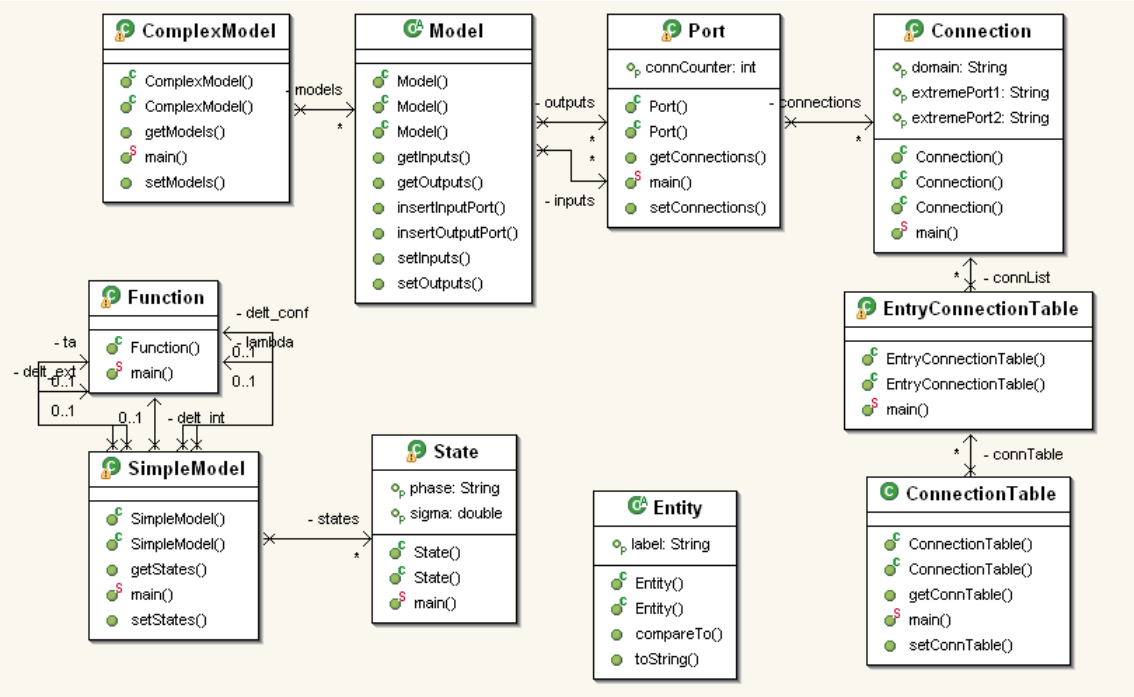

Figura 4.5.1.1.3 Diagrama de asociación del modelo, preliminar

Este diagrama UML muestra la composición de las distintas clases entre sí. Así, puede observarse cómo un modelo complejo contiene un conjunto de "modelos", sean simples o, a su vez, complejos. Cada modelo (Model) contiene una serie de puertos de entrada y de puertos de salida, y a su vez cada puerto varias conexiones. Además, se ha definido una tabla de conexiones (ConnectionTable) donde especificar los modelos conectados. Los modelos simples/atómicos tienen varias funciones y un conjunto de estados.

No se presenta el diseño de la GUI dado que ésta no ha sufrido cambios relevantes en el diseño final, y allí se expone con claridad éste. Cabe destacar que desde la creación del primer diseño se tomó la decisión de ajustarse al patrón MVC (Modelo-Vista-Controlador), ya que tenemos modelos a los que asociar propiedades gráficas, y una manera sencilla y fácilmente ampliable de implementar estas características es el uso de dicho patrón.

Diseño Final[10]

Como ya se ha comentado en el apartado 4.4, decidimos utilizar como librería gráfica *JGraph*. Inicialmente desconocíamos todas sus funcionalidades, ya que no habíamos trabajado con ella antes. Pero según fuimos avanzando en nuestra labor de investigación, nos dimos cuenta de que nuestro diseño se fusionaba con ciertas estructuras de *JGraph*. Hay que hacer notar que nuestro diseño era totalmente correcto y respetaba rigurosamente el formalismo DEVS. Por tanto, llegados a este punto, tuvimos que decidir entre dos alternativas:

- **Introducir nuestro diseño completamente independiente de JGraph**: La librería nos proporciona unas estructuras que siguen el patrón MVC (que como hemos comentado es el que habíamos decidido utilizar por ser el que mejor se ajustaba a las necesidades de nuestro proyecto). Así cada celda de JGraph tiene un UserObject, que puede apuntar a un modelo definido por el usuario. Podíamos tomar las componentes gráficas de la librería y asociarles nuestros modelos según correspondiera. El problema de esta opción era que ciertas estructuras incluidas en nuestro diseño podríamos ahorrárnoslas si aprovechábamos todas las funcionalidades que nos ofrecía la librería. Es más: mucho trabajo que debería de ser programado era enormemente facilitado por las funciones del API de Jgraph.

- La segunda alternativa era **utilizar las funcionalidades que JGraph nos ofrecía y adaptar nuestro diseño** al suyo para no duplicar ciertas estructuras. Decidimos que ésta era la mejor opción. Así, por medio de herencia (como explicaremos más adelante en este punto) las clases de nuestro diseño extienden ciertas clases de la librería. Debido a que dichas clases de JGraph se estructuran según el modelo MVC, la aplicación desarrollada sigue respetando dicho patrón.

Otro de los cambios que sufrió el diseño se debió al progreso del código del simulador: con la definición exhaustiva del mismo, se decidió relegarle la tarea de

[10]A lo largo de este punto se hará referencia a varias clases de la librería JGraph, que serán comentadas en el apartado 4.5.2 JGraph

discernir los estados y funciones de los modelos atómicos, que por tanto ya no aparecerán en nuestras clases (más que como atributos del modelo atómico).

A raíz de estos dos cambios fundamentales nuestro diseño se transformó en lo que detallamos a continuación. Pero antes cabe destacar que los conceptos principales que ya aparecían en el primer diseño se mantienen, al igual que se respeta rigurosamente el formalismo DEVS. Además el nuevo diseño también está completamente integrado con el patrón MVC que proporciona *JGraph*. Lo único que se ha hecho es simplificar la estructura de clases, desechando las que eran innecesarias por aportarnos la librería otras con funcionalidades similares. Como ya se ha comentado, con esta integración se permite utilizar las funciones de Jgraph al máximo.

A continuación presentamos las clases finales del proyecto correspondientes a la implementación de la estructura que debe tener un modelo para cumplir el formalismo DEVS:

- *MyComplexNode*: Se correspondería con la antigua clase *ComplexModel*. Se identifica (hereda) con el grafo completo *JGraph* proporcionado por la librería. Sus hijos serán los nuevos nodos que se van conectando para formar un modelo. Esta estructura se correspondería con todo un lienzo del editor.

- *MyNode*: Se correspondería al concepto de modelo, ya sea este simple o complejo. Se identifica (hereda) del *DefaultGraphCell* de Jgraph, manteniendo todas sus propiedades gráficas y ampliándola con toda la información de nuestro modelo.

- *MyPort*: Se corresponde en el diseño preliminar con la clase Port. Hereda de *DefaultPort* de la librería, manteniendo todas sus propiedades gráficas y ampliándola con toda la información con la que cuenta un puerto en nuestro diseño. Cabe señalar que JGraph no distingue entre puertos de entrada y de salida. Nosotros con nuestro diseño sí lo hacemos, para que las conexiones solo puedan partir de un puerto de salida y llegar a uno de entrada. Esta distinción se realiza tanto a nivel de modelo (con atributos en la clase), como a nivel gráfico

(los puertos de entrada y salida se sitúan a izquierda y derecha de cada celda, respectivamente).

- *MySimpleModel*: No tiene herencia. Esta clase contiene una tabla de atributos. Cuando *MyNode* representa un modelo atómico, incluye (por composición) un atributo de tipo *MySimpleModel* que contiene sus atributos.

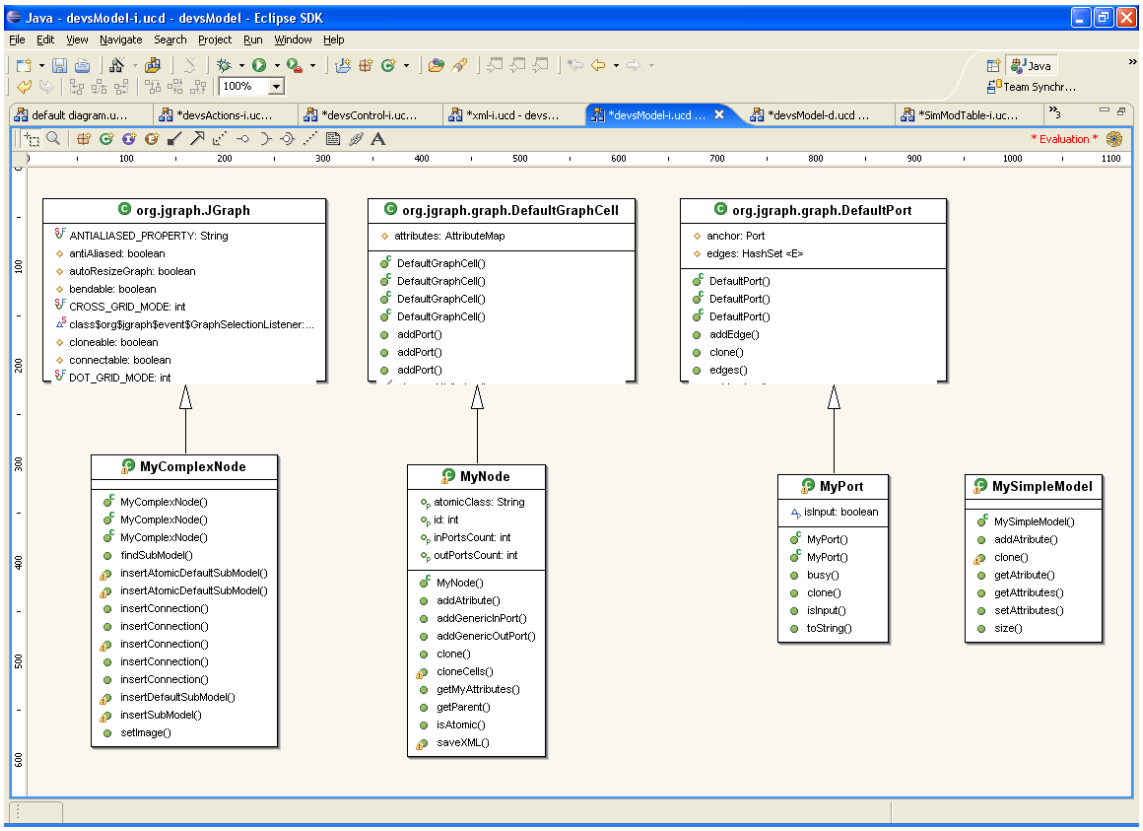

Figura 4.5.1.2.1: Diagrama de herencia del modelo (con las superclases de Jgraph)

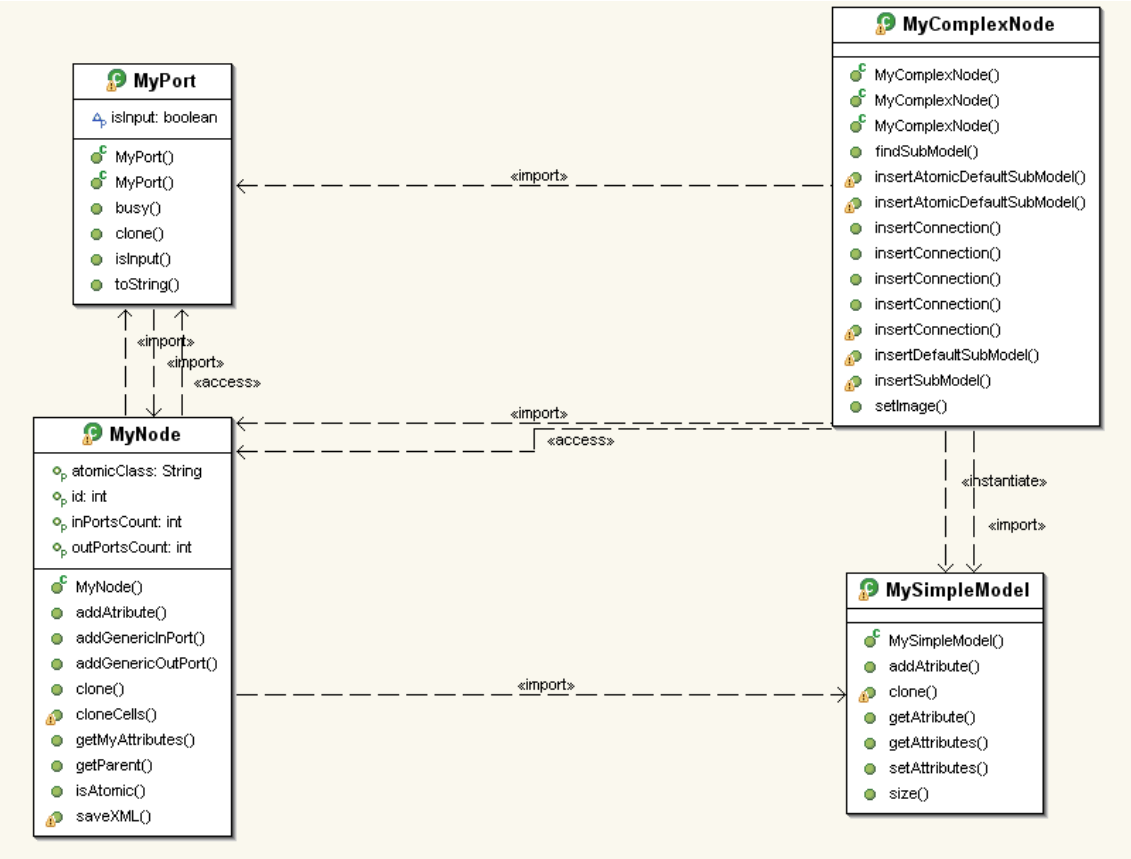

Figura 4.5.1.2.2: Diagrama de dependencia del modelo

Por último queda comentar el diagrama UML de la GUI. La clase principal es *VMainWindow*, encargada de presentar la ventana principal. Tanto esta clase como *VSplash*, se llaman desde la clase *Main*, es decir, el ejecutable. *VSplash* se ejecuta mientras carga el editor, y presenta una imagen de bienvenida. No tiene ninguna conexión con *VMainWindow*, sólo una relación de precedencia en su ejecución.

Por otro lado, la ventana principal contiene referencias a *VCanvas*, tantas como lienzos haya en el editor. Por tanto es una relación de composición. Además, cada *VCanvas* contiene a su vez una referencia a *VMainWindow*, para poder acceder a los componentes de la GUI, para que ésta se vea modificada según varíe la selección dentro del lienzo activo.

En cuanto a *Tools*, proporciona un método estático encargado de cargar imágenes de manera independiente a la plataforma. Se utiliza en la carga de los iconos de *VMainWindow* y en la imagen de bienvenida de *VSplash.*

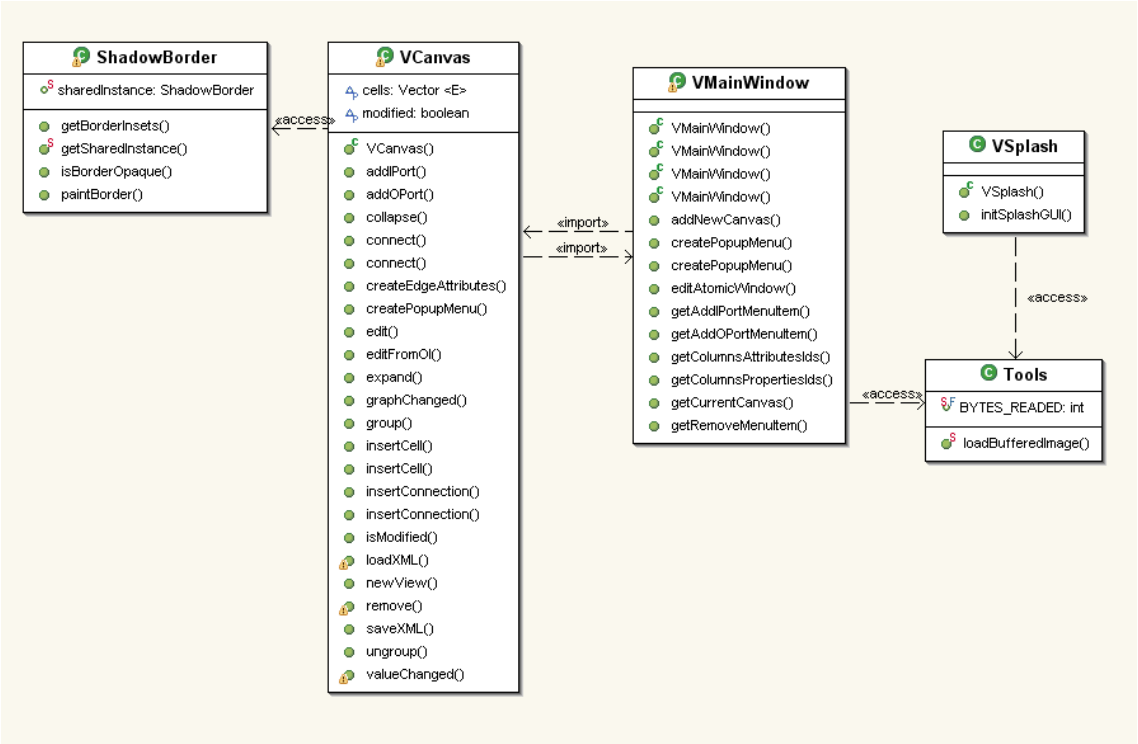

Figura 4.5.1.2.3: Diagrama de dependencia de la GUI

En conjunto, nuestro diseño ha crecido en complejidad a medida que ha avanzado el desarrollo, como puede apreciarse en el siguiente diagrama de paquetes en comparación con el del punto anterior:

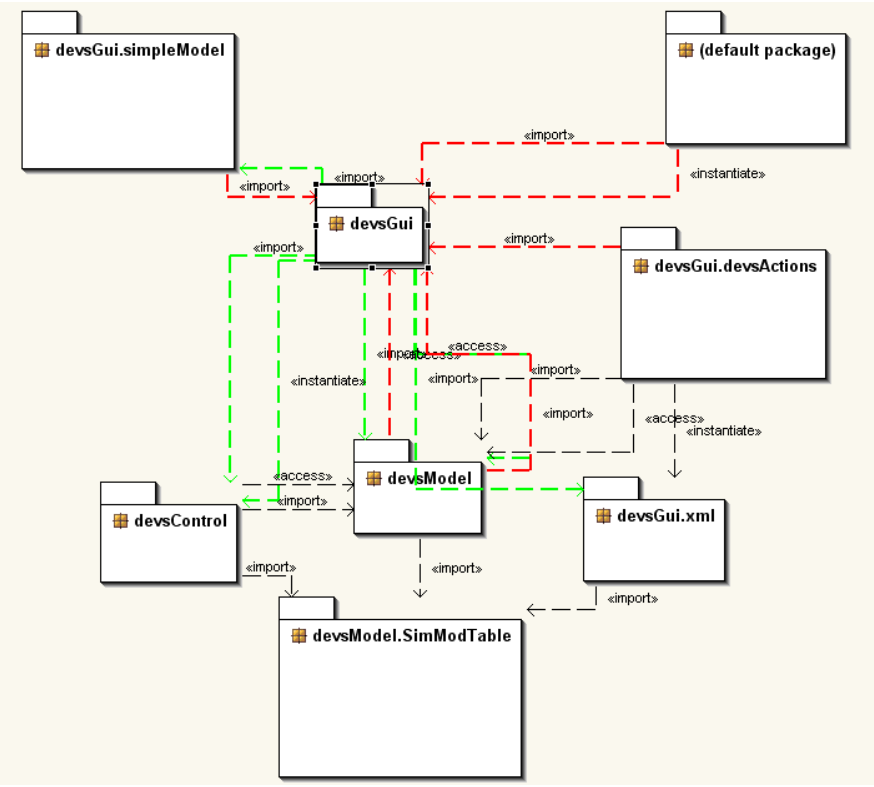

Figura 4.5.1.2.4: Diagrama de paquetes del editor

JGraph

En este apartado se explicará el diseño interno de la librería JGraph. Esta información se puede encontrar ampliada y con ejemplos de uso en el manual de usuario, ubicado en http://www.jgraph.com/pub/jgraphmanual.pdf.

Esta información es interesante únicamente para potenciales desarrolladores, que pretendan modificar o ampliar la herramienta, y hacia ellos será dirigido este apartado. Se realiza un resumen de las características utilizadas en la construcción de la herramienta, pero hay muchas otras características que no se han utilizado, y por tanto serán obviadas o descritas superficialmente. En cualquier caso se recomienda la lectura del manual de usuario.

JGraph fue pensada como una librería para dibujar grafos, dado un modelo. Para ello se apoya en las clases de Swing, y sigue sus mismos principios de diseño, por lo que cierta experiencia con Swing facilitará su asimilación. De hecho, JGraph extiende a *JComponent* y puede usarse como tal.

Un objeto JGraph está compuesto de un modelo *GraphModel* y de una vista *GraphLayoutCache*, lo cual indica que el patrón MVC[11] está presente en la implementación de la librería. Además, todos los objetos visuales del grafo pueden ser tratados como *celdas*, ya sean nodos del grafo, puertos o aristas.

4.5.2.1 Modelo

En el modelo, la información se guarda de manera arborescente, siendo la raíz el atributo *root*. Así, en la ilustración 4.5.2.1.1 se representa un grafo en el que hay 2 nodos, con un puerto cada uno, y una arista. Las aristas tienen referencias a los puertos fuente y destino. Esta estructura jerárquica será muy útil cuando las celdas se agrupen y se colapsen.

[11] Modelo Vista Controlador: Patrón de diseño que separa los gráficos (la vista) de los datos (el modelo), y los conecta mediante un controlador que se encarga de la comunicación entre ambas partes.

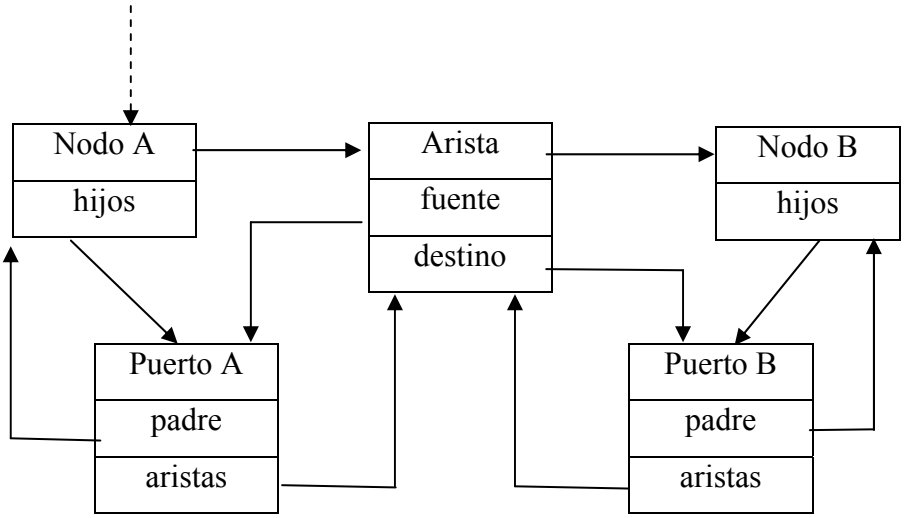

Ilustración 4.5.2.1.1 – Estructura de GraphModel

Con esta estructura ya se puede definir un modelo DEVS acoplado, pero es necesario aplicar ciertas restricciones, como hacer que los puertos sean unidireccionales. Estos cambios se detallan en el siguiente apartado.

Como se ha comentado antes, todos los elementos son tratados como celdas, y éstas tienen un conjunto de atributos, que se encarga de almacenar los atributos visuales de cada celda, tales como posición, tamaño, color, etc. Curiosamente, esta información se almacena en el modelo, en lugar de la vista, como suele ser habitual. Esto es debido a que JGraph se centra en la visualización de grafos, más que en el análisis de grafos, por lo que sacrifica rendimiento en esos casos para facilitar ciertas tareas de visualización.

Los atributos visuales se almacenan en la estructura *AttributeMap*, que funciona como una tabla con claves y valores. Las claves vienen dadas por la clase *GraphConstants*, y especifican todos los atributos que pueden ser modificados. En cuanto a la información que no encaja en ese conjunto, JGraph proporciona un *objeto de usuario*, en el cual se puede encapsular cualquier información. Por defecto se inserta la etiqueta que identifica al la celda mediante un *String*, que puede ser vacío.

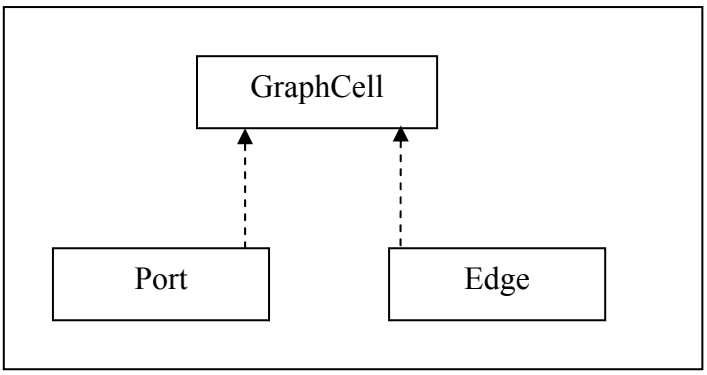

Ilustración 4.5.2.1.2 – Jerarquía de interfaces de celda

GraphCell es el interfaz que implementa la clase *DefaultGraphCell*, que representa a los nodos, mientras que los puertos y las aristas se consideran especializaciones de los nodos, y tienen que implementar los interfaces *Porte* y *Edge*, respectivamente, que derivan de *GraphCell.* Las clases que implementan estos dos nuevos interfaces son *DefaultPort* y *DefaultEdge*, y obviamente derivan de *DefaultGraphCell*. Por tanto, si se quieren definir restricciones en los grafos, como es el caso del editor, es necesario redefinir estas clases, simplemente sobrescribiendo los métodos involucrados en las restricciones.

4.5.2.2 Vista

Por otro lado tenemos la clase *GraphLayoutCache*, que almacena las vistas de las celdas, es decir, de los nodos, puertos y aristas. Contiene también una correspondencia entre las celdas y su vista, y este es el único sitio en JGraph donde se puede seguir la dirección modelo-vista. Esta clase permite tener varias vistas distintas e independientes de un mismo modelo, lo que facilita la expansión y colapso de celdas.

Para poder trabajar con las vistas de las celdas, JGraph ofrece una serie de métodos, en los que se recibe como entrada la celda (o celdas) en cuestión. Estos métodos suelen devolver una referencia de tipo *CellView*. Esta clase permite cambiar la apariencia de las celdas, pero a distinto nivel que los atributos. Por ejemplo, podemos hacer que los nodos se dibujen como elipses en lugar de rectángulos, o cambiar las formas que aparecen en los extremos de las aristas.

CellView es un interfaz que implementan las clases *PortView*, *VertexView* y *EdgeView*, las cuales se encargan de la vista de cada tipo de celda. Son estas clases las que se deberían extender o sustituir para cambiar la vista de algún componente.

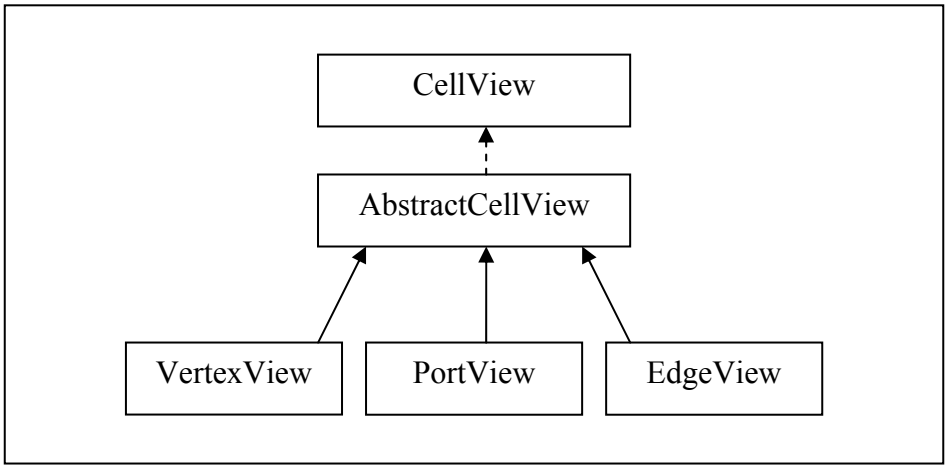

Ilustración 4.5.2.2.1 – Jerarquía de interfaces de vista de celda

4.5.2.3 Interacción

Por último, sólo queda definir las posibles interacciones del usuario con los grafos. Esta característica es una de las grandes ventajas de JGraph, ya que la mayor parte de la funcionalidad está implementada. Al estar pensada como una librería de visualización, las acciones típicas como desplazar, redimensionar, mover ortogonalmente, etc., ya se encuentran en el comportamiento por defecto.

Los eventos disparados por JGraph vienen dados por cambios en el modelo, o bien por cambios en la vista. En ambos casos los motivos de las notificaciones son por adición, substracción o modificación de alguna de las celdas. Estos cambios se aplican automáticamente en las dos partes.

Además, existe un evento asociado a la selección, que se lanza cuando ésta varía. Este evento es muy importante, ya que permite saber qué celdas están seleccionadas, si lo están, y qué hacer con esa selección. Por ejemplo, una selección de varias celdas ha de permitir agruparlas, mientras que si sólo se ha seleccionado una, el botón debe estar desactivado. Para ello basta con implementar el interfaz *GraphSelectionListener*, cuyo método *valueChanged()* tiene la responsabilidad de actuar en consecuencia.

Interfaz gráfica de usuario (GUI)

4.5.3.1 Especificación Inicial

La especificación inicial del editor, a nivel de interfaz de usuario, se basaba en el esquema descrito en la Ilustración 4.5.3.1.1 Constaba de una barra de menú, con las acciones típicas de cualquier aplicación de escritorio, más las necesarias para este tipo de editor. También incluía una barra de herramientas, con los accesos directos a los comandos más importantes de la barra de menú. El inspector de objetos se situaba a la izquierda, mientras que la vista ocupaba el mayor espacio de la ventana. No se precisaba la gestión de las vistas, si se trataban ventanas internas independientes o pestañas, dejando la elección al desarrollador.

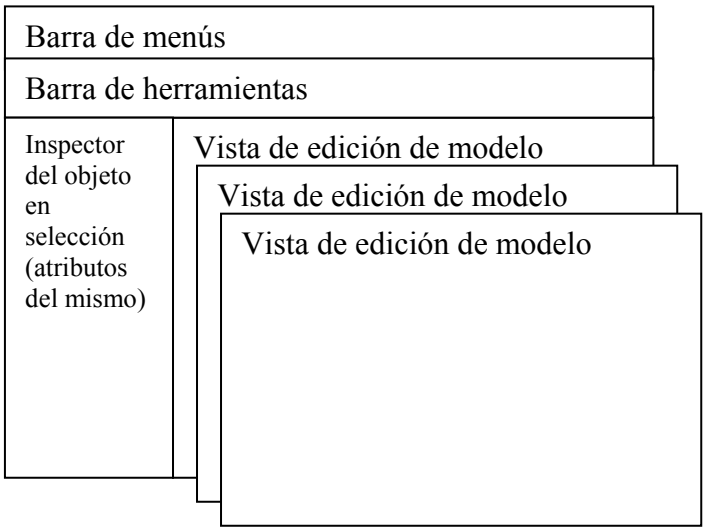

Ilustración 4.5.3.1.1 – Especificación Inicial

Esta especificación se fue transformando a lo largo de las sucesivas reuniones, añadiendo nuevos elementos, como la librería, y modificando la disposición de los elementos para mejorar la usabilidad.

En conclusión, la especificación se ceñía a unas reglas muy básicas, dejando como responsabilidad del desarrollador todos los detalles de diseño y de implementación.

4.5.3.2 Prototipo

El primer prototipo funcional de la interfaz gráfica de usuario incluía ya ciertas mejoras con respecto a la especificación inicial, como se aprecia en la Ilustración 4.5.3.2.1. En esta primera versión ya se incluye un esbozo de la librería, que se encarga de leer los directorios y ficheros XML que se encuentran en el directorio *library/* de la aplicación y generar las pestañas y los botones necesarios.

El directorio *library/* contiene subdirectorios que agrupan distintos modelos DEVS según su utilidad, y que son en realidad ficheros XML que describen los atributos gráficos y las propiedades de los modelos. En este ejemplo el directorio *library/* contiene:

```
Library/
      BasicElements/
            atomic.xml
            coupled.xml
            inport.xml
            outpor.xml
      Continuos/
            integrator.xml
            sin.xml
      Sources/
            sineWave.xml
      Links/
            scope.xml
```

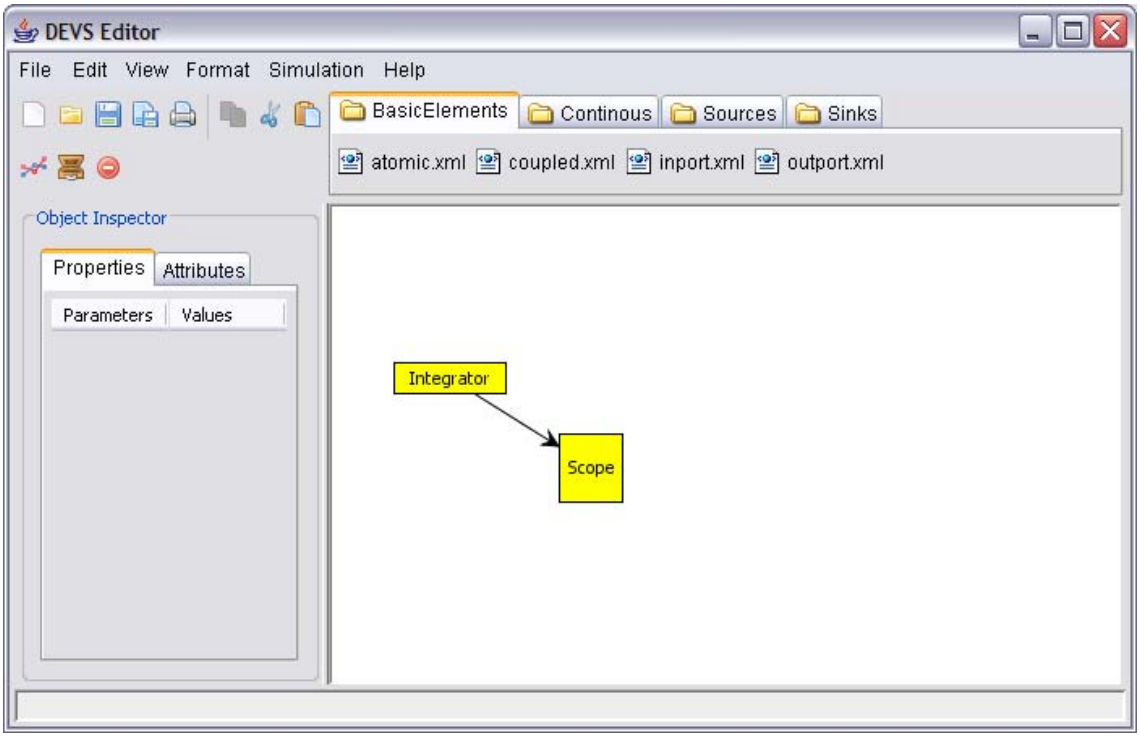

Ilustración 4.5.3.2.1 - Prototipo

Otra mejora consiste en la inclusión de una barra de estado en la parte inferior de la ventana, que proporciona información adicional a los comandos del menú, según se pasa el cursor por encima de ellos. En cuanto a las vistas, aún no se incluye ningún método para gestionar varios lienzos, funcionalidad que se incluyó posteriormente. El inspector de objetos se sitúa a la izquierda de la ventana, pero carecía de funcionalidad en ese momento.

Aparentemente, esta primera versión ya contenía gran parte de los requisitos de la especificación.

4.5.3.3 Diseño Final

Finalmente, la ventana principal del editor tiene el aspecto detallado en la ilustración 4.5.3.3.1. La barra de menú y la barra de herramientas permanecen en la parte superior de la ventana, aunque se han añadido todas las funciones que faltaban.

La librería se ha visto modificada, tanto en su posición como en su representación. Ahora los botones siguen un modelo de árbol de directorios, más cercano a la organización física en el disco, en busca de facilitar su identificación al usuario.

En la última versión se incluye el navegador entre lienzos, y se ha optado por gestionarlo mediante pestañas, muy populares últimamente. Esta decisión se fundamenta sobre todo en la comodidad por parte del usuario para pasar de un lienzo a otro, y evitar redimensionar y minimizar las ventanas internas, que se barajaron como otra alternativa.

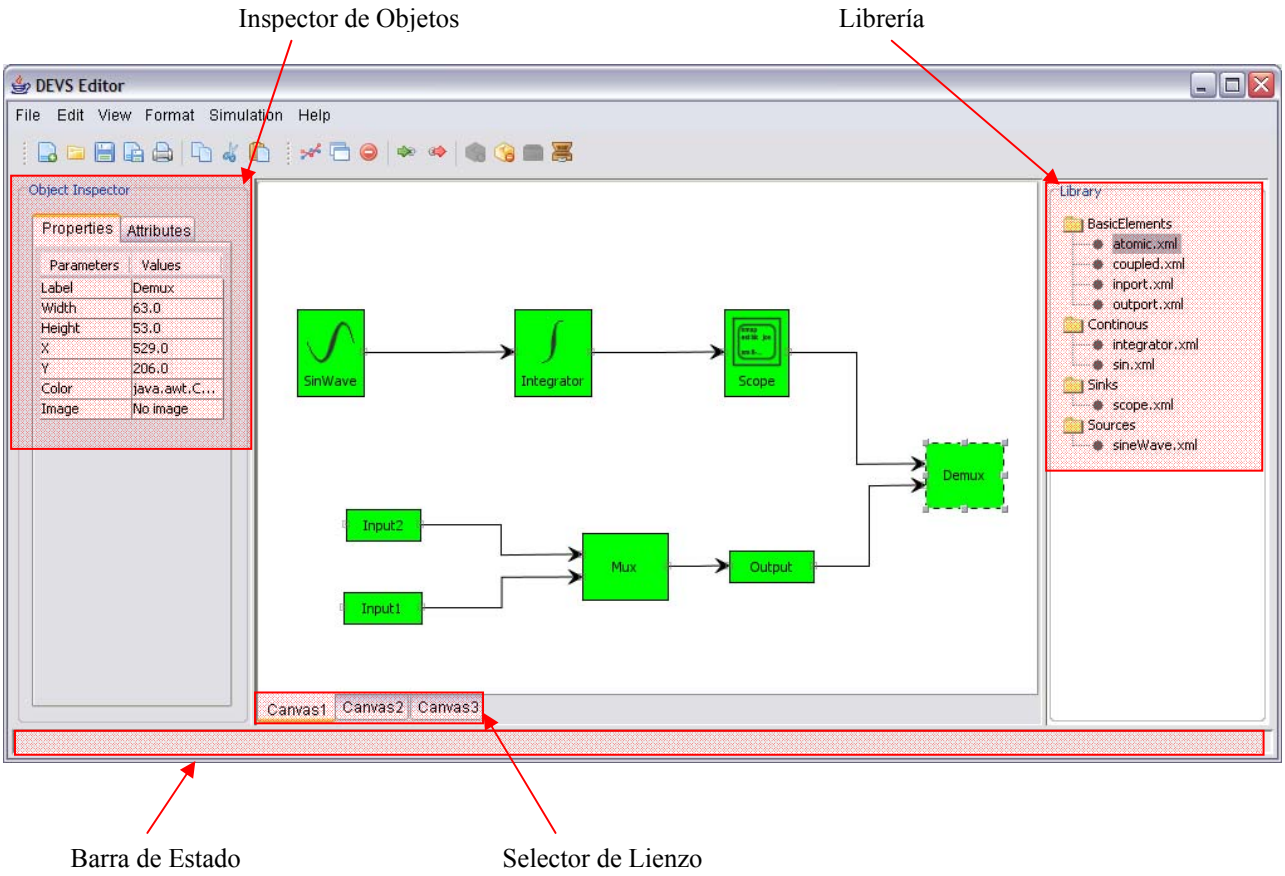

Ilustración 4.5.3.3.1 – Diseño Final (Ventana Principal)

Por otro lado, se ha incluido un menú emergente contextual, que se activa con la pulsación del botón derecho del ratón. El menú varía según el elemento o elementos seleccionados en el lienzo, o según dónde se pulse. Así, al pulsar el botón derecho sobre

un modelo atómico, el menú contendrá las opciones comunes, como copiar, pegar o borrar, y una adicional exclusiva, la de editar modelo atómico. Esto se reproduce en los diferentes elementos de la interfaz, y permite el acceso a las diferentes operaciones de muy diversas maneras. Un detalle de este menú se observa en la ilustración 4.5.3.3.2

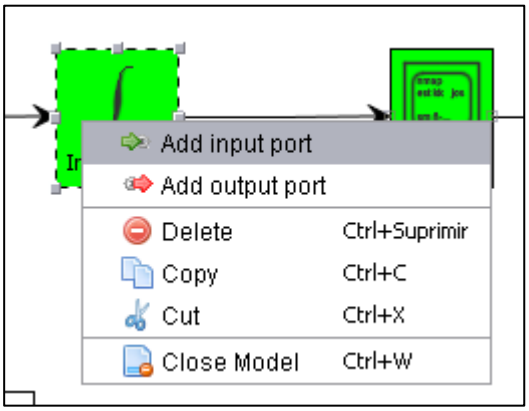

Ilustración 4.5.3.3.2 – Detalle de menú emergente contextual

Para aumentar la facilidad de uso, objetivo de toda interfaz gráfica de usuario, se han incluido iconos en la gran mayoría de las operaciones, para que con el uso, el usuario identifique la operación visualmente, y sólo con ver el icono sepa cual es la opción que busca. Como fuente de los iconos se ha usado una librería con licencia Creative Commons, lo que nos ha permitido modificarlos para adaptar su aspecto a las necesidades específicas de la aplicación.

Los detalles de implementación se detallan más adelante en el capítulo 4.6.2.

Implementación

Aplicación del patrón MVC en grafos

Como se explicó en la sección 4.5.2, la librería JGraph está implementada siguiendo el patrón de diseño MVC, separando en dos partes el modelo y la vista. Gracias a ello no ha sido necesario realizar toda la infraestructura, sólo en algunos puntos determinados.

El controlador de JGraph es totalmente transparente para el desarrollador, es decir, si se aplican cambios a alguna de las dos partes, ya sea el modelo o la vista, la otra se actualiza con los nuevos cambios automáticamente. Este comportamiento se ha aprovechado, ya que al crear clases adaptadoras que se encuentran por encima de las clases de JGraph, se obtiene las funcionalidades de las mismas, añadiendo además las requeridas por el editor. Estas clases adaptadoras se definieron en la sección 4.5.1.

Gracias al soporte de JGraph, las clases encargadas de la gestión de los modelos son independientes entre sí, cumpliendo los objetivos de encapsulamiento y modularización. De todas maneras, ha sido necesario resolver otros conflictos similares, como la comunicación del inspector de objetos con los lienzos. Esta comunicación se realiza realmente entre el modelo de la celda seleccionada y el modelo de la tabla, que forma parte de Swing y también sigue el patrón MVC.

Para acometer esta tarea se han implementado varias clases, siguiendo el omnipresente MVC. La primera cuestión era el unificar ambos modelos en una estructura que fuera comprensible para los elementos involucrados en la comunicación.

Para ello se han implementado los interfaces *TableModel* y *GraphCell*, utilizando en la medida de lo posible las clases abstractas proporcionadas por los APIs. De esta manera, la lectura y modificación de los datos se realiza de manera transparente, sin necesidad de invocar de manera externa los métodos o extraer la información manualmente.

En resumen, JGraph ha demostrado ser una muy buena elección, permitiendo seguir un diseño modular y desacoplado sin tener que crear una infraestructura previa, evitando los riesgos que ello conlleva.

Aplicación del patrón COMMAND en GUI

Antes de nada, se detallarán los paquetes, clases y árbol de directorios que intervienen en la implementación de la interfaz gráfica de usuario (GUI en adelante).

`devsGui/`	Paquete principal. Contiene las clases más importantes
`devsActions/`	Contiene las clases que implementan las operaciones de la GUI
`simpleModel/`	Contiene las clases que gestionan el editor de modelos atómicos
`xml/`	Contiene las clases que se encargan de salvar y cargar los modelos en formato XML

Las clases del paquete *devsGui* son estas:

`devsGui/ShadowBorder.java`	Implementa bordes sombreados.
`VMainWindow.java`	En esta clase se implementa la ventana principal. Se encarga de dibujar la ventana y todos sus elementos.
`VCanvas.java`	Esta clase representa un lienzo. Contiene toda la funcionalidad de la GUI. Se encarga de representar los grafos. Es la clase más importante de toda la aplicación.
`VSplash.java`	Muestra una imagen al cargar la aplicación.
`Tools.java`	Contiene un cargador de imágenes multiplataforma.

Siendo rigurosos, comentaremos únicamente las clases directamente relacionadas con la representación de las ventanas, ya que son prácticamente independientes de la parte funcional del editor.

4.6.2.1 Clase VMainWindow

La organización de esta clase está basada en la que genera el plugin *Visual Editor de Eclipse*. Éste genera atributos privados para todos los elementos de la GUI, así como sus métodos accesores. Es decir, un botón determinado es definido por un atributo privado inicializado a *null* y un método accesor que genera el botón en sí, si éste no ha sido creado previamente. Esta peculiaridad es particularmente útil, ya que se puede reutilizar código al crear los diversos menús. Por ejemplo, el menú contextual se construye con los mismos ítems que se encuentran en el menú principal, evitando así la repetición innecesaria de código.

El problema de ésta organización es que el código así generado es bastante grande, pero modulariza la construcción de la ventana, permitiendo localizar los fallos rápidamente.

Para organizar todo el contenido, se ha hecho un uso intensivo de paneles, anidándolos y empleando la clase *BorderLayout*. En la ilustración 4.6.2.1.1 se detalla el alcance de éstos. El panel central se organiza mediante *JSplitPane*, de manera anidada, y dando a la zona de los lienzos la prioridad de cambiar de tamaño, al redimensionar la ventana.

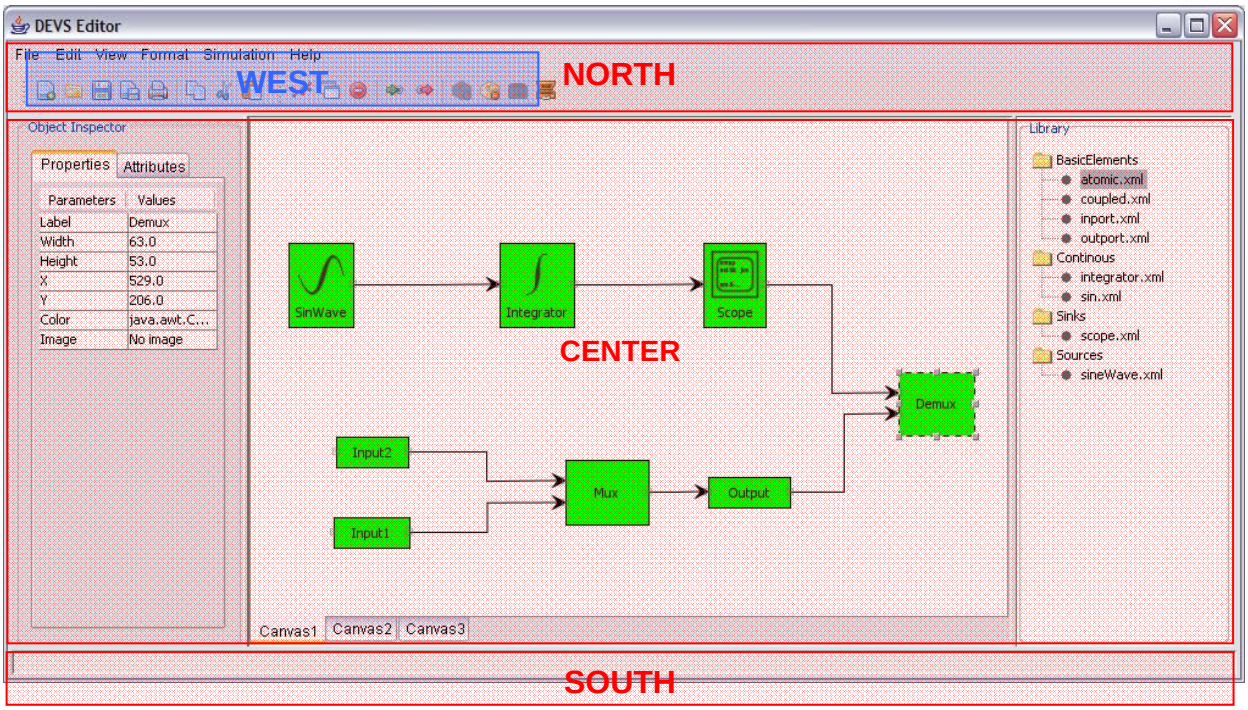

Ilustración 4.6.2.1.1 – Organización

Por otro lado tenemos el comportamiento funcional, con acciones y oyentes. Se ha implementado una clase *AbstractActionDefault*, de la que derivan todas las acciones de la GUI. Así, cada operación, se realiza con un objeto externo, que, o bien invoca a una función externa que realiza la operación, o bien la realiza él mismo. Depende de la propia naturaleza de la operación. Con esto conseguimos que la conexión entre la parte meramente gráfica y la funcional esté aislada en esas clases.

Esta implementación está basada en el patrón *Command*, aunque con ciertas modificaciones. Esto permite el desarrollo incremental de las operaciones. Se pueden ir añadiendo nuevas operaciones en la GUI, e ir implementando su funcionalidad a posteriori, a medida que se va construyendo la aplicación. De esta manera, se puede tener construida la GUI completamente antes de haber implementado la parte funcional.

En cuanto al comportamiento de la barra de estado, la clase *StatusBarMouseListener* se encarga de mostrar información relativa al componente que esté bajo el foco del cursor. Esta acción se implementa creando referencias a la clase anterior, pasando como parámetro el texto informativo. De esa manera centralizamos en una sola clase el comportamiento, mientras que la información depende del objeto sobre el que esté el ratón.

En resumen, se ha intentado construir una interfaz que sea fácilmente ampliable. Para demostrar este hecho, veamos un ejemplo:

Supongamos que queremos añadir una operación que nos distribuya los nodos del grafo de manera arborescente. Para acceder a esa opción queremos añadir un ítem al menú Edit, un ítem al menú contextual emergente, y un botón en la barra de herramientas.

Crearemos un JMenuItem y un JButton, para los menús y para la barra de herramientas, respectivamente. También se debe añadir la acción asociada a la operación, para ello creamos una nueva clase que herede de AbstractActionDefault y que implemente el método actionPerformed(). En concreto, este método invocará a una función (que se implementará a posteriori) miembro de VCanvas, que es la clase responsable de mostrar los grafos. El último atributo que hemos de añadir será del tipo StatusBarMouseListener, que lo inicializaremos con el texto informativo que queramos.

Una vez hecho esto, se han de implementar los métodos accesores al ítem y al botón. En estos métodos se define el aspecto, el texto y los oyentes, que hemos declarado previamente. Si queremos añadir un icono, utilizaremos la función Tools.loadBufferedImage(), para asegurar el correcto funcionamiento en todas las plataformas. Por último, se añaden estos componentes mediante la instrucción componente.add(getBotonOItem(). En el menú Edit, añadiendo la línea en el método getMenuItem(), en el menú contextual, añadimos ***el mismo ítem*** *en la función createPopup(), y en la barra de herramientas añadimos el botón al método getMoreToolbar().*

Sólo quedaría implementar en VCanvas la funcionalidad pedida.

4.6.2.2 Clase AbstractActionDefault

Esta clase extiende *AbstractAction* y lo único que añade es una referencia a la ventana principal. Pertenece al paquete *devsGui.devsActions.* De esta manera podemos invocar a funciones de la propia *VMainWindow*, si la acción en cuestión modifica algún parámetro de la ventana principal, o podemos invocar funciones del lienzo activo, si la operación hace referencia al grafo.

4.6.2.3 Clase MyMarqueeHandler

Esta clase es importante, ya que redefine la interacción del usuario con las celdas del grafo. Pertenece al paquete *devsGui.devsActions*. Sobrescribe los métodos m*ousePressed(), mouseReleased(), mouseDragged()* y *mouseMoved()*. Cualquier acción que se ejecute mediante alguno de esos eventos, ha de ser definida en esta clase. Por ejemplo, el menú contextual asociado a las celdas se invoca desde esta clase, pero se construye en la ventana principal.

5. RESULTADO FINAL

Metodología utilizada

A lo largo del desarrollo del proyecto se han tenido muy en cuenta las cuestiones metodológicas. Como se ha podido observar a lo largo del contenido del punto 4, se ha preferido realizar un diseño exhaustivo en **UML** antes de comenzar el proceso de implementación. De esta forma, lejos del *Extreme Programming*, profundas discusiones sobre el papel evitaron tener que re-estructurar sucesivas veces el código programado.

Además, como también se pudo observar en el punto 4, se ha seguido una implementación siguiendo distintos **patrones de diseño**. Al no poseer los miembros del equipo formación al respecto, esto requirió una nueva labor investigadora. Buenos ejemplos serían los ya comentados Model-View-Controller (MVC) y Command. La librería Jgraph utilizada a su vez implementa el MVC.

Al utilizar como lenguaje de programación Java, se han utilizado numerosas **estructuras del API** proporcionado por Sun. Como estructuras de datos complejas, hemos utilizado principalmente tablas hash, árboles, distintos tipos de listas y colecciones, e iteradores. Como estructuras gráficas, múltiples componentes de Swing. Además, se ha hecho un uso exhaustivo de las posibilidades que nos ofrece el API de Jgraph. A continuación se detallan los principales usos de algunas estructuras:

- *java.util.**Hashtable*** (a veces referenciado como *java.util.Map*): para almacenar por clave los atributos del modelo atómico, consiguiendo acceso constante a sus componentes.
- *java.util.**TreeSet*** (a veces referenciado como *java.util.Set*): para gestionar los "edge" de cada puerto
- *java.util.**ArrayList*** *o* *java.util.**Vector*** : para distintos usos, como implementaciones de List.
- *java.util.**Iterator***: para recorrer en bucles las listas, por ejemplo la lista de puertos de un nodo

- *javax.swing.tree.**DefaultMutableTreeNode*** (a veces referenciado como *javax.swing.tree.MutableTreeNode*): como clase padre de *DefaultGraphCell*, hemos utilizado varios de sus métodos.
- *java.io.**PrintWriter***: para salvar el archivo XML.
- *java.io.**File***: para construir la librería leyendo los subdirectorios.

Ejemplo

En este ejemplo cargaremos y simularemos el conocido Atractor de Lorentz. El atractor de Lorenz, concepto introducido por Edward Lorenz en 1963, es un sistema dinámico determinístico tridimensional no linear derivado de las ecuaciones simplificadas de rollos de convección que se producen en las ecuaciones dinámicas de la atmósfera terrestre. La forma de mariposa (al representarlo en tres dimensiones) de este sistema puede haber inspirado el nombre del efecto mariposa en la Teoría del Caos.

Cargamos el editor, cuya primera vista viene representada por la Figura 5.2.1

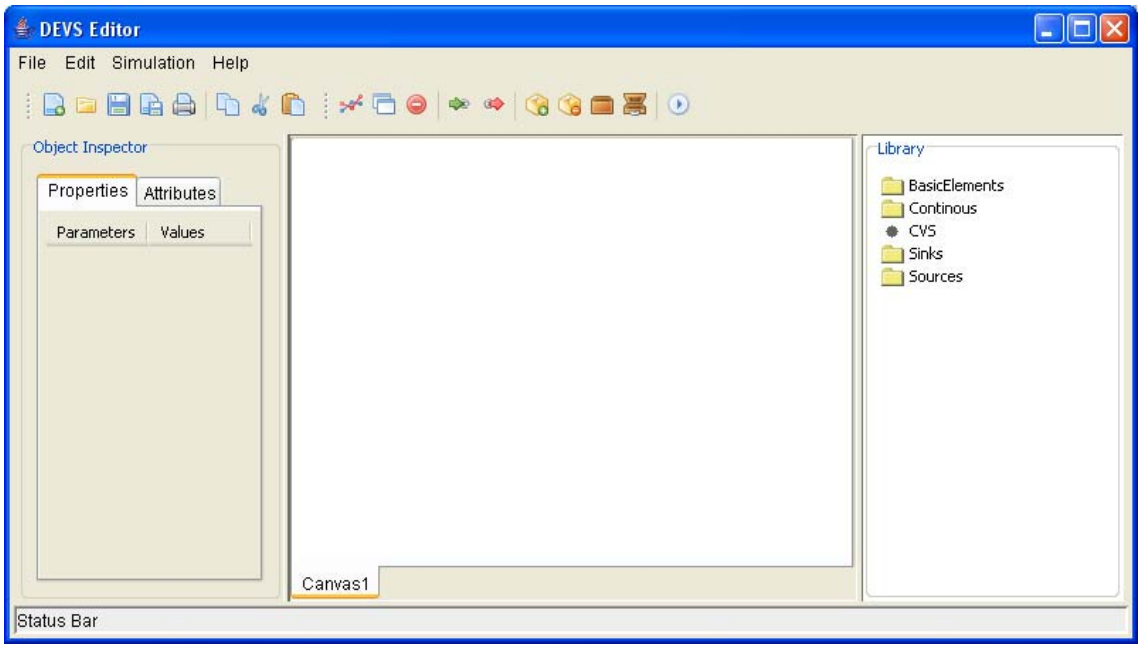

Figura 5.2.1: Vista inicial del editor gráfico

A continuación, se despliega el directorio *BasicElements* de la librería (*Library*) y seleccionamos el **AtractorLorentz.xml**, con lo que se carga el modelo en el lienzo *Canvas1*. Se pueden observar los puertos conectados entre sí por conexiones, y cuatro

modelos con su icono identificativo: dos *funciones*, un *integrador* y un *scope* al final para recoger la salida visualmente. En la captura de la Figura 5.2.2 está seleccionada la primera *función*, con lo que sus parámetros se pueden visualizar en el *Object Inspector*. Al estar seleccionada la pestaña de propiedades, se dan datos gráficos únicamente: altura, posición, icono… La pestaña de atributos proporciona los parámetros del modelo. Por ejemplo, en la Figura 5.2.3 pueden visualizarse los atributos del *scope*.

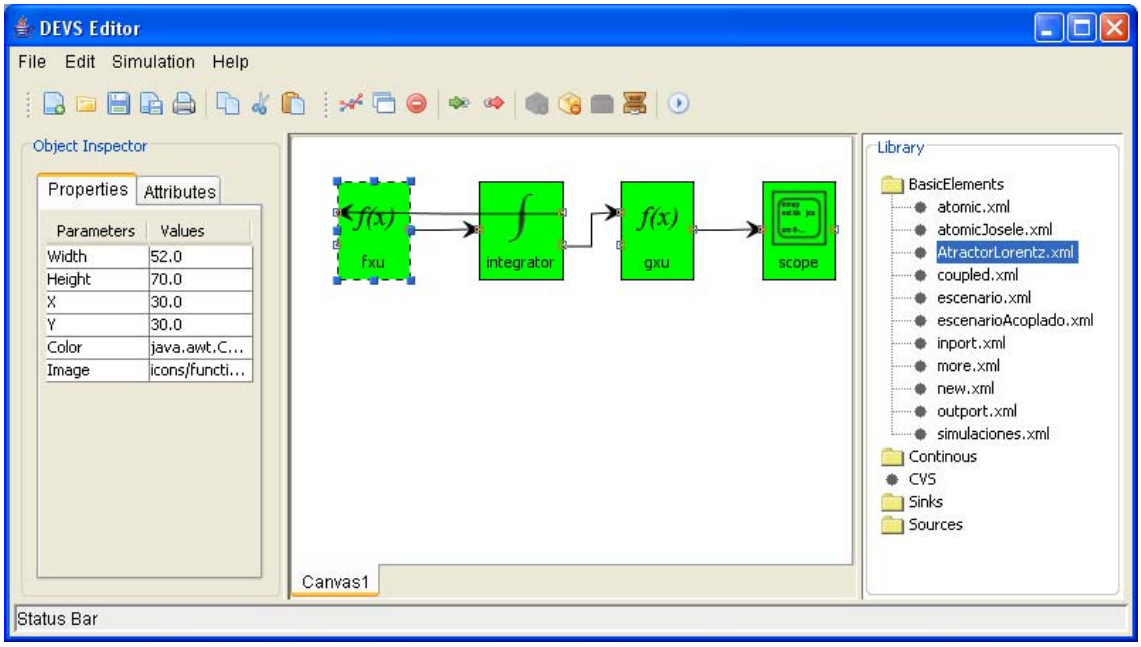

Figura 5.2.2: Atractor de Lorentz cargado en el editor

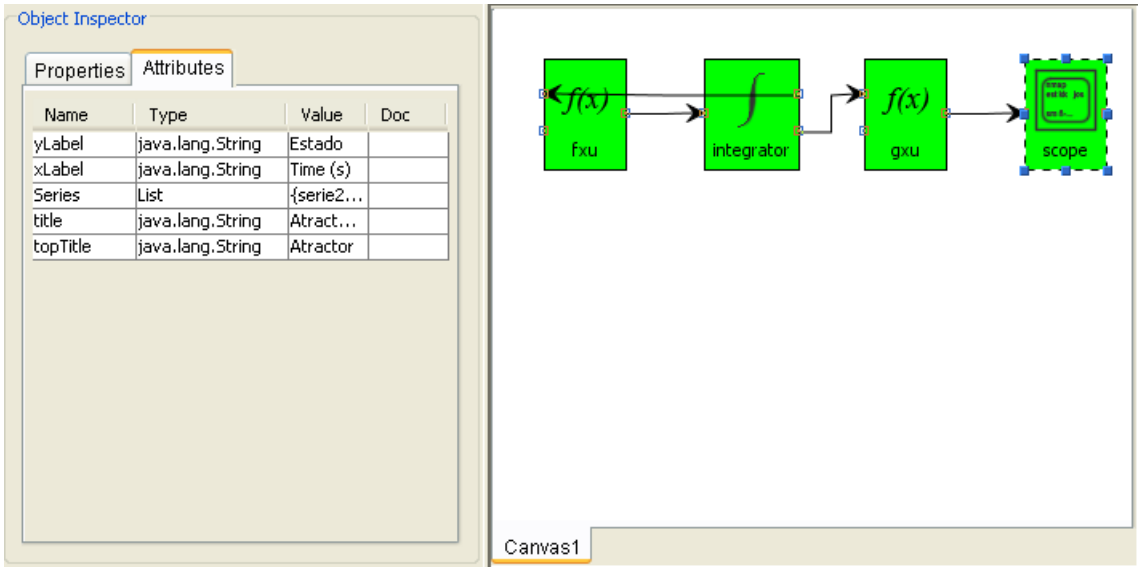

Figura 5.2.3: Detalle de los atributos de un modelo.

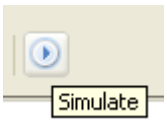

Figura 5.2.4: Detalle del botón *Simulate*

A continuación se procederá a simular dicho Atractor de Lorentz. El simulador asociado puede lanzarse desde el botón indicado en la Figura 5.2.4. Así, se abrirá el simulador preparado para ejecutar el *AtractorLorentz.xml* que tiene cargado, como puede verse en la Figura 5.2.5. A su vez, se abre la consola con el diálogo de opciones del simulador, y con la "r" se lanza en ejecución (Figura 5.2.6). El resultado de la simulación se puede visualizar en una gráfica en dos dimensiones (Figura 5.2.7), pero también podía haberse configurado el simulador para guardar su resultado en XML, como ya se ha comentado previamente.

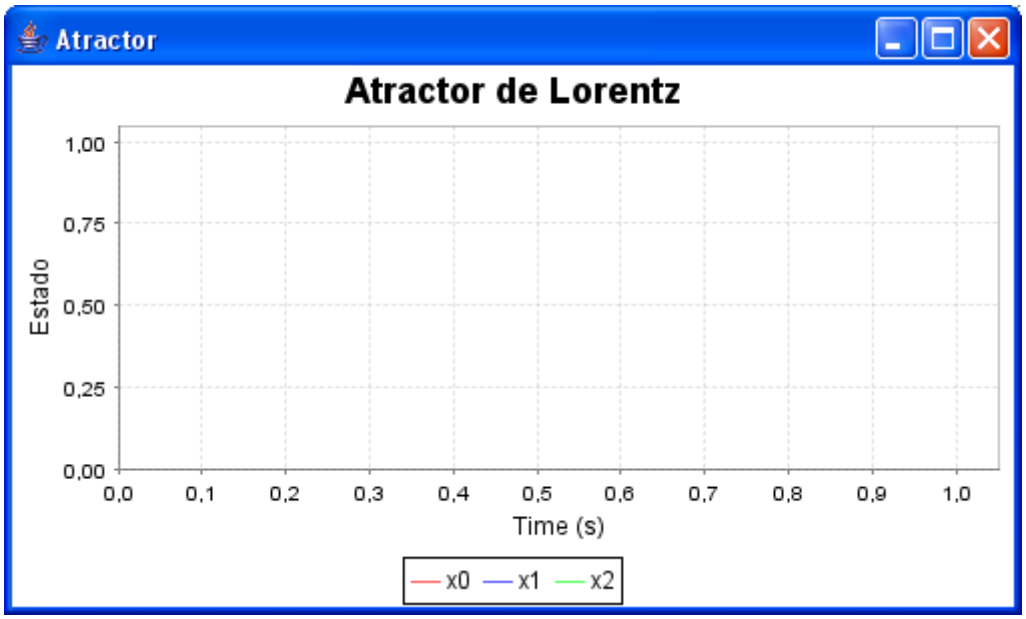

Figura 5.2.5: Simulador con el Atractor de Lorentz cargado.

```
----------------------------------------
JDEVS Simulation
José Luis Risco Martín
Jesús Manuel de la Cruz García
----------------------------------------
r: Run simulation completely
s: Run simulation in step mode
t: Run simulation in timed mode
q: Exit/Stop simulation
----------------------------------------
Your choice: r
----------------------------------------
66.59999999999945% time executed ...
Simulation ended.
```

Figura 5.2.6: Consola del simulador

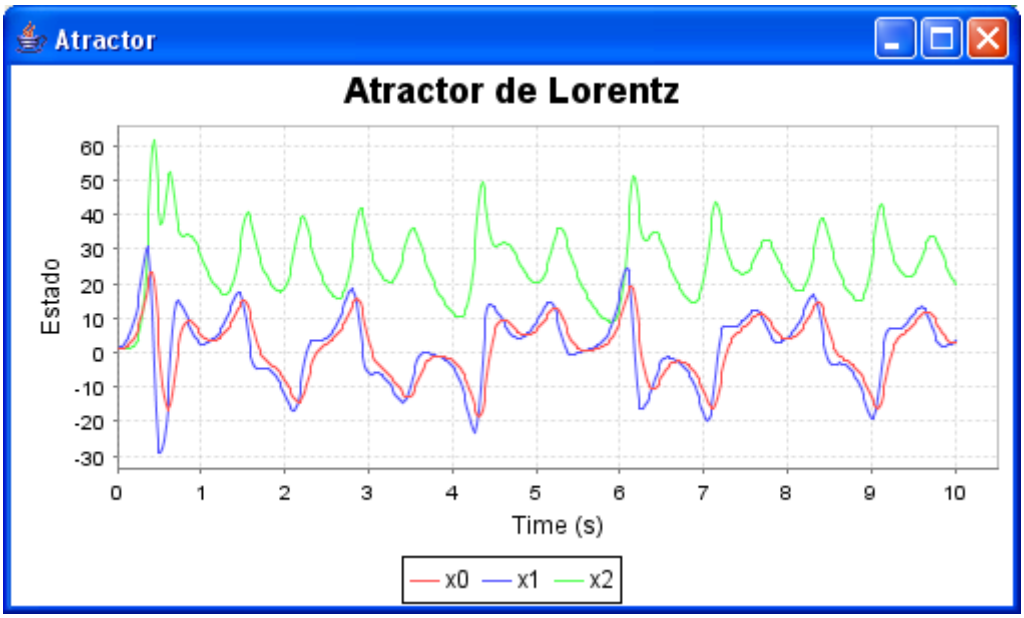

Figura 5.2.7: La gráfica en dos dimensiones del atractor de Lorentz

Posibles vías de evolución

Nuestro sistema completo, ya definido en el punto 1.3 (Figura 1.3.1), consta de dos etapas, cuyos elementos principales son el editor y el simulador. La fuente de datos del sistema, sea a través de un archivo externo, o un archivo generado por el propio editor, está siempre en formato XML. A su vez, el resultado de la simulación también se ofrece como XML. Este formato tan extendido constituye el estándar para el intercambio de información estructurada entre diferentes plataformas, permitiendo la compatibilidad entre sistemas de una manera segura, fiable y fácil. Esta característica tan fundamental se convierte en la base de futuras ampliaciones, al maximizar la modularidad del proyecto. Algunas posibles ampliaciones serían:

- Desde una herramienta cualquiera que genere modelos que sigan el formalismo DEVS, se podría construir un traductor con salida en formato XML (traductor muy útil, por otra parte, en cualquier contexto). Sólo habría que hacer que esos XML cumplieran el XML-Schema definido para que se integraran de forma natural en nuestro sistema, posibilitando la edición y posterior simulación sin dificultad alguna. Así, por ejemplo, podría generarse un traductor para los modelos del DEVSJAVA del profesor Zeigler (ya comentado en el punto 1.5) para hacerlos compatibles con nuestro editor.

- Es más, con un traductor apropiado, nuestros modelos generados en el editor en XML podrían utilizarse en el simulador de Zeigler (DEVSJAVA). Y así sucedería para cualquier simulador de modelos DEVS para el que se tuviera un traductor adecuado. De este modo el simulador quedaría sustituido sin tener que modificar ni una línea en el editor. De ahí la importancia de la modularidad en un proyecto de estas características.

- Otra posibilidad sería conectar una herramienta no ya en la fuente, sino posterior al resultado obtenido. Así, un programa de visualización de gráficas en tres dimensiones, o incluso un entorno virtual, podrían tomar sus datos fuente del resultado obtenido por el simulador. Una vez más, sin retocar el código de nuestro proyecto se pueden obtener resultados asombrosos.

- Si se optara por modificar el código para ampliar sus funcionalidades, podría aprovecharse su íntima relación con Simulink (como se comentó en el punto 1.2) para conectarlo con esta famosa herramienta. Podría hacerse a partir del XML (exportando desde Simulink) o con una interrelación más profunda modificando el código. Así, podrían simularse modelos construidos con la otra plataforma.

6. MANUAL DE USUARIO

En este manual se enumeran las características que ofrece el editor y presenta ciertas acciones comunes para empezar a trabajar con él. Para obtener una interfaz más amigable de cara al usuario, algunas funciones pueden realizarse de varios modos distintos (por ejemplo con una opción de menú y con un botón, ambos con el mismo icono…).

La siguiente ilustración muestra un ejemplo del funcionamiento de nuestro editor. La utilizaremos para definir cada una de las partes del mismo, con el fin de proporcionar una mejor comprensión de las funcionalidades que presenta.

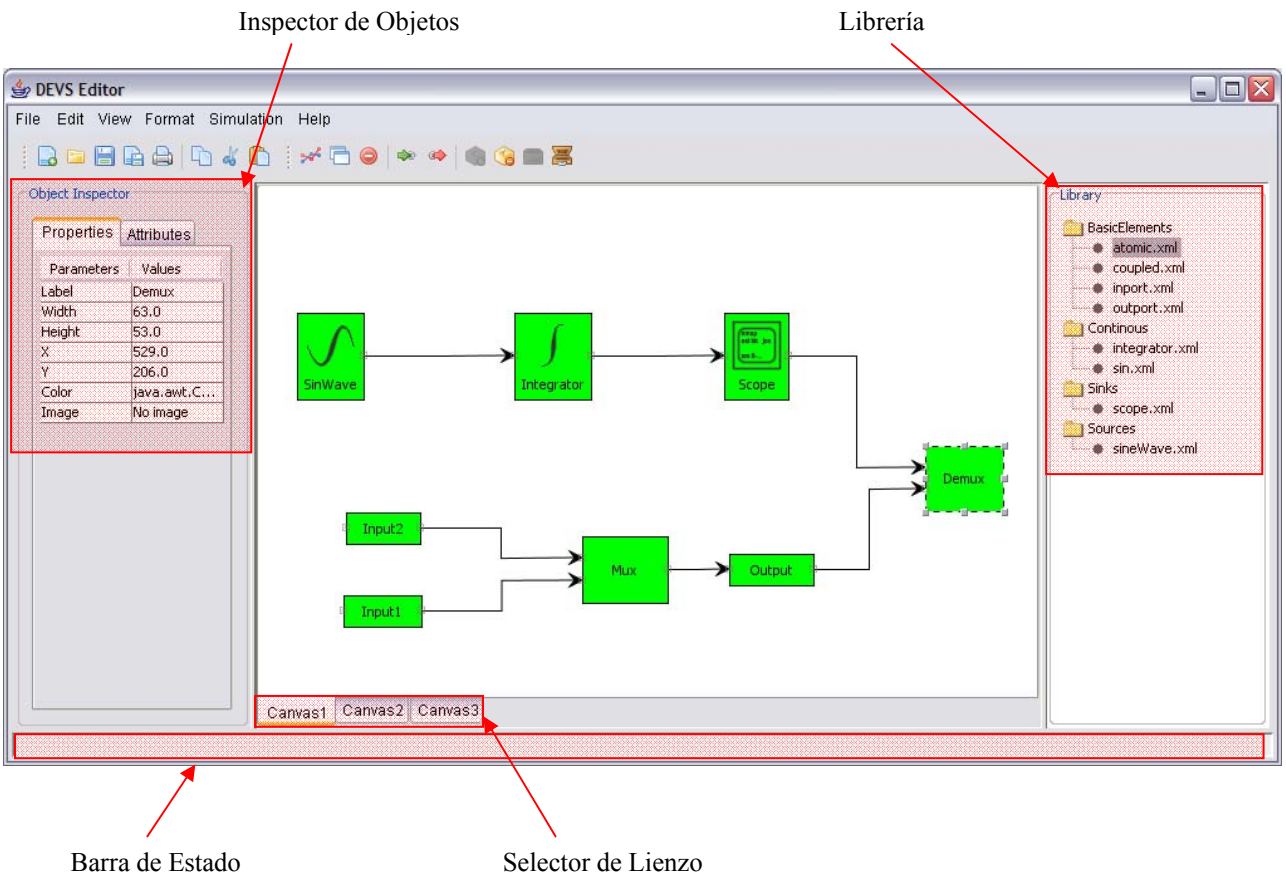

Ilustración –Ventana Principal

- La parte central del editor es lo que se denomina *lienzo*. Puede crear sobre él su modelo gráficamente, que podrá almacenar posteriormente como un archivo XML. En el lienzo los submodelos que componen el modelo acoplado se representan como cajas negras con un icono gráfico y un nombre. Estas cajas pueden conectarse unas con otras ya que disponen de puertos visibles de entrada y de salida. Además se pueden añadir dinámicamente puertos sobre cada uno de los submodelos con las opciones del menú o de la barra de herramientas que se detallarán posteriormente. También se pueden acoplar bajo una caja negra un conjunto de submodelos del lienzo, y desacoplarlos posteriormente. En la ilustración anterior aparecen tres pestañas con distintos lienzos. Éstas permiten navegar entre los distintos modelos que tengamos abiertos.

- En el *inspector de objetos* se muestran las propiedades gráficas (medidas, posición, archivo del icono…) de la caja seleccionada (en caso de haber alguna seleccionada) y los atributos del modelo que dicha caja representa. Si se trata de un modelo atómico cada uno de sus atributos cuenta con un nombre, tipo, documentación explicativa y valor por defecto (excepto en el caso del tipo *List* que no tiene valor por defecto). Tanto en los modelos atómicos como en los acoplados aparecerán también como atributos los puertos de entrada y los de salida. Los campos de la tabla que muestra los atributos del modelo son editables, así que el usuario puede modificar los atributos desde el inspector de objetos.

- En la parte superior se encuentra el *menú* con 5 opciones principales:
  - *File*: Submenú con todas las opciones permitidas para la gestión de los archivos (Abrir un nuevo lienzo, abrir un archivo XML que contenga un modelo (definido con cualquier herramienta) y cargarlo en el editor, cerrar el lienzo actualmente seleccionado, salvar, salvar como, exportar como librería, imprimir y salir de la aplicación).

  - *Edit*: Con las principales opciones de edición como son seleccionar todo, cortar, copiar, pegar, abrir la celda seleccionada en un nuevo lienzo y borrar los elementos seleccionados. Además desde este submenú se pueden asociar nuevos

atributos a un elemento atómico: si seleccionamos "Edit → Edit Atomic" aparece un formulario como el que se muestra en la figura 3.2.2:

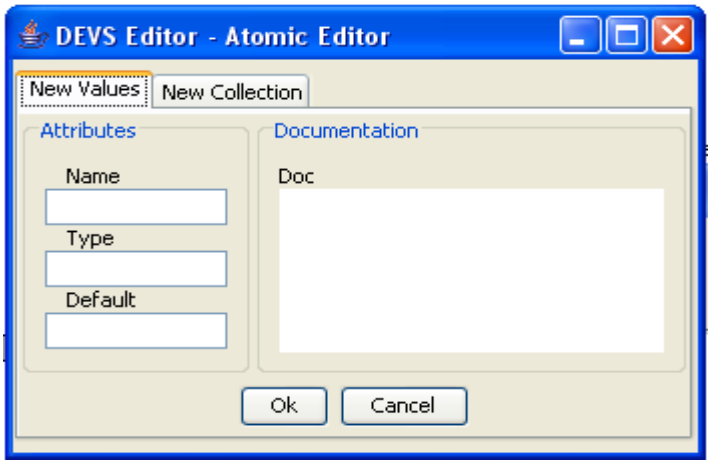

Ilustración - Formulario para asignar propiedades a un modelo atómico

Como podemos ver en la figura, el nuevo atributo tendrá un nombre, un tipo, un valor por defecto y una documentación explicativa. Dicho atributo también podrá ser una lista (en la pestaña no seleccionada), en cuyo caso tendrá nombre, tipo, y documentación explicativa.

- *Simulation*: Con la opción de lanzar el simulador, que ejecutará el XML correspondiente.

Justo bajo el menú encontramos la barra de herramientas, que contiene accesos directos a las opciones más utilizadas del menú superior. La barra de estado muestra información explicativa de cada botón, además de las etiquetas contextuales.

- *Librería (Biblioteca)*: Muestra el contenido del subdirectorio *library*, dividido en los cuatro tipos principales de modelos que existen, guardados en formato XML en los distintos directorios. El usuario podrá personalizar esta librería añadiendo nuevos directorios con los modelos atómicos o acoplados que defina.
- *Menú emergente con el botón derecho del ratón*: Para mayor comodidad del usuario, según se seleccione un modelo u otro, con el botón derecho del ratón aparece un menú emergente con distintas opciones, todas las cuales están replicadas bien en el menú superior, o bien en la barra de herramientas (y por tanto ya han sido

comentadas en este apartado). Un ejemplo del menú emergente puede verse en la ilustración:

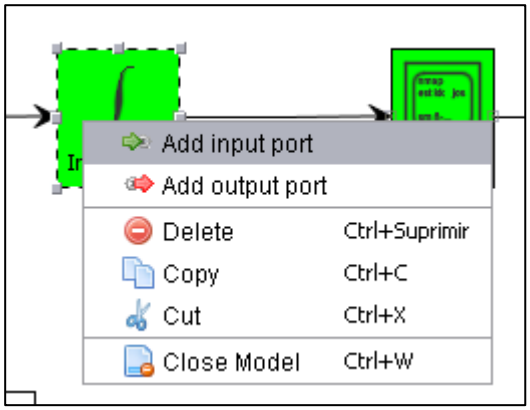

Ilustración – Detalle de menú emergente contextual

Uso de DEVS Editor

¿Cómo añado un modelo atómico?

En la librería, despliegue la carpeta *BasicElements*, haciendo doble clic sobre el icono. Pulse sobre *atomic.xml*. En ese momento aparecerá una ventana que le permitirá definir los atributos del modelo atómico.

Una vez terminada la definición de atributos pulse *Ok*. En ese momento el modelo aparecerá en el lienzo. Puede seleccionarlo y observar en el *Object Inspector* que los atributos son correctos. Puede desplazar el modelo arrastrando con el ratón. Si pulsa *Shift* mientras arrastra el ratón el modelo se moverá solamente en horizontal o vertical.

¿Cómo selecciono las celdas?

Simplemente pulse sobre la celda en cuestión, o pulse sobre el lienzo y arrastre el ratón. Todas las celdas que se encuentren dentro del marco serán seleccionadas.

¿Cómo modifico los parámetros de una celda?

Seleccione la celda que quiera modificar. Ahora busque en el *Object Inspector* los parámetros que quiera modificar, y haga doble clic sobre el valor que quiera cambiar. Una vez terminado pulse *Enter*.

Si hace doble clic sobre una celda, puede modificar la etiqueta asociada, ya sea un modelo o una conexión.

¿Cómo conecto dos modelos?

Si no puede ver los puertos del modelo, pulse el botón de la barra de herramientas *Connect*  , en ese momento aparecerán los cuadrados que representan los puertos. Para conectar dos puertos, pulse encima del puerto de salida y arrastre el ratón hasta el puerto de entrada. Cuando el ratón se ancle, suelte el botón izquierdo del ratón.

¿Cómo puedo borrar una selección?

Simplemente pulse el botón de la barra de herramientas *Remove*  , o pulse *Ctrl+Supr*. También puede pulsar con el botón derecho del ratón sobre la selección y seleccionar la opción del menú.

¿Cómo puedo agrupar un conjunto de celdas?

Lo primero que debe hacer es seleccionar las celdas a agrupar, y luego pulsar el botón *Group*  de la barra de herramientas. Si posteriormente desea desagruparlos, simplemente pulse el botón *Ungroup*  de la barra de herramientas.

¿Cómo acoplo un conjunto de celdas?

Lo primero que debe hacer es seleccionar las celdas a colapsar, y luego pulsar el botón *Collapse*  de la barra de herramientas. Si posteriormente desea expandirlos, simplemente pulse el botón *Expand*  de la barra de herramientas.

¿Puedo abrir un modelo acoplado en una nueva pestaña?

Sí, simplemente seleccione el modelo acoplado y pulse el botón *Open Model*  , que añadirá una nueva pestaña en el selector.

La nueva pestaña contendrá las celdas que forman el modelo acoplado. De hecho, son las mismas, y si modifica éstas, serán modificadas en el modelo acoplado.

He terminado el modelo ¿cómo lo guardo?

Tiene dos opciones, o bien lo guarda con un archivo XML en el directorio que desee, o bien expórtelo como elemento de la librería. Ambas opciones se encuentran en el menú *File*.

Estas dos opciones realizan la misma función, guardando un archivo XML, con la diferencia que al exportar para la librería, el documento se guardará en el subdirectorio de /*library* que usted desee, y se encontrará disponible para su posterior uso.

7. CONCLUSIONES

Hemos hablado en apartados previos de esta memoria, de cómo este proyecto está centrado en la investigación de nuevas tecnologías, y de la labor de investigación y formación que esto ha supuesto para nosotros. Así, hemos obtenido una intensa formación en campos tan cruciales y útiles como los patrones de diseño o los archivos XML. La aplicación de patrones de diseño proporciona un código más estructurado y modular, permitiendo un mejor tratamiento de errores y facilitando la tarea de futuras ampliaciones. Por otro lado, XML es el estándar para el intercambio de información estructurada entre diferentes plataformas. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil. Por estas razones fue el formato elegido en nuestro proyecto, ya que aporta la característica de ser multiplataforma, siendo además editable en formato texto.

Dentro de los conocimientos adquiridos a lo largo de la carrera, han sido de gran utilidad los aprendidos en "Ingeniería del Software" acerca de los patrones de diseño, aunque la profundidad de la aplicación de los mismos en este proyecto ha requerido una labor de formación más profunda. También en dicha asignatura se adquirieron conocimientos de los diagramas UML, que han sido aplicados en el diseño de la nuestra herramienta. En "Laboratorio de programación 3" se desarrollaron prácticas en las que aprendimos a utilizar la API de Java en profundidad, y se nos mostró como crear interfaces sencillas y amigables de cara al usuario, con funciones duplicadas en distintos elementos (menús y barras de herramientas, por ejemplo) para aumentar la sencillez de uso, conceptos que se han tenido muy en cuenta a la hora de desarrollar nuestra GUI. La API de java se ha utilizado para nuestro programa, con las nuevas funcionalidades que se han añadido en la versión 5.0. En asignaturas como "Inteligencia Artificial Aplicada al Control" o "Control Digital", se ha manejado en profundidad la herramienta de Matlab Simulink, y la estructura de nuestro editor (organización gráfica de la librería, interacción con el usuario…) se ha basado en dicha herramienta, dada la difusión que tienen actualmente. También aprendimos nociones básicas sobre la construcción de editores gráficos en distintas asignaturas de la carrera, pero no de la complejidad del

editor de este proyecto. Por último, cabe destacar que ya teníamos experiencia en la participación de proyectos grandes en asignaturas como "Ingeniería del Software" y "procesadores del lenguaje"

Otro aspecto destacable en relación con la temática del proyecto es el auge que, cada vez más, está teniendo DEVS. Actualmente está potenciado por el Departamento de Defensa de los Estados Unidos (el mayor comprador de software del mundo), y está preparado para ser estándar de IEEE. Por todo ello es destacable la proyección de futuro que puede tener una herramienta de estas características, y el amplio abanico de aplicaciones que tiene.

Como ya se ha explicado en el apartado sobre las "Vías de Evolución", destaca el posible impacto que puede tener nuestro programa sobre la disciplina, ya que no existe ninguna herramienta con la potencia y todas las características que la nuestra proporciona[12] .

Destaca la profunda modularidad que se ha empleado a lo largo del desarrollo. Internamente la estructuración de las clases posee una modularidad que permite una gran independencia entre ellas, evitando el acoplamiento. Y externamente el editor en su conjunto puede utilizarse como una pieza de un todo mayor, sin tener que modificar código.

También es destacable la importancia que están adquiriendo los generadores de código, y que va en aumento de cara al futuro. Permiten que el usuario cree sus modelos en entornos amigables, sin necesidad de escribir código. Nuestra herramienta también ofrece esta importante característica, y además lo hace en un lenguaje tan en auge como es XML.

Finalmente podemos comparar la aplicación con la herramienta Matlab-Simulink. Este último adolece del problema del tratamiento de eventos. Se ha construido una herramienta (*Stateflow*) para Simulink, pero es muy complicada de manejar. Como nuestro sistema se ha construido pensando en el tratamiento de eventos,

[12] En el apartado "Antecedentes" se compara nuestra herramienta con las que existen actualmente.

su manejo es más sencillo y natural. Simulink también ofrece la posibilidad de trabajar con XML, pero también es algo complejo. Todo ello reafirma el posible impacto que puede tener nuestro programa sobre la disciplina.

8. BIBLIOGRAFÍA

- Investigación: Bernard P. Zeigler y Hessam S. Sarjoughian "*Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models*", University of Arizona, 2003.

- Investigación: Bernard P. Zeigler, Herbert Praehofer, Tag Gon Kim, "*Theory of Modeling and Simulation*", Academic Press, 2000

- Referencia: web de Jgraph:
  www.jgraph.com

- Referencia: API de Jgraph:
  http://www.jgraph.com/pub/api/

- Referencia: API de Java 5.0:
  http://java.sun.com/j2se/1.5.0/docs/api/

- Investigación: DEVSJAVA de la Universidad de Arizona
  http://www.acims.arizona.edu/SOFTWARE/software.shtml#dj.nl

- Investigación JDEVS de la Universidad de Córcega
  http://spe.univ-corse.fr/filippiweb/theseen/index.html

- Validador XML-Schema de W3C: http://www.w3.org/2001/03/webdata/xsv#

- Investigación XML-Schema: http://www.w3.org/TR/xmlschema-1/

- Investigación Patrones de diseño: Gang Of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides), "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley Professional Computing Series, 1994.

9. GLOSARIO

En este apartado vamos a recopilar los términos y acrónimos más importantes usados en la redacción de esta memoria.

API	**A**pplication **P**rogramming **I**nterface. Es un conjunto de especificaciones de comunicación entre componentes. Proporciona abstracción de las capas inferiores del software.
Celda	Componente de JGraph que se refiere a cualquier elemento del grafo, ya sea nodo, arista o puerto.
Command	Patrón de diseño Command, que consiste en invocar una operación sin conocer realmente el contenido de esta operación.
DEVS	**D**iscrete **EV**ent **S**ystem Specification. Es un formalismo matemático empleado para modelar y simular cualquier sistema.
DEVSJAVA	Simulador creado por Bernard P. Ziegler y su equipo, en el que los modelos se programan en JAVA.
DEVS-C++	Simulador creado por Bernard P. Ziegler y su equipo, en el que los modelos se programan en C++.
GUI	**G**raphic **U**ser **I**nterface. Es la interfaz gráfica de ususario.
IDE	**I**ntegrated **D**evelopment **E**nvironment. Programa compuesto por un conjunto de herramientas para un programador. Proveen un marco de trabajo amigable para la mayoría de los lenguajes de programación.
ISCAR	**I**ngeniería de **S**istemas, **C**ontrol, **A**utomatización y **R**obótica. Grupo de investigación que proporcionó el simulador DEVS.
JGraph	Librería libre cuya función es la visualización e interacción con grafos.
Lienzo	Componente del editor donde se representan los modelos. Pueden existir varios lienzos en distintas pestañas.
MVC	Patrón de diseño Modelo-Vista-Controlador, que trata de separar la lógica y los datos de las aplicaciones de su representación visual.
Swing	Biblioteca gráfica para Java que forma parte de las Java Foundation Classes (JFC). Incluye widgets para interfaz gráfica de usuario tales como cajas de texto, botones, desplegables y tablas.
SWT	**S**tandard **W**idget **T**oolkit. Conjunto de componentes para construir interfaces gráficas en Java, desarrollados por el proyecto Eclipse. Recupera la idea original de la biblioteca AWT de utilizar componentes nativos.
XML	e**X**tensible **M**arkup **L**anguage. Lenguaje de etiquetas utilizado para el intercambio de información.
XML Schema	Un documento XML que define la estructura de otros archivos XML
SDK	Software Development Kit, for Java developing.
DTD	Document Type Definition.

UML	Unified Modeling Language, un lenfuaje de especificación y modelado de objetos utilizado en ingeniería del software.
SGML	Standar Generalized Markup Language

10. AUTORIZACIÓN

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Carlos García Arano

Samer Hassan Collado

María Teresa Murguialday Barrio