Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura en Ciencias de la Computación

Entorno Avanzado para la Simulación de Eventos Discretos

Autor: Matías Bonaventura

LU: 179/02

abonaven@dc.uba.ar

Director: Gabriel Wainer

Rodrigo Castro

Co-Directores: Ezequiel Glinsky

Año 2010

Resumen

DEVS es un formalismo general para el modelado y simulación de sistemas a eventos discretos. La creación de modelos DEVS y el análisis de los resultados de las simulaciones puede ser una tarea dificultosa, especialmente para usuarios inexpertos. Si bien se han desarrollado muchos simuladores DEVS, el software que ayuda en el ciclo de modelado y simulación requiere conocimientos de programación, y no utilizan interfaces estándar, por lo que son difíciles de extender.

El diseño y arquitectura de CD++Builder que presentamos en este trabajo simplifica la creación, análisis y simulación de modelos DEVS, facilita el reúso de modelos y promueve buenas prácticas de modelado, mediante herramientas gráficas avanzadas para la edición de modelos e integrando herramientas en un único entorno. El entorno, basado en Eclipse, brinda nuevos editores gráficos para modelos DEVS acoplados, y atómicos DEVS-Graphs y C++ (que incluyen generación automática de código que se mantienen sincronizados con su representación gráfica). La integración con Eclipse facilita la extensibilidad, el desarrollo de nuevas funcionalidades, la instalación y la actualización del software.

Abstract

DEVS is a general formalism for modeling and simulation of discrete event systems. Creating DEVS models and analyzing simulation results can be a difficult and time-consuming task, especially for non-experienced users. Although several DEVS simulators have been developed, the software that aids the modeling and simulation cycle still requires advanced development skills, and they are implemented using non-standard interfaces, which makes them difficult to extend. We present the architecture and design of CD++Builder that can simplify the construction and simulation of DEVS models, facilitate model reuse and promote good modeling practices by allowing enhanced graphical editing and integration of tools into a single environment. The Eclipse-based environment includes new graphical editors for DEVS coupled models, DEVS-Graphs and C++ atomic models (including code templates that are synchronized with the graphical versions). Integration with Eclipse allows extensibility while simplifying software development, installation and updates.

Agradecimientos

Antes que nada, a los que dieron comienzo a todo esto: mis viejos. A Roberto y María del Carmen, por bancarme, apoyarme, enseñarme, guiarme durante estos 27 años. Especialmente a mi madre, por educarnos, por darnos y dejar todo por nosotros, por la fuerza con la que criaste a tus hijos y por mucho más: gracias má!

A las personas que me acompañaron a crecer, con las que compartí la mayor parte de mi vida y me conocen mejor que nadie. Mis hermanos, Melisa y Martin, por compañeros, por complotarnos para lograr lo que queríamos, por entendernos con una mirada, por ser maravillosos y hacerme saber que siempre voy a poder contar con ustedes.

A "Lo' pibe": mis amigos y compañeros. Los increíbles individuos con los que elegí compartir mi vida. Por los pensamientos e ideas que compartimos, por la infinidad de peleas y discusiones, por los viajes, asados, partidos: Salud! En especial a los que fueron mis concubinos por muchos años: el Negro Buttara y Martin (nuevamente, pero esta vez como amigo).

A mi morochita, el amor de mi vida, por su infinita paciencia para aguantarme y soportarme en los momentos más difíciles, por su amor y pasión, por darme fuerza para seguir siempre para adelante. Por los hermosos momentos que vivimos juntos y por nuestros planes, que fueron inspiración para terminar la carrera.

A los inspiradores de este trabajo, Rodrigo, Gabriel y Ezequiel. Por el tiempo y paciencia que me tuvieron durante este tiempo, por la sabiduría y experiencia aportada.

Al Departamento de Computación, a la Facultad de Ciencias Exactas y Naturales y a la UBA en general, como organismos públicos, que me permitió y le permiten a mucha otra gente desarrollarse, crecer y aprender.

Índice

Índice de Ilustraciones

Capítulo 1 - Introducción

1.1 Motivación

Prácticamente todas las disciplinas de la ciencia y la ingeniería utilizan alguna técnica de modelado como herramienta para resolver los problemas de su interés. Los físicos modelan el movimiento de los cuerpos a través de las ecuaciones de Newton, los químicos utilizan la teoría de la química cuántica para modelar el comportamiento de átomos y moléculas, un ingeniero civil suele construir maquetas antes de construir un puente o edificio.

Salvo experimentando directamente sobre el sistema real, la simulación es la única técnica para el análisis del comportamiento de un sistema. En muchos casos la experimentación directa sobre el sistema real es imposible, muy costosa o peligrosa.

A lo largo del tiempo diferentes áreas de investigación e ingeniería han utilizado diversas técnicas de simulación según se ajustaban al problema de interés específico de cada una. Al utilizar diferentes formalismos para describir los problemas, se hacía muy difícil reusar el trabajo realizado por otro grupo y realizar simulaciones uniendo modelos de diferente naturaleza, por lo que surgió la necesidad de utilizar técnicas multiformalismo que permitan describir sistemas híbridos.

En 1976 el matemático Bernard Zeigler propuso por primera vez el formalismo conocido como DEVS (**D**iscrete **EV**ents **S**ystems specifications) [Zei76]. DEVS es un formalismo matemático general para representar Sistemas de Eventos Discretos (DES). DEVS permite no sólo modelar sistemas en forma modular y jerárquica sino que también provee una representación formal para simularlos. Estos conceptos fueron luego integrados con nociones de programación orientada a objetos para permitir computar simulaciones [Zei84] [Zei90] [ZKP00].

En los último años, el formalismo DEVS se ha vuelto muy reconocido para el modelado y simulación (M&S) de sistemas complejos, ya que permite expresar otros formalismos (Petri Nets, autómatas de estados finitos, Bond Graphs, etc) pudiendo desarrollar y luego acoplar modelos de diferentes índoles.

Se han desarrollado muchos simuladores que implementan el formalismo utilizado diversas tecnologías. En la mayoría de los simuladores DEVS, los modelos se definen utilizando algún tipo de lenguaje de programación (Java, C++, SmallTalk, etc.). Esto hace que el modelado de sistemas complejos del mundo real sea una tarea difícil para usuarios que no son desarrolladores. Para solucionar este problema, muchas herramientas simplifican el proceso de crear modelos DEVS, ejecutar las simulaciones y analizar sus resultados. Algunas de estas herramientas proveen capacidades de modelado gráfico y soporte para visualizar los resultados de las simulaciones, aunque a través de infraestructura e interfaces propias, haciendo difícil agregar nuevas funcionalidades.

1.2 Definición de la problemática

Muchas de estas herramientas poseen limitaciones en cuanto a las capacidades de modelado gráfico y requieren que los usuarios posean conocimientos de programación. En general proveen capacidades gráficas para definir modelos acoplados, pero los modelos atómicos deben ser definidos utilizando algún lenguaje de programación (Java, C++, SmallTalk, etc.). Esto hace difícil crear nuevos modelos para usuarios sin experiencia. En algunos casos donde se proporciona alguna ayuda para generar estructura de código, no brinda soporte gráfico para la definición del comportamiento de los modelos atómicos.

Por otro lado, si bien la mayoría de las herramientas permiten definir modelos DEVS acoplados gráficamente, éstas fueron desarrolladas en diferentes tecnologías (Java, C++, Pascal, Visual Basic, etc.), sin utilizar una plataforma estándar o interfaces conocidas. De esta manera, es difícil extender su funcionalidad, ya que es necesario conocer los detalles particulares de su implementación, y no pueden utilizarse dentro de un mismo entorno. Esto también genera necesidad de exportar las definiciones de los modelos de una herramienta a la otra, obligando al usuario a tener que trabajar con varias aplicaciones al mismo tiempo, dificultando el aprendizaje y aumentando los tiempos de desarrollo de los modelos. Al ser cada herramienta un desarrollo independiente, sus funcionalidades e interfaces no están integradas y es mantenerlas actualizadas resulta engorroso.

CD++ [Wai02] es una herramienta de simulación que implementa el formalismo DEVS, desarrollada originalmente en el Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires, que cuenta con herramientas asociadas que permiten la creación de modelos DEVS, su ejecución y posterior análisis de resultados. CD++ provee varios lenguajes para especificar modelos atómicos y acoplados DEVS. Los modelos acoplados pueden definirse mediante un lenguaje de alto nivel declarativo, mientras que los modelos atómicos pueden definirse tanto declarativamente utilizando DEVS-Graphs o C++.

Algunas herramientas de CD++ permiten crear modelos DEVS acoplados y atómicos DEVS-Graphs gráficamente y visualizar los resultados de una simulación, pero poseen también varias limitaciones. Por un lado, al igual que otras herramientas, son desarrollos independientes que no utilizan interfaces estándar, no poseen mecanismos para la distribución de sus actualizaciones, y son difíciles de extender. Por otro lado, están desacopladas del simulador y utilizan formatos propios para representar los modelos. Esto hace imposible visualizar gráficamente modelos CD++ ya existentes y haciendo necesario pasar de un formato al otro, siendo y difícil mantener consistente la representación gráfica con la definición ejecutable de los modelos. Si bien permiten definir modelos atómicos DEVS-Graphs, estas herramientas no brindan ningún soporte para implementar nuevos modelos atómicos utilizando C++. Esta es una gran limitación para el modelador ya que no todos los modelos atómicos pueden ser definidos mediante DEVS-Graphs, restringiendo la complejidad de los modelos que pueden generarse gráficamente y permitiendo crear solo modelos relativamente simples. Más allá de las limitaciones anteriores, en el uso de las herramientas se identificaron algunas falencias de usabilidad general, como falta de comandos

habituales, acciones poco intuitivas, dificultad para navegar los modelos acoplados jerárquicamente, etc.

1.3 Objetivos

El objetivo del presente trabajo es crear un entorno de desarrollo integrado (IDE, Integrated Development Environment) para el modelado y simulación DEVS, que permita y asista al usuario realizar todas las tareas (creación y edición de los modelos, compilación, ejecución y análisis de resultados). El principal fin es construir una única herramienta que permita a científicos, ingenieros y estudiantes, utilizar DEVS como instrumento de simulación para construir modelos de forma rápida, sencilla e intuitiva.

Debe estar orientado a usuarios con y sin conocimientos de programación; permitiendo crear y editar modelos DEVS de forma gráfica sin necesidad de contar con conocimientos de programación, respetando el formalismo original propuesto por Zeigler [Zei/90]. También se debe dar soporte y facilitar la tarea aquellos modeladores que deseen crear modelos más complejos utilizando C++ como lenguaje de programación.

Continuamente se actualizan y desarrollan nuevas herramientas, por lo que es importante que la plataforma provea una arquitectura extensible. Se debe poder agregar y desarrollar nuevas funcionalidades, cada una de forma totalmente independiente de las otras. La distribución, instalación y actualización debe sencilla y clara para los usuarios, permitiendo mantener el software actualizado de forma automática.

1.4 Contribuciones

En este trabajo presentamos las nuevas características de CD++Builder 2.0, un plugin para Eclipse que se centra en resolver los problemas anteriores, basado en el simulador CD++. La integración de varias herramientas asociadas a CD++ en un mismo entorno reduce la curva de aprendizaje a nuevos usuarios y estudiantes, y simplifica el proceso de M&S evitando la necesidad de utilizar varias aplicaciones.

Los nuevos editores gráficos de CD++Builder para modelos acoplados y modelos atómicos DEVS-Graphs proveen todas las capacidades de modelado gráfico de las herramientas anteriores, integrados dentro del entorno de Eclipse. Los nuevos editores resuelven los problemas de usabilidad a través del uso de los asistentes estándar de Eclipse, comandos de copiar y pegar, teclas de acceso rápido, zoom in/out de la vista del modelo, etc. También resuelven las limitaciones de modelado, permitiendo utilizar no solo modelos DEVS-Graphs sino también modelos atómicos definidos en C++ para componer los modelos acoplados. . Si bien estos modelos atómicos utilizan C++ para definir su comportamiento, los editores proveen una representación gráfica de la interfaz (nombre del modelo, puertos y parámetros) de los mismos.

Al eliminar la necesidad de exportar los modelos, la representación gráfica y la utilizada por CD++ para ejecutar las simulaciones se mantienen consistentes automáticamente, permitiendo a los usuarios actualizar ambas vistas. Con las nuevas características desarrolladas para los editores, es

posible abrir definiciones de modelos realizadas con versiones anteriores de la herramienta, para las cuales se generará una representación gráfica del modelo. Esto es muy beneficioso para los modeladores, ya que CD++ cuenta con una vasta librería de modelos ya implementados y testeados, pero estos no contaban con una representación gráfica. Al poseer una vista gráfica que estos modelos, es más sencillo comprender su estructura y funcionamiento.

La posibilidad de crear y editar modelos atómicos y acoplados gráficamente permite a los usuarios especificar modelos DEVS completos sin programar. Sin embargo, En los casos en donde sea necesario desarrollar modelos más complejos en C++, se proveen templates de código para evitar tareas repetitivas y propensas a errores. Los templates poseen estructuras de código estándar y comentarios que promueven buenas prácticas de programación y modelado. La integración con el plugin para Eclipse *C/C++ Development Tools (CDT)* [EFCDT10] provee asistencia para el desarrollo en C++ (coloreo de código, resumen de métodos, etc.).

Para el análisis de los resultados de las simulaciones, se reutilizaron los editores de las herramientas anteriores [CW06], que muestran animaciones de los envíos de mensajes entre modelos y los valores de los puertos de entrada y salida. Las animaciones pueden ser ejecutadas desde la interfaz de Eclipse, generando las vistas a partir de las nuevas representaciones gráficas. Al utilizar las nuevas representaciones gráficas, es posible visualizar los resultados de modelos acoplados compuestos por modelos atómicos C++ que antes no era posible.

CD++Builder 2.0 fue desarrollado en la plataforma de Eclipse. Dado que Eclipse es multiplataforma proporciona un entorno que permite desarrollar modelos DEVS tanto en sistemas Linux como Windows. Por otro lado, es una plataforma estándar para desarrollo de IDEs conocida mundialmente, por lo que provee de una interfaz ya conocida para los usuarios y fácil de comprender. Eclipse provee un framework desarrollado para ser extensible por lo que se podrá en el futuro añadir nuevas herramientas para trabajar con modelos DEVS integradas con las ya existentes dentro de la misma plataforma. Eclipse también puede ser extendido fácilmente para proveer un sitio de actualizaciones, que permite instalar y descargar actualizaciones para los plugins de un único punto accesible a todos los usuarios, facilitando la distribución de nuevas herramientas y soluciones a bugs encontrados. Por otro lado el desarrollo de CD++Builder 2.0 incluye tests automáticos de regresión, que facilitan la mantención y extensibilidad del plugin, ya que pueden ser utilizados para comprobar el correcto funcionamiento de las funcionalidades ya desarrolladas luego de introducir nuevo código.

1.5 Resultados

CD++Builder 2.0 provee una plataforma e interfaz única para llevar a cabo todas las tareas en el proceso de M&S utilizando CD++. Los editores gráficos permiten crear y editar modelos DEVS de forma gráfica, a través de los lenguajes de alto nivel descriptivos de CD++ o utilizando C++. El entorno permite realizar la compilación de los modelos atómicos C++ y ejecutar las simulaciones a través de los botones de acción (desarrollados en la versión anterior de CD++Builder [CW07]). Las animaciones para los resultados de las simulaciones permiten comprender y visualizar el flujo de la ejecución de forma más intuitiva.

Eclipse provee la infraestructura necesaria que permite agregar nuevas funcionalidades de forma totalmente desacoplada, en forma de plugins. Esta infraestructura también provee mecanismos de actualización e instalación para plugins, simplificando y automatizando distribución de nuevas versiones de CD++Builder y CD++.

Capítulo 2 - Background

La simulación es una metodología para estudiar sistemas complejos. El proceso de simulación comienza siempre con un problema que se quiere resolver o comprender. Por ejemplo un químico puede tratar de comprender las complejas reacciones que se dan dentro de un tubo de ensayos o un ingeniero civil construir un edificio para que pueda ser evacuado fácilmente. En muchos casos el proceso de simulación comienza por la observación de un **sistema real** a partir del cual se identifican las entidades y se construye un **modelo** del mismo. Una vez que el modelo esta correctamente definido se experimenta sobre el mismo mediante un **simulador** que ejecuta el modelo. Finalmente los resultados obtenidos de la simulación son comparados con los del sistema real para validar la correctitud del modelo. En la mayoría de los casos es imposible crear un modelo que represente todos los aspectos del sistema real. Es por eso que se utiliza un **marco experimental** que define los objetivos del modelador y el alcance del modelo. La Figura 1 - Relaciones de modelado y simulación.Figura 1 muestra la relación entre estos conceptos:

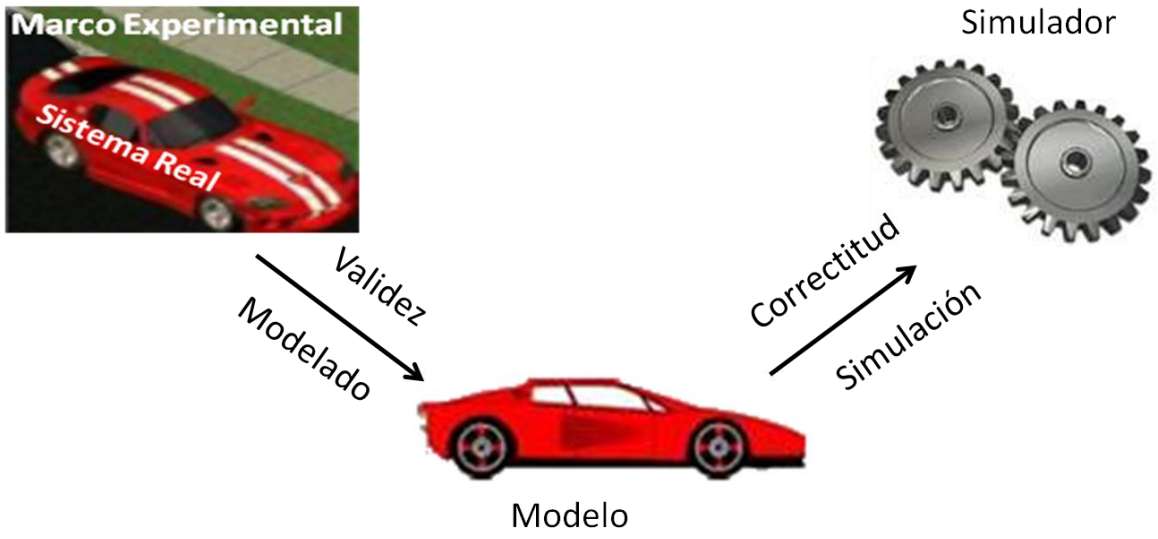


Figura 1 - Relaciones de modelado y simulación.

Formalmente podemos definir un **sistema** como una entidad real o artificial que representa una parte de una realidad y está restringida por un entorno. Puede decirse que un sistema es un conjunto ordenado de objetos lógicamente relacionados que atraviesan actividades, interactuando para cumplir los objetivos propuestos [Wai03]. Por otro lado, un **modelo** es una representación inteligible (abstracta y consistente) de un sistema.

Es muy común en el ámbito científico realizar experimentos directamente sobre el sistema real para estudiarlo. Por ejemplo cuando un químico mezcla diferentes sustancias y observa los resultados. Sin embargo en muchos casos no se puede experimentar sobre el sistema a estudiar, o se desea evitar costos, peligro, etc. [Wai03]. Se pueden distinguir dos grandes grupos de métodos

para modelar sistemas complejos a partir de un sistema real: los modelos analíticos y los modelos basados en simulación.

Los **modelos analíticos** están basados en el razonamiento deductivo y permiten obtener soluciones generales al problema. Un formalismo analítico muy difundido para el modelado de problemas es el uso de ecuaciones diferenciales. Un problema de estos métodos reside en que, al considerar sistemas complejos, estos (con pocas excepciones) serán analíticamente intratables y numéricamente prohibitivos de evaluar. Por ende, para poder utilizar estos métodos en problemas que existen en el mundo real, se debe simplificar el modelo a un nivel tal en que las soluciones obtenidas se alejan de la realidad. [Wai03]

En los casos en que se hace imposible modelar el sistema mediante métodos analíticos, suelen utilizarse **modelos basados en simulación**. En este tipo de modelos no se intenta buscar una solución general al problema, sino que se buscan soluciones particulares lo que permite tratar cierta complejidad. Algunos problemas con estos modelos son sus tiempos de desarrollo, la necesidad de validación de los resultados y la necesidad de una colección extensiva de datos para reproducir el comportamiento del sistema.

Algunas ventajas de realizar simulaciones frente a la experimentación directa sobre el sistema son [Wai03]:

- **Repetitividad**: Una simulación puede ser realizada tantas veces como sea necesario.
- **Experimentación Controlada**: La simulación de un problema no afecta al modelo real.
- **Compresión del tiempo**: Una simulación permite, en algunos casos, ser llevada a cabo en menos tiempo que lo que duraría el experimento en un sistema real.
- **Análisis de sensibilidad:** el modelador decide cual es el marco experimental (leyes que rigen la simulación)
- **Automatización:** se puede automatizar la simulación para encontrar los resultados deseados.

También pueden clasificarse los modelos según su base de tiempo y el valor de sus variables como muestra la Figura 2. Con respecto a la base de tiempo, hay paradigmas a **tiempo continuo** donde el tiempo evoluciona de forma continua, y a **tiempo discreto** donde el tiempo avanza por saltos de un valor entero a otro. Con respecto a los conjuntos de valores de las variables descriptivas del modelo, hay paradigmas de estados o **eventos discretos** (las variables toman sus valores en un conjunto discreto), **continuos** (las variables son números reales), y **mixtos** (ambas posibilidades) [GG96].

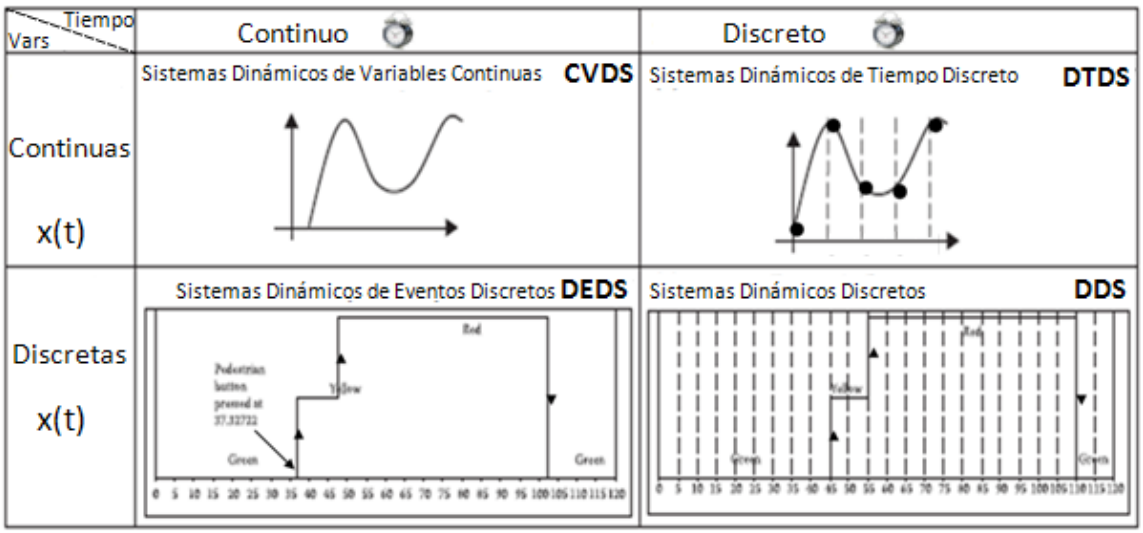


Figura 2 - Clasificación de modelos según su base de tiempo y el valor de sus variables [Wai09].

Por ejemplo, los modelos descriptos con ecuaciones diferenciales pueden encontrarse tanto dentro del primer cuadrante como dentro del segundo (variables continuas a tiempo discreto / continuo). Dentro del tercer cuadrante (variables y tiempo discretos) se encuentran los modelos descriptos con máquinas de estados finitos, Petri Nets, etc. Finalmente se encuentran en el cuarto cuadrante los paradigmas de variables discretas a tiempo continuo. Tales sistemas reciben el nombre de **Sistemas Dinámicos de Eventos Discretos** (**DEDS** – Discrete Events Dynamic Systems). Un ejemplo de esto último es el paradigma en el que se basa este trabajo conocido como DEVS.

2.1 Formalismo DEVS

En 1976 el matemático Bernard Zeigler [ZKP00] propuso un formalismo conocido como **DEVS** (*Discrete EVents Systems specifications*). DEVS es un formalismo universal para modelar y simular DEDS.

El formalismo DEVS provee una forma de especificar sistemas cuyas entradas, estados y salidas son constantes en intervalos, y cuyas transiciones se identifican como eventos discretos. El formalismo define como generar nuevos valores para las variables y los momentos en los que estos valores deben cambiar. Los intervalos de tiempo entre ocurrencias son variables, lo que trae algunas ventajas con respecto a otros formalismos [Wai03].

DEVS hace una clara separación de los conceptos de modelado, simulación y ejecución. El formalismo es independiente de todo mecanismo de simulación. Por un lado permite describir los modelos en forma **modular,** lo que facilita la reutilización de funcionalidad en distintos modelos reduciendo los tiempos de desarrollo y prueba. Por otro lado, DEVS ataca la complejidad de los modelos permitiendo un modelado en forma **jerárquica.** Además de un medio de construir modelos simulables, provee una representación formal para manipular matemáticamente sistemas de eventos discretos. Para la ejecución de las simulaciones, esta aproximación se integró posteriormente con nociones de programación orientada a objetos [Zei84] [Zei90] [ZKP00].

En cuanto a la construcción de modelos DEVS podemos encontrar dos tipos de modelos: los **modelos atómicos** (o de comportamiento) y los **modelos acoplados** (de estructura). Los modelos acoplados se construyen en base a un conjunto de modelos atómicos y otros modelos acoplados, lo que da forma a una estructura de modelos jerárquica y modular. Cada modelo atómico define el comportamiento básico de una parte del sistema, mientras que los modelos acoplados especifican cómo se conectan las entradas y salidas de sus componentes para formar partes de mayor nivel.

A continuación veremos la definición formal de los diferentes modelos DEVS.

2.1.1 Definición formal: Modelo Atómico

Un Modelo Atómico DEVS se representa formalmente mediante una estructura:

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

Donde:

X	Es el conjunto de **eventos** externos **de entrada**;
Y	Es el conjunto de **eventos** externos generados por la **salida**;
S	Es el **conjunto de estados** secuenciales;
$\delta_{ext}: Q \times X \rightarrow S$	Es la **función de transición externa**. Define los cambios de estados por causa de eventos externos.
donde $Q = \{ (s,e) / s \in S, e \in [0, ta(s)] \}$ y e es el tiempo transcurrido desde la última transición de estado s.	
$\delta_{int}: S \rightarrow S$	Es la **función de transición interna**. Define los cambios de estados por causa de eventos internos.
$\lambda: S \rightarrow Y$	Es la **función de salida**.
$ta: S \rightarrow R_0^+ \cup \infty$	Es la **función de duración de un estado**. Ta(s) es el tiempo que el modelo se queda en el estado s si no hay un evento externo.

Para especificar modelos DEVS, es conveniente considerar que el modelo tiene una interfaz consistente de puertos de entrada y salida que interactúan con el entorno. Esta interfaz de puertos puede definirse como:

$$I = < P^X, P^Y >$$

Dónde:

$\forall i \in \{X, Y\}$, P_j^i *es una definición de un puerto (de entrada o salida respectivamente), donde* $j \in N, j \in [1, \mu], (\mu \epsilon N, \mu < \infty)$ *y*

$$P_j^i = \{(N_j^i, T_j^i) \setminus N_j^i \in [i_1, i_\mu](Nombre\ del\ puerto), y\ T_j^i = Tipo\ del\ puerto\}$$

Estos elementos son interpretados de la siguiente manera:

Un modelo atómico DEVS siempre se encuentra en un estado s ϵ S y cambia su estado sólo a través de las funciones de transición (δ_{int} y δ_{ext}). Si no ocurre ningún evento externo, el modelo

permanece en el estado s por el tiempo indicado por la función **ta(s)**. Una vez transcurrido el tiempo **ta(s)**, el modelo primero genera un evento de salida ejecutando la función de salida ($\lambda(s)$), y luego cambia su estado al estado indicado por δ_{int}.

En el caso de que ocurra un evento externo x ϵ X , el modelo cambia su estado al estado indicado por la función de transición externa $\delta_{ext}(s, e, x)$, donde s indica el estado actual del modelo, e indica el tiempo transcurrido desde la última transición (notar que siempre se cumple $e \leq ta(s)$) .

En la figura siguiente puede verse el comportamiento de un modelo atómico DEVS:

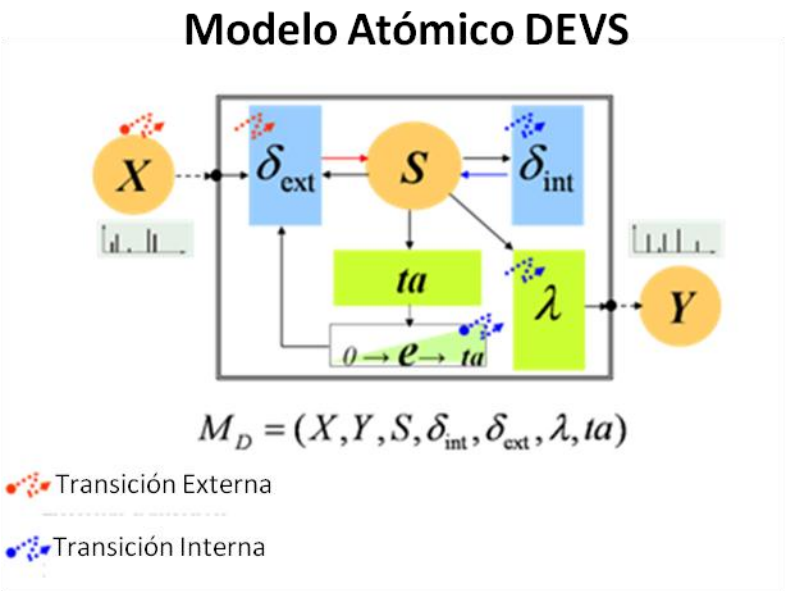


Figura 3 - Comportamiento de un modelo atómico DEVS.

2.1.2 Definición formal: Modelo Acoplado

Los modelos DEVS pueden componerse con otros modelos DEVS para formar un nuevo modelo más complejo que llamamos modelo acoplado. Una propiedad muy importante de los modelos DEVS es la de clausura bajo acoplamiento, que garantiza que dada la definición de un modelo acoplado existe un modelo atómico equivalente. Es por esto que al ser los modelos acoplados un modelo DEVS, pueden también formar parte de otros modelos acoplados. A los modelos que componen a un modelo acoplado los llamaremos submodelos.

Un modelo DEVS acoplado se define como:

$$CM = <X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

Donde:

X_{self}	Es el conjunto de **eventos** externos de **entrada.**
Y_{self}	Es el conjunto de **eventos** externos de **salida.**
D . $D \in N < \infty$.	Es el conjunto de índices de modelos **componentes**.

M_i	Es un **modelo componente básico** correspondiente al componente i (modelo DEVS atómico o acoplado), definido como: $M = < X_i, Y_i, S_i, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i >. i \in D.$
I_i	Es el conjunto de modelos **influenciados** por el modelo i (modelos que pueden ser influenciados por salidas del modelo i).
$Z_{i,j}$: Yi → Xj	es la **función de traducción** de la influencia i en j.
Select	Es la **función de prioridad** o secuencialización.

2.1.3 Definición formal: Modelo Atómico DEVS-Graphs

Una extensión a la definición original de modelos atómicos DEVS define una gramática que permite una notación gráfica de los mismos [DCW03]. Esta notación permite definir el comportamiento de los modelos atómicos basándose en DEVS-Graphs [PP93] como muestra la siguiente figura:

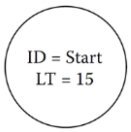
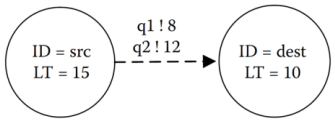
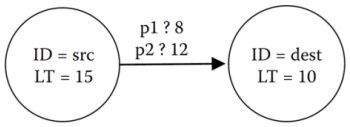
Definición	Estados	Transiciones Internas	Transiciones Externas
Gráfica			
Textual	state: stateId stateId: lifetime	int: src dest {q ! v}* {(action;)*}	ext: src dest exp ([p?v]*) {(action;)*}

Figura 4 - Notación gráfica DEVS-Graphs.

Los modelos DEVS-Graphs se basan en grafos para definir los estados y sus transiciones. Cada DEVS-Graphs define los cambios de estado a partir de transiciones internas y externas, y cada una es traducida a una definición analítica [Wai09]. Los vértices representan los estados del modelo y las aristas representan transiciones externas e internas. Los estados (nodos del grafo) especifican su tiempo de vida (función **ta(s)** de los modelos tradicionales) permitiendo formar tuplas (estado, duración) asociadas a las transiciones internas. De esta manera cuando tiempo de vida de un estado se consume, el modelo cambia su estado ejecutando una función de transición interna.

Un modelo atómico DEVS-Graphs puede definirse formalmente como:

$$DEVS\text{-}Graphs = < X_M, S, Y_M, \delta_{int}, \delta_{ext}, \lambda, D >$$

Donde:

$Xm = \{(p,v) \mid p \in IPorts, v \in Xp\}$	Es el conjunto de **puertos de entrada**;
$Ym = \{(p,v) \mid p \in OPorts, v \in Yp\}$	Es el conjunto de **puertos de salida**;
$S = BxP(V)$	Son los estados del modelo
$B = \{b \mid b \in Bubbles\}$	Es un conjunto de **estados** del modelo
$V = \{(v,n) \mid v \in Variables, n \in \Re_0\}$	Son **variables** de estado intermedias del modelo y sus valores
$\delta_{int}, \delta_{ext}, \lambda, D$	Tienen el mismo significado que en los modelos

atómicos DEVS tradicionales

La herramienta CD++, que se explicará más adelante, define una representación textual equivalente que permite la ejecución y simulación de estos modelos.

2.1.4 Definición formal: Modelo Celular

Una extensión al formalismo DEVS original permite definir autómatas celulares [Wol86]. Los modelos celulares suelen representar espaciales en las que se organiza el espacio como una red de células distribuidas geométricamente. Wainer y Giambiasi presentaron el formalismo Cell-DEVS [Wai09], una extensión a la teoría de autómatas celulares que utiliza DEVS para representar cada celda. Este formalismo permite definir comportamientos celulares complejos con instrucciones simples y construir espacios celulares n-dimensionales para representar modelos a eventos discretos complejos.

Un modelo atómico Cell-DEVS puede ser definido formalmente como:

$$TDC = < X, Y, S, N, type, d, \tau, \delta_{int}, \delta_{ext}, \lambda, D >$$

Donde:

X	Es el conjunto de **eventos** externos de **entrada.**
Y	Es el conjunto de **eventos** externos de **salida.**
S	Es el conjunto **estados**.
$N \mathcal{E} X^n$	Es el conjunto de **valores de entrada**
$d\ \mathcal{E} \Re_0^+$	Es la demora de la celda
Type	Es el **tipo de demora** (de transporte/inercial/otro)
$r: N \to S$	Es la **función de computo local**
$\delta_{int}: S \to S$	Es la **función de transición interna**
$\delta_{ext}: Q \times X \to S$	Es la **función de transición externa**
$\lambda: S \to Y$	Es la **función de salida**
$ta: S \to \Re_0^+ \cup \infty$	Es la **función de duración de un estado.**

Una vez definido el comportamiento de una celda, necesitamos formar un espacio de celdas. A continuación se muestra la definición de espacios Cell-DEVS de dos dimensiones con vecindades adyacentes. Se puede encontrar la definición de espacios n-dimensionales con vecindades genéricas en [Wai98].

Un modelo acoplado Cell-DEVS puede definirse formalmente como:

$$GCTD = < X, Y, Xlist, Ylist, \eta, N, \{m, n\}, C, B, Z, select >$$

Donde:

X	Es el conjunto de **eventos** externos de **entrada.**
Y	Es el conjunto de **eventos** externos de **salida.**
$Ylist = \{(k,l)\|k \in [0,m], l \in [0,n]\}$	Es la lista de **acoplamiento de salida.**
$Xlist = \{(k,l)\|k \in [0,m], l \in [0,n]\}$	Es la lista de **acoplamiento de entrada.**
Select	Es la **función de prioridad** o secuencialización.
$n \in N$	Es el **tamaño de la vecindad.**
$N = \{(i_p,j_p)\|\forall p \in N, p \in [1,n] \Rightarrow i_p,j_p \in Z \wedge i_p,j_p \in [-1,1]\}$	Es el **conjunto de vecindad.**
{m,n}	Es el **tamaño del espacio celular**
$C = \{C_{ij}\|i \in [1,m], j \in [1,n]\}$	Es el **conjunto del espacio celular**. C_{ij} son modelos atómicos Cell-DEVS.
B	Es el conjunto de **celdas borde.**
Z	Es la función de traducción

2.2 Herramientas De Simulación Existentes

Como se vio en la sección "2.1 Formalismo DEVS", el formalismo DEVS es independiente de todo mecanismo de simulación, por lo que han surgido a lo largo del tiempo muchos desarrollos que implementaron el formalismo. Se han utilizado diferentes tecnologías, cada herramienta provee funcionalidades diferentes con sus pros y sus contras. La gran mayoría de estos simuladores cuentan con herramientas gráficas que permiten crear modelos y visualizar los resultados de las simulaciones.

Si bien este trabajo está centrado en la implementación desarrollada en el Departamento de Computación de la UBA conocida como CD++ (que se detallará en la sección "2.3 Herramienta CD++"), en esta sección se describirán otras implementaciones del formalismo DEVS, las diferentes interfaces gráficas utilizadas y se discutirán sus ventajas y desventajas.

Existen muchos desarrollos que permiten el modelado y la simulación de modelos utilizando el formalismo DEVS. Algunas de estas implementaciones se encuentran listadas en http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm. Dentro de las más destacadas se encuentras las siguientes:

ADEVS [Nut] **(A D**iscrete **EV**ent **S**ystem simulator) es una librería en C++ para realizar simulaciones basadas en Parallel DEVS y Dynamic DEVS. Posee un motor de simulación performante [GW08], que permite crear modelos de red y de estructura variable, pero no posee un lenguaje de alto nivel para describirlos ni una interfaz gráfica que facilite la visualización de los resultados de la simulación.

CoSMos [SE09] (Component-based System Modeler and Simulator) integra las interfaces de CoSMo (**Co**mponent-based **S**ystem **Mo**deler) y DEVS-Suite permitiendo el desarrollo de familias de modelos integrando, generación de código y el almacenamiento de modelos en bases de datos. Permite generar la estructura de código necesaria para definir el comportamiento de modelos atómicos en DEVS-Suite a partir de la descripción gráfica.

CoSMoS también provee un editor de código (Java) para los modelos atómicos basado en el editor de NetBeans con las funcionalidad de coloreo de código, reconocimiento de las palabras clave, etc.

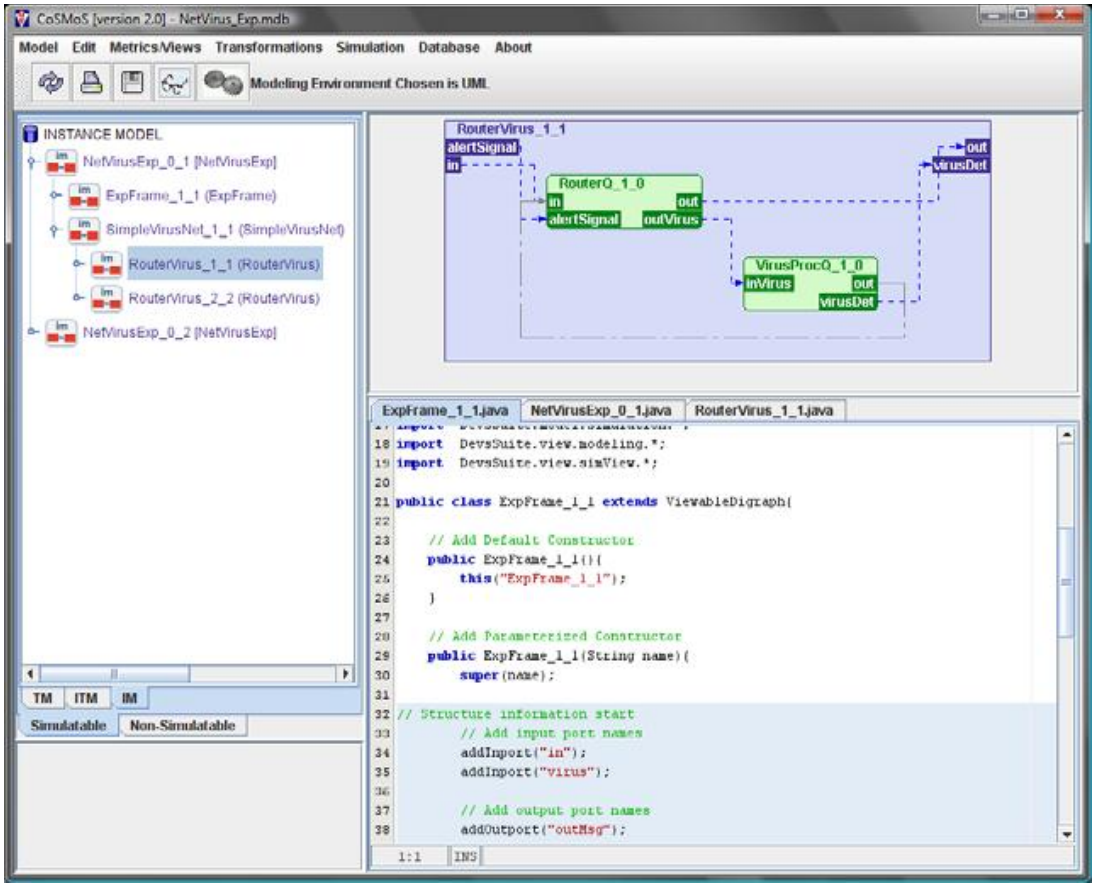


Figura 5 - Interfaz de usuario de CoSMoS con el editor para agregar comportamiento a los modelos [SE09].

Como se puede ver en la Figura 5, para poder mantener consistentes la definición gráfica y de comportamiento de los modelos atómicos, el editor de código Java tiene secciones de código bloqueadas que no pueden editarse.

DEVSJAVA [SZ98][ZS03] es una de las herramientas más conocidas para la creación y simulación de modelos DEVS, desarrollada por Hessam Sarjoughian y Bernard Zeigler en la universidad de Arizona. Es un desarrollo realizado en Java que provee 4 módulos para separar el modelado y simulación (**genDEVS.modeling** posee las entidades de modelado y **genDEVS.simulation** implementa el simulador abstracto) de la interfaz de usuario (SimView). DEVSJAVA permite la ejecución paralela de simulaciones y la visualización de modelos jerárquicos.

SimView es el módulo de visualización de DEVSJAVA. SimView muestra los modelos atómicos mediante una caja rectangular y sus puertos de entrada/salida lindantes mediante pequeños círculos. En el interior de estos rectángulos se muestra dinámicamente la información relevante de los modelos atómicos (por ejemplo el *elapsed time*). Los modelos acoplados están representados por el perímetro de un rectángulo y en su interior los modelos que este contiene junto con las

interacciones. Durante la ejecución de la simulación pueden verse los mensajes que se envían entre modelos acoplados. La interfaz gráfica de SimView puede verse en la siguiente figura:

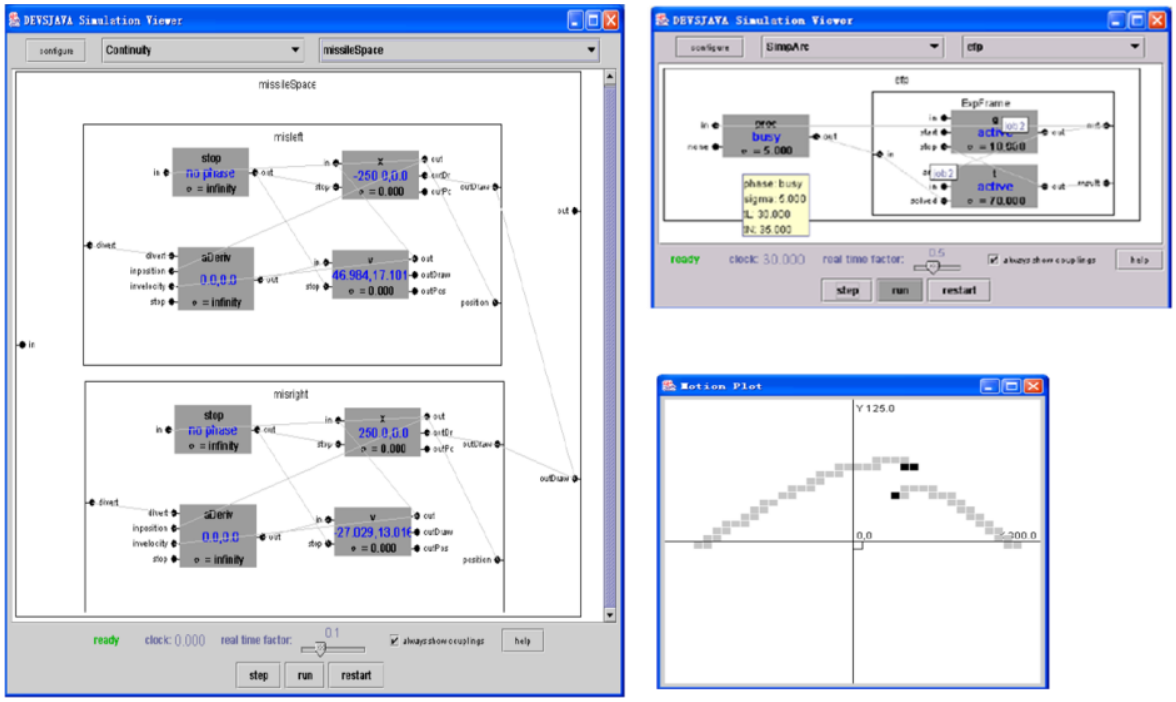


Figura 6 - Interfaz gráfica de DEVSJAVA SimView [ZS03].

Si bien la visualización de modelos acoplados permite distinguir rápidamente su composición es sencilla solo para modelos de poca profundidad. Al tener muchos modelos acoplados dentro de otros modelos acoplados se dificulta su visualización. Por otro lado los puertos de entrada son distinguibles de los puertos de salida solo por su posición respecto del modelo.

En DEVSJava el módulo de visualización está acoplado con el simulador por lo que desde la interfaz gráfica se puede iniciar, pausar y reanudar la simulación fácilmente. El diseño visual de los componentes debe realizarse por código (manualmente o mediante una herramienta de generación de código). Esto último, por un lado hace necesaria la recopilación para modificar la visualización del modelo. Por otro lado provoca que el código de la lógica del modelo y el código que especifica su visualización queden juntos, complicando el modelado.

DEVS-Suite [SSE09] extiende el entorno provisto por DEVSJAVA combinándolo con DEVS Tracking Environment [SS04] para soportar el diseño de experimentos en forma gráfica y la visualización de información recolectada de la simulación dinámicamente. La interfaz de usuario de DEVS-Suite (Figura 7) consiste de cuatro partes: (1) Visualizador del modelo en la esquina superior izquierda, (2) Control del simulador en la esquina inferior izquierda, (3) SimView en la esquina superior derecha y (4) TimeView en la esquina inferior derecha. La interfaz también permite adaptar el tamaño y ocultar de estos paneles. TimeView permite mostrar dinámicamente el valor generado por puertos de entrada o salida y de las variables que se rastrean.

Figura 7 - Interfaz de usuario de DEVS-Suite, con SimView y TimeView [KSE09].

La posibilidad de tener varios paneles ayuda al modelador a tener distintas visiones de un mismo modelo y tener accesibles las herramientas más utilizadas. Si bien en DEVS-Suite se puede adaptar el tamaño de los paneles su personalización es limitada. En CD++Builder se utilizaron de las funcionalidades provistas por Eclipse para el manejo de vistas que permite modificar su posición y tamaño, colapsarlos (docking), personalizar que vistas son visibles, etc.

JDEVS [FDB02] es una implementación en Java del formalismo DEVS desarrollada por Jean-Baptiste Filippi en la Universidad de Corsica, Francia. JDEVS está compuesto por cuatro módulos: el motor de simulación, una interfaz gráfica de usuario y dos componentes para visualizar simulaciones en 2D y 3D.

Más allá de su módulo de simulación provee un modelo de interfaz de usuario, un módulo de visualización 2D y otro de visualización 3D. La interfaz de usuario permite crear modelos, describir sus interacciones y provee una librería de modelos. Los modelos son guardados en formato XML permitiendo la colaboración entre modeladores. La interfaz gráfica de usuario de JDEVS puede verse en la Figura 8 a).

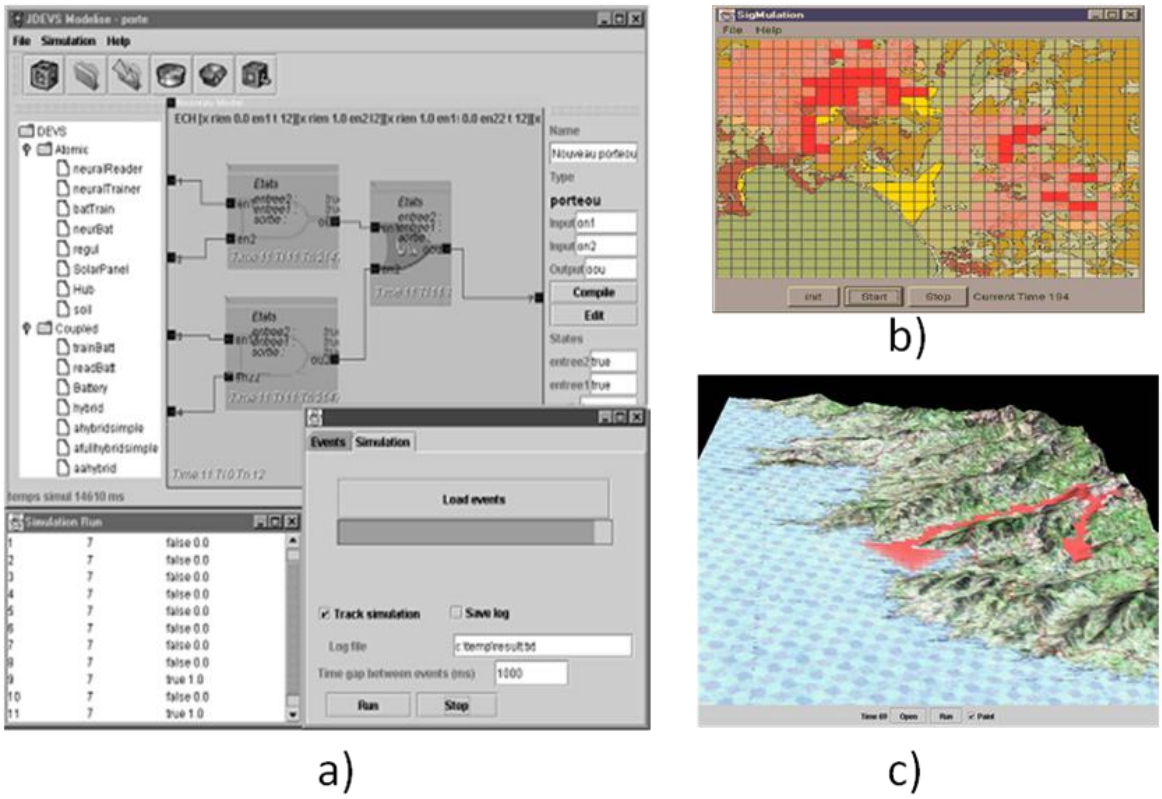


Figura 8 - Entorno JDEVS a) Interfaz gráfica de usuario. B) Visualización 2D. c) Visualización 3D. [FDB02]

En la Figura 8 a) y b) puede verse la ejecución de una simulación utilizando los módulos 2D y 3D respectivamente. Estos módulos de visualización están especialmente desarrollados para la visualización de sistemas naturales y permiten exportar datos de sistemas GIS dando una vista atractiva a la simulación.

Si bien JDEVS permite crear y simular modelos DEVS de propósito general, los módulos de visualización son sólo aptos para modelos naturales. Por otro lado la interfaz gráfica no permite la creación de modelos jerárquicos (unicamente permite la visualizar un modelo a la vez).

PowerDEVS [PLK03] es un desarrollo realizado en la Universidad Nacional de Rosario, Argentina, por el Dr. Ernesto Koffman. PowerDEVS es un conjunto de herramientas de software que corren en el entorno Windows. Estas permiten editar modelos DEVS especificando el acoplamiento de manera gráfica y las funciones de los modelos atómicos en lenguaje C++. PowerDEVS fue concebido como una herramienta computacionalmente eficiente y a la vez simple de utilizar en áreas de ingeniería. Es por esto que el motor de simulación fue desarrollado en C++ mientras que posee una interfaz gráfica implementada en Visual Basic basada en entornos como Simulink. PowerDEVS permite también la simulación en tiempo real con capturas de interrupciones externas.

El entorno de PowerDEVS permite editar modelos DEVS especificando el acoplamiento de manera gráfica (Figura 9 b). Cuenta además con una librería que permite arrastrar los modelos a la

ventana de edición para su reutilización (Figura 9 a). Los modelos atómicos pueden configurarse a través de parámetros utilizando el editor, evitando el desarrollo de nuevos modelos similares. El editor de modelos atómicos es una aplicación independiente y permite especificar el código C++ que se utilizara en cada una de las funciones (inicialización, transición interna, transición externa, de salida y avance de tiempo) (Figura 9 c).

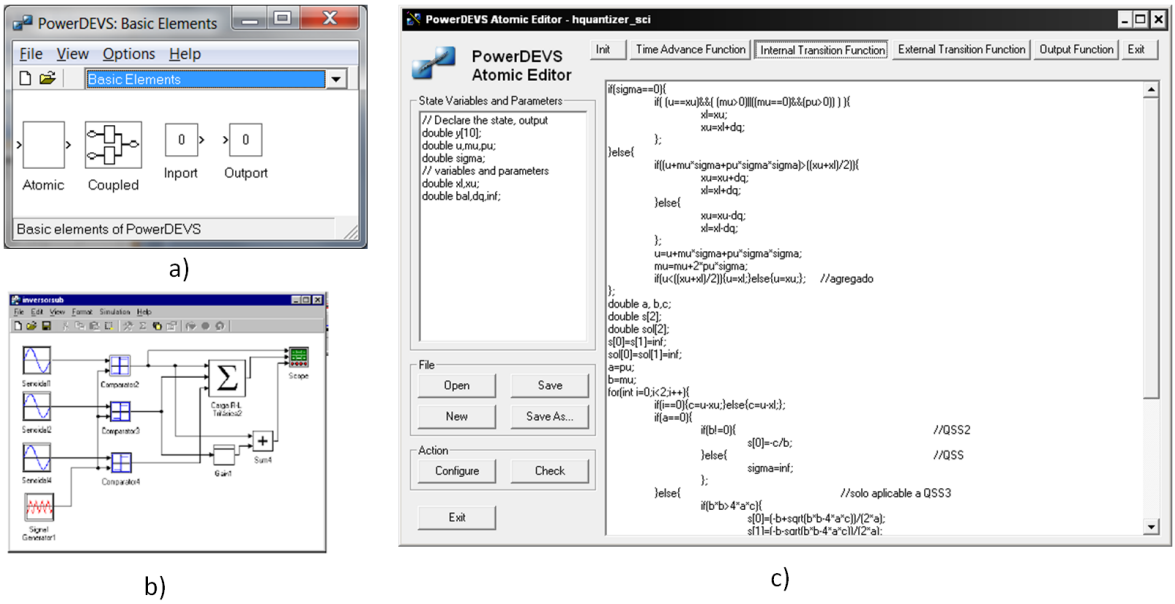


Figura 9 - Entorno PowerDEVS a) Librería de modelos. B) editor de acoplados. c) editor de atómicos. [PLK03]

Dentro de la librería de modelos que provee PowerDEVS se encuentra una serie de modelos atómicos que permiten capturar valores de la simulación e interactuar con diferentes dispositivos del sistema. Por ejemplo provee un modelo que guarda a disco en formato CSV los valores de sus puertos, un modelo que escribe por el puerto de impresión LPT y otro que utiliza librerías estándar de GNU para graficar los valores de los puertos (Figura 10). Estos visualizadores son de suma utilidad para el modelador para comprender la dinámica ya que permiten visualizar diferentes valores a medida que la simulación avanza.

Si bien estos modelos que interactúan con dispositivos del sistema son muy útiles durante el desarrollo del modelo, deben ser incluidos como parte del modelo ya que son la única herramienta para extraer datos de la simulación. Esto provoca que el modelo conceptual quede fusionado con las herramientas de seguimiento de la simulación.

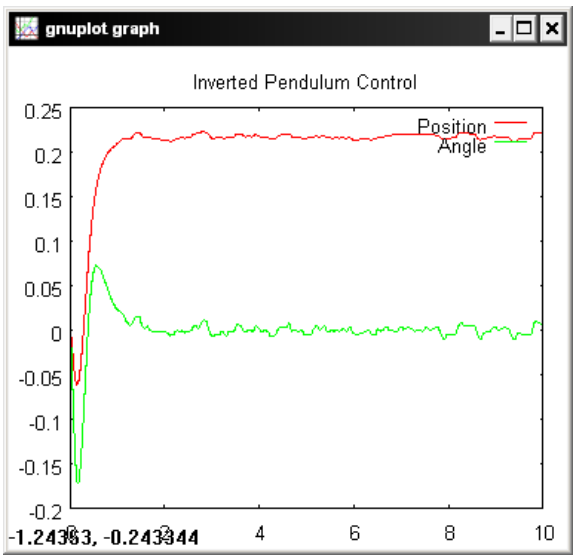


Figura 10 - Gráfico GNU Plot de PowerDEVS.

VLE [QDRT07] es un entorno de M&S, implementado en C++, orientado a integrar formalismos heterogéneos encapsulando todos los modelos dentro de un modelo DEVS para permitir la interoperabilidad. Provee módulos separados para la interfaz de usuario, para visualizar los resultados de la simulación y para la implementación de los algoritmos básicos de simulación. Para la definición de modelos acoplados puede utilizarse el editor gráfico GVLE, mientras que los modelos atómicos se definen mediante C++.

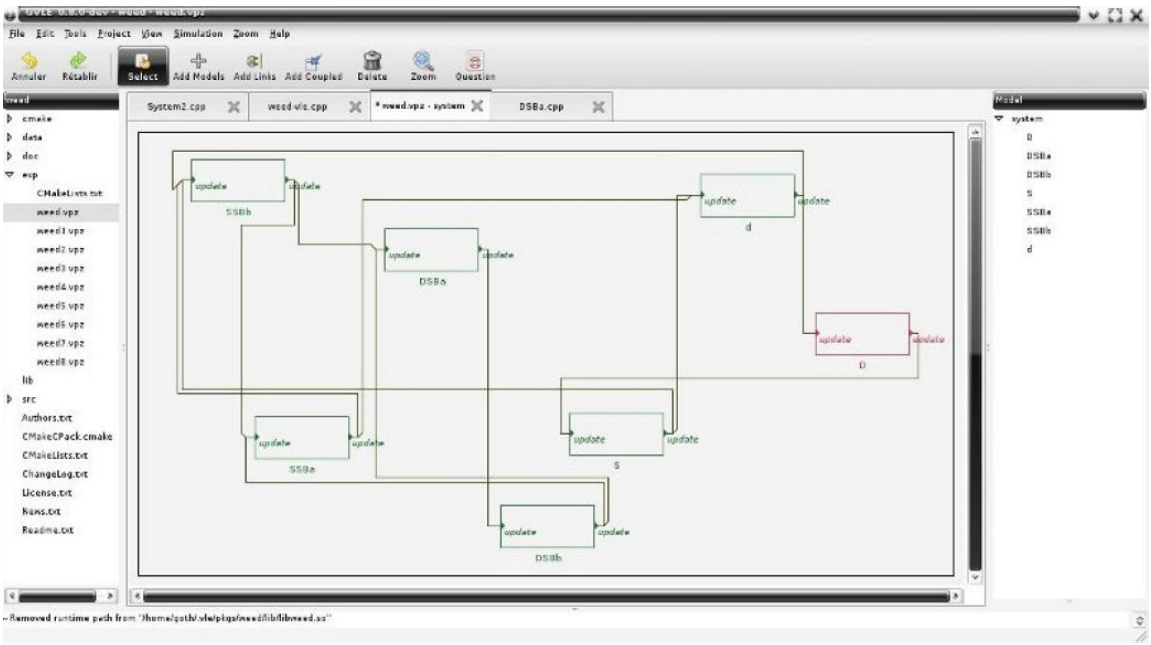


Figura 11 - Interfaz gráfica de VLE [QDRT07]

En general las herramientas gráficas permiten describir el acoplamiento entre modelos, pero el comportamiento de los modelos atómicos debe realizarse en el lenguaje del simulador (C++ o Java).

2.3 Herramienta CD++

En este trabajo nos enfocaremos en la implementación conocida como CD++ desarrollada en el Departamento de Computación, FCEyN de la UBA. En esta sección se verán brevemente las diferentes versiones, y se detallarán los procesos para crear y simular modelos en la herramienta. Se verán también otras herramientas gráficas relacionadas a CD++ que facilitan el proceso de M&S.

CD++ es una herramienta basada en el paradigma de objetos, que implementa el formalismo DEVS presentado por Zeigler. CD ++ cuenta con diferentes versiones para distintos escenarios. La versión de Parallel CD ++ permite la ejecución de modelos DEVS y Cell-DEVS en ambientes distribuidos/paralelos [TW03]. Embedded CD++ [YW07] permite correr simulaciones en ambientes embebidos donde por lo general es necesario proveer salidas en tiempo real [GW02].

La versión StandAlone de CD++ es utilizada por alumnos de la materia Simulación de Eventos Discretos que se dicta todos los años en el Departamento de Computación de la FCEyN de la UBA. Se utiliza tanto como herramienta para realizar los trabajos prácticos así como también como plataforma de experimentación para estudiar, extender y modificar el simulador.

CD++ fue utilizado satisfactoriamente para modelar problemas y aplicaciones de diferentes áreas de interés como por ejemplo tráfico urbano [DVW01][DVW01], sistemas físicos [AWG03], arquitectura de computadores [WDSW01][WDSW01] , en sistemas embebidos para el control de tráfico de redes de comunicación [JWBC09], etc.

Dentro de estas clases que provee el framework CD++ se distinguen dos jerarquías más importantes: la de los modelos y la de sus simuladores.
La raíz de la jerarquía de modelos es la clase **Model** que consta de dos subclases directas: **Atomic** y **Coupled** como se muestra en la figura:

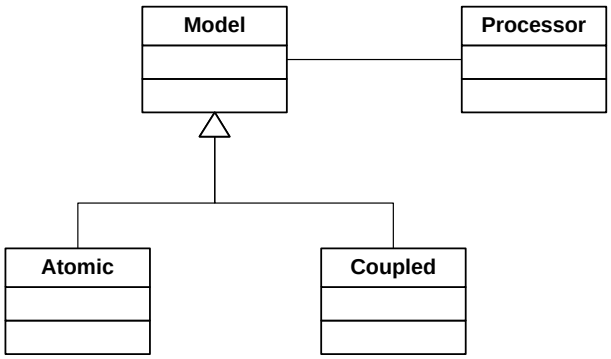


Figura 12 - Jerarquía de Modelos de CD++ [Wai09].

La clase **Atomic** posee métodos que deben ser implementados obligatoriamente para crear un nuevo modelo atómico. Dichos métodos definen el comportamiento del nuevo modelo. Estos métodos son:

- ***initFunction()***: es invocado cuando se inicializa la simulación. Se puede utilizar para definir valores iníciales.
- ***externalFunction()***: implementa la función de transición externa. Es invocada cuando un evento llega a alguno de los puertos del modelo.
- ***internalFunction()***: implementa la función de transición interna. Es invocada cuando el tiempo de simulación es igual al tiempo programado por la función de avance de tiempo.
- ***outputFunction()***: implementa la función de salida y puede generar valores de salida por los puertos. Es invocada antes de la función de transición interna.

Para crear un nuevo modelo atómico el usuario debe:

1. Escribir una clase que derive de **Atomic** sobrecargando los 4 métodos virtuales.
2. Modificar el archivo **register.cpp** para registrar el nuevo modelo.
3. Recompilar el simulador.

CD++ incluye también una librería de modelos que contiene:

- **Modelos incluidos:** CD++ posee una serie de modelos simples de uso general. Estos modelos están implementados extendiendo directamente la clase Atomic del framework. Algunos de los modelos que se suelen incluir con las distribuciones de CD++ son Queue, Generator, Transducer, etc.
- **Intérpretes de gramáticas de alto nivel:** También se incluyen en CD++ intérpretes de gramáticas específicas, que permiten describir modelos sin utilizar un lenguaje de programación. Ejemplos de estos intérpretes son el de CELL-DEVS, DEVS-Graphs, ATLAS, etc.

Utilizando librería de modelos, los usuarios pueden definir nuevos modelos atómicos. Los nuevos modelos atómicos pueden definirse mediante alguno de los lenguajes de alto nivel ó utilizando el lenguaje de programación C++ extendiendo directamente la clase Atomic del framework. En el último caso, es necesario recompilar el simulador para que incluya los nuevos modelos dentro del ejecutable.

2.3.1 Modo de utilización

Para crear y simular un modelo DEVS utilizando la herramienta CD++ el usuario suele atravesar un proceso que consta de 4 etapas: implementación, estabilización, ejecución, análisis de los resultados.

El motor de simulación de *CD++* se alimenta de un archivo con el modelo a ejecutar, otro archivo con los posibles eventos externos y un conjunto de variables que definen el contexto de ejecución del mismo. Realiza la simulación y devuelve un archivo con el resultado de la simulación.

La interfaz con el usuario es por medio de línea de comandos. Los parámetros que recibe el ejecutable del simulador son los siguientes:

```
simu [-ehlmotdpvbfrsqw]
e: events file (default: none)
h: show this help
l: message log file (default: /dev/null)
m: model file (default : model.ma)
o: output (default: /dev/null)
t: stop time (default: Infinity)
d: set tolerance used to compare real numbers
p: print extra info when the parsing occurs (only for cells models)
v: evaluate debug mode (only for cells models)
b: bypass the preprocessor (macros are ignored)
f: flat debug mode (only for flat cells models)
r: debug cell rules mode (only for cells models)
s: show the virtual time when the simulation ends (on stderr)
q: use quantum to calculate cells values
w: sets the width and precision (with form xx-yy) to show numbers
```

Figura 13 - Parámetros de ejecución de CD++

2.3.2 Creación de Modelos atómicos en CD++

Para la implementación del modelo, el primer paso es desarrollar los modelos atómicos que darán el comportamiento básico. Estos modelos atómicos se implementan extendiendo la clase **Atomic** de C++ del framework de CD++ siguiendo los pasos explicados anteriormente.

A continuación se muestra la implementación del clásico modelo de una cola (Queue) [Wai09]. Para esto es necesario definir 2 archivos:

1. Un archivo de cabecera (con extensión .h) que contenga la definición del estado del modelo (variables de estado), la declaración de la clase incluyendo puertos de entrada y salida.

```
class Queue : public Atomic {
       public:
Queue(); // Default constructor
       protected:
// Model Behavior Methods
Model &initFunction();
Model &externalFunction( const ExternalMessage & );
Model &internalFunction( const InternalMessage & );
Model &outputFunction( const InternalMessage & );
private:
// Model Ports
const Port &in, &stop, &done;
Port &out;

// Model State
Time preparationTime;
typedef list<Value> ElementList ;
ElementList elements ;
Time timeLeft;
}; // class Queue
```

Figura 14 - Queue.h: Definición del modelo cola

2. Un archivo de código C++ (con extensión .cpp) que contenga la implementación del constructor de la clase y los 4 métodos virtuales.

```
Queue::Queue( const string &name ): Atomic( name )
, in( addInputPort( "in" ) )
, stop( addInputPort( "stop" ) )
, done( addInputPort( "done" ) )
, out( addOutputPort( "out" ) )
, preparationTime( 0, 0, 10, 0 )
{
       string time( MainSimulator::Instance().getParameter( description(),
"preparation" ) ) ;
       if( time != "" )
              preparationTime = time ;
}
```

Figura 15 - Queue.cpp. constructor del modelo cola

```
Model &Queue::initFunction()
{
elements.erase( elements.begin(), elements.end() ) ;
return *this ;
}
```

Figura 16 - Queue.cpp. initFunction del modelo cola

```
Model &Queue::externalFunction( const ExternalMessage &msg )
{
if( msg.port() == in )
       {
elements.push_back( msg.value() ) ;
if( elements.size() == 1 )
holdIn( active, preparationTime );
       }

if( msg.port() == done )
       {
elements.pop_front() ;
if( !elements.empty() )
holdIn( active, preparationTime );
       }

if( msg.port() == stop )
if( state() == active && msg.value() )
              {
timeLeft = nextChange();
passivate();
} else
if( state() == passive && !msg.value() )
holdIn( active, timeLeft );
return *this;
}
```

Figura 17 - Queue.cpp: función de transición externa del modelo cola

```
Model &Queue::internalFunction( const InternalMessage & )
{
passivate();
return *this ;
}
```

Figura 18 - Queue.cpp. Función de transición interna del modelo cola

```
Model &Queue::outputFunction( const InternalMessage &msg )
{
sendOutput( msg.time(), out, elements.front() ) ;
return *this ;
}
```

Figura 19 - Queue.cpp. Función de salida del modelo cola

Una vez que esta creado el modelo atómico es necesario registrarlo en el simulador. Para esto se modifica el archivo **register.cpp** utilizando la función **MainSimulator::registerNewAtomics**. La Figura 21Figura 20 muestra como registrar el modelo Queue:

```
void MainSimulator::registerNewAtomics()
{
        SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Queue>() , "Queue" ) ;
}
```

Figura 20 - register.cpp. Registrando el modelo Queue

Una vez que la clase del nuevo modelo atómico está definida y registrada, se está en condiciones de recompilar el simulador (para esto es necesario utilizar el código fuente del simulador). La compilación genera un ejecutable que permite utilizar el nuevo modelo.

Como vimos anteriormente el simulador no permite utilizar modelos atómicos en forma aislada sino que es necesario especificar un modelo acoplado para simular. Es por esto que para testear los nuevos modelos atómicos es necesario crear un modelo acoplado simple que lo contenga. Este modelo acoplado tendrá como único componente al nuevo modelo atómico y la misma cantidad de puertos unidos uno a uno con los puertos del modelo atómico. Más adelante se explicará en detalle la definición de modelos acoplados. Por ejemplo el modelo acoplado simple para la Cola será:

```
[Top]
Components: queue@Queue
In: in stop done
Out: out
Link: in in@queue
Link: stop stop@queue
Link: done done@queue
Link: out@queue out
```

Figura 21 - Modelo acoplado simple para el modelo atómico Queue

Especificando el modelo acoplado simple y un archivo de eventos externos (como se explicará más adelante) se puede ejecutar la simulación para testear el nuevo modelo atómico. La simulación genera archivos de salida que se utilizan para analizar los resultados de la simulación y corregir potenciales errores.

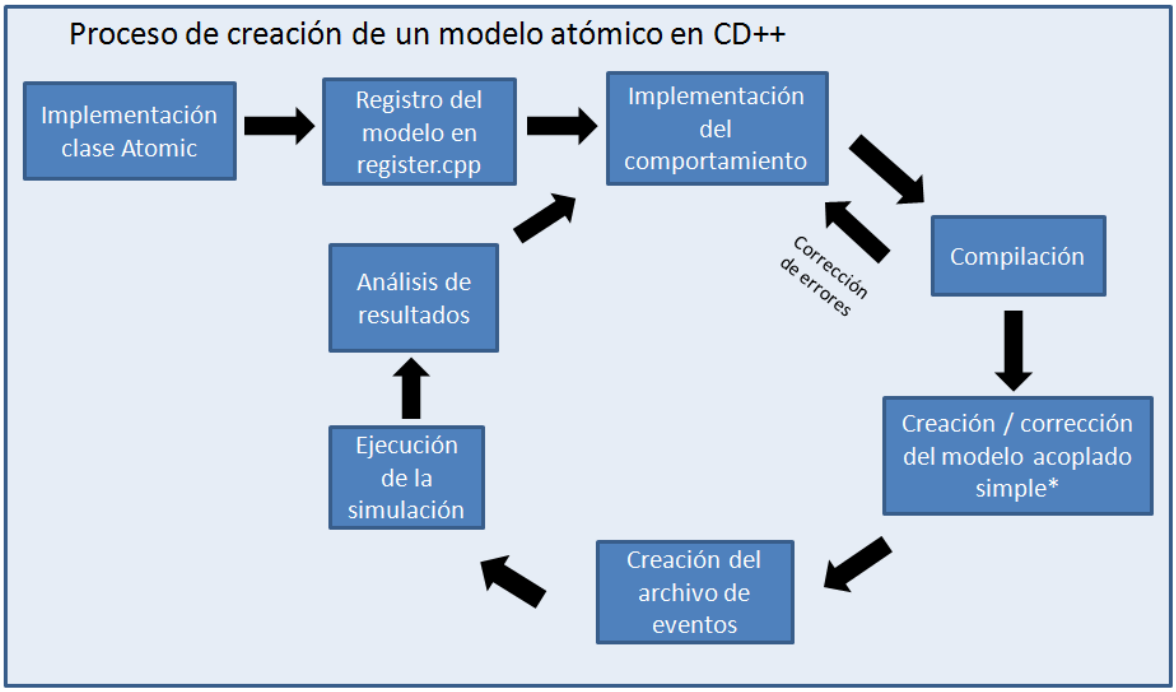


Figura 22 - Proceso de creación de un modelo atómico en CD++.

2.3.3 Creación de Modelos acoplados en CD++

Una vez definidos y creados todos los modelos atómicos necesarios (repitiendo los pasos anteriores para cada modelo atómico), el segundo paso para implementar un modelo completo es crear los modelos acoplados.

Los modelos acoplados en CD++ se definen utilizando un lenguaje de especificación creado especialmente para este propósito. Mediante este lenguaje se definen los componentes que forman parte del modelo acoplado (modelos atómicos y otros modelos acoplados), los puertos de entrada/ salida y sus uniones. Los modelos acoplados suelen definirse en archivos con extensión **.ma** utilizando la siguiente sintaxis:

- **Nombre del modelo**

```
[model_name]
```

Todos los modelos comienzan su definición especificando en una línea su nombre entre corchetes. Por ejemplo, el modelo Queue comienza con la línea **[Queue]**. El modelo acoplado superior (el que se encuentra más alto en la jerarquía y contiene todos los otros modelos) debe ser el primero en definirse y debe llamarse **[Top]** obligatoriamente.

- **Componentes**

```
components: model_name1[@atomicClass1] model_name2[@atomic-Class2]…
```

Para listar los componentes que componen al modelo acoplado se utiliza la palabra **components** seguida de la lista de componentes. Los modelos atómicos se especifican con el nombre de instancia del modelo seguido del símbolo **@** y el nombre del tipo de modelo (por

ejemplo queueInstance@Queue). Para los modelos acoplados, solo es necesario definir el nombre del modelo y luego definir el modelo dentro del mismo archivo. Es obligatorio definir al menos un componente para cada modelo acoplado.

- **Puertos de salida**

```
out: port_name1 port_name2 ...
```

Para listar los puertos de salida de un modelo acoplado se utiliza la palabra **out** seguida de los nombres de los puertos de salida. No es obligatorio definir puertos de salida.

- **Puertos de entrada**

```
in: port_name1 port_name2 ...
```

Para listar los puertos de entrada de un modelo acoplado se utiliza la palabra **in** seguida de los nombres de los puertos de entrada. No es obligatorio definir puertos de entrada.

- **Uniones**

```
link : source_port[@model] destination_port[@model]
```

Para definir como se unen los diferentes puertos y componentes de un modelo acoplado se utiliza la palabra **link** seguida del puerto origen y el puerto destino. Si en los puertos no se define el nombre del modelo, se asume que el puerto pertenece al modelo que está siendo definido (por ejemplo el **link: in start@queue_instance**, define una unión entre el puerto **in** del modelo que se está definiendo y el puerto **start** del modelo **queue_instance**). No es obligatorio definir uniones y puede haber tantas como sea necesario.

La siguiente figura muestra la definición del modelo GPT [Wai09]:

```
[top]
components : transducer@Transducer generator@Generator Consumer
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out
[Consumer]
components : queue@Queue processor@Processor
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
```

Figura 23 - Definición de un modelo acoplado DEVS

2.3.4 Creación de modelos DEVS-Graphs en CD++

Una extensión a los modelos atómicos de *CD++* soporta la creación de modelos DEVS-Graphs mediante una gramática llamada G*ADScript* [DCW03]. Esto le permite a modeladores que no poseen gran experiencia desarrollando en la plataforma C++ crear modelos atómicos DEVS con mayor facilidad. Si bien desarrollar los modelos directamente en C++ provee mucha más flexibilidad para definir su comportamiento, utilizar una notación gráfica permite pensar el

problema en una manera más abstracta concentrándose en la solución y no en los detalles de implementación.

Este desarrollo en CD++ extiende la clase **Atomic** del framework implementando un intérprete para la gramática GGADScript que genera el comportamiento de los modelos.

GGADScript es una especificación textual equivalente a la definición formal de modelos DEVS-Graphs que permite su interpretación y ejecución dentro del framework de CD++. Los modelos atómicos DEVS-Graphs suelen definirse en archivos con extensión .cdd utilizando GGADScript. A continuación se mostrará los detalles más importantes del lenguaje GGADScript. La especificación completa del lenguaje GGADScript puede encontrarse en [DCW03].

- **Nombre del modelo**

```
[modelname]
```

La definición de un modelo DEVS-Graphs debe comenzar especificando en una línea el nombre del modelo encerrado entre corchetes. Por ejemplo el modelo **Clock** comienza con la línea **[Clock]**.

- **Estados**

```
state : stateId1 stateId2 … stateIdn
stateId1 : lifetime
stateId2 : lifetime
...
stateIdn : lifetime
initial: statename
```

Para definir los estados del modelo se utiliza la palabra **state** seguida de los nombres de los estados. Puede haber tantos estados como sea necesario.

Para definir el tiempo de vida de un estado se utiliza el nombre del estado seguido del tiempo deseado. Por ejemplo, para asignarle un tiempo de vida de 3 segundos al estado *start* se especifica **start: 00:00:03:00.** Es obligatorio que todos los estados tengan definidos una y solo una vez su tiempo de vida.

Para definir el estado inicial del modelo se utiliza la palabra **initial** seguida del nombre del estado. Por ejemplo para definir al estado *start* como estado inicial se especifica **initial: start**. Es obligatorio que el modelo defina uno y solo un estado inicial.

- **Transiciones internas**

```
int : source destination [q!v]* ( { (action;)* } )
```

Para definir una transición interna se utiliza la palabra **int** seguida del estado origen, el estado destino, una lista de (puerto salida, valor) y una lista de acciones. Puede haber tantas transiciones internas como sea necesario y una transición interna puede enviar múltiples eventos de salida y ejecutar múltiples acciones. Tanto los valores de los puertos como las acciones pueden definir expresiones.

Una transición interna se ejecuta al consumirse el tiempo de su estado origen y provoca que el modelo pase al estado destino. Al ejecutarse una transición interna se envían por los puertos de salida en la lista los valores especificados (**q!v** representa enviar el valor **v** por el puerto **q**). En la ejecución de la transición interna también se ejecutan las acciones listadas (Las acciones son asignaciones del resultado de evaluar expresiones a variables). Por ejemplo la transición **int : start work p1!8 p2!12 {var1=6}** se ejecuta el transcurrir el tiempo de vida del estado **start** dejando al modelo en el estado **work**, la variable **var1** con valor **6**, envía por el puerto **p1** el valor **8** y por el puerto **p2** el valor **12.**

- **Transiciones externas**

```
ext : source destination EXPRESSION([p?v]*) { (action;)* }
```

Para definir una transición externa se utiliza la palabra **ext** seguida del estado origen, el estado destino, la condición de ejecución y lista de acciones. Puede haber tantas transiciones externas como sea necesario y una transición externa puede ejecutar múltiples acciones.

Una transición externa se ejecuta cuando se cumplen todas las siguientes condiciones:

- Al llegar un evento externo
- El modelo se encuentra en el estado origen de la transición externa
- La condición de ejecución evalúa a verdadero al momento de la llegada del evento externo

En todo momento de la simulación puede haber como máximo una transición externa que cumpla todas las condiciones a la vez. De haber más de una transición externa que cumpla todas las condiciones el modelo abortará la simulación.

- **Puertos de salida**

```
out : port1 port2 ... portn
```

Para listar los puertos de salida del modelo se utiliza la palabra **out** seguida de los nombres de los puertos de salida. No es obligatorio definir puertos de salida.

- **Puertos de entrada**

```
in : port1 port2 ... portn
```

Para listar los puertos de entrada del modelo se utiliza la palabra **in** seguida de los nombres de los puertos de entrada. No es obligatorio definir puertos de entrada.

- **Variables**

```
var : var1 var2 var3 …
var1 : value1
var2 : value2
```

Para definir las variables del modelo se utiliza la palabra **var** seguido de los nombre de las variables. Puede haber tantos estados como sea necesario. Opcionalmente se pueden definir

los valores iniciales de las variables utilizando el nombre de la variable seguido del valor deseado. Por ejemplo **var1: 5** inicializa la variable **var1** con el valor **5.**

- **Expresiones**

Las transiciones hacen uso de expresiones para definir las acciones y en el caso de las transiciones externas para definir la condición de ejecución. Estas expresiones son combinaciones de operaciones que toman parámetros (constantes, variables o puertos). Por ejemplo la operación Add toma dos valores y devuelve la suma de ellos (**Add (port1, 5)**) devuelve el valor del puerto **port1** incrementado en **5**). Estas operaciones se pueden componer para formar expresiones complejas.

CD++ tiene incorporada una serie de operaciones que se pueden utilizar para definir expresiones, pero también es posible definir nuevas operaciones en C++. A continuación se listan las funciones incorporadas a CD++.

Función	**Descripción**
Add(n1, n2)	Sum of n1 and n2
And(n1, n2)	Return true if both n1 and n2 are true
Any(port)	Return true if the port has a valid value
Between(n1, n2, n3)	Return true if n1 ≤ n2 ≤ n3
Compare(n1, n2, n3, n4, n5)	Return n3, n4, or n5 if n1 is greater than, equal to, or less than n2
Divide(n1, n2)	n1/n2
Equal(n1, n2)	Return true if n1 = n2; otherwise, return false
Greater(n1, n2)	Return true if n1 > n2; otherwise, return false
Less(n1, n2)	Return true if n1 < n2; otherwise, return false
Minus(n1, n2)	n1 - n2
Multiply(n1, n2)	n1 * n2
Not(n)	Return the negation of n
NotEqual(n1, n2)	Return true if n1 != n2
Or(n1, n2)	Return true if n1 or n2 is true
Pow(n1, n2)	Return n1 power n2
Rand(n1, n2)	Generate a random value between n1 and n2
Value(n)	Return the value of n

Figura 24 - Funciones para definir expresiones en modelos DEVS-Graphs mediante GGADScript en CD++

La siguiente figura muestra un ejemplo de modelo atómico DEVS-Graphs definido con el lenguaje GGADScript:

```
[controller]
state: stopping stdbyStop moving Stopped stdbyMov aux1
initial : stdbyStop
in: button stop sensor
out: move
var: floor cur_floor direction
ext: stdbyStop moving Equal(button,cur_floor)?0 {floor = button;direction =
```

```
compare(cur_floor,floor,2,0,1);}
ext: stdbyMov aux1 Equal(sensor,floor)?0 {cur_floor = sensor;}
ext: stdbyMov Stopped Equal(sensor,floor)?1 {cur_floor = sensor;}
ext: stopping stdbyStop Value(stop)?1
int: moving stdbyMov move!direction
int: aux1 stdbyMov
int: Stopped stopping move!0
stopping:00:00:00:00
stdbyStop:00:00:1000:00
moving:00:00:00:00
Stopped:00:00:00:00
stdbyMov:00:00:1000:00
aux1:00:00:00:00
floor:0
cur_floor:0
direction:0
```

Figura 25 - Ejemplo de un modelo atómico DEVS-Graphs definido con GGADScript

2.3.5 Definición de eventos externos

Como vimos anteriormente, se puede especificar los eventos externos que recibirá. Los eventos externos se definen en forma separada a la descripción de los modelos en archivos con extensión **.ev**. Este archivo consiste de una secuencia de líneas, donde cada línea describe un evento con el siguiente formato:

```
HH:MM:SS:MS PUERTO VALOR
```

Figura 26 - Formato del archivo de eventos

El nombre de los puertos debe coincidir con algunos de los puertos de entrada del modelo que se desea simular. El valor puede ser cualquier valor entero o de punto flotante.

En la siguiente Figura, se puede apreciar como en el segundo **10** (00:00:10:00) el puerto **in** asume el valor **1**. Luego en el segundo **15** el puerto **done** asume el valor **1.5**. En el segundo **30** el puerto **in** asume el valor **0.271** y en el **31** el valor **-4.5.**

```
00:00:10:00 in 1
00:00:15:00 done 1.5
00:00:30:00 in .271
00:00:31:00 in -4.5
...
```

Figura 27 - Ejemplo de archivo de eventos

2.3.6 Resultados de una simulación

Al terminar la ejecución de una simulación se generan dos archivos de salida como se especifica en con las opciones –l y –o del simulador: El archivo que registra el flujo de mensajes entre los modelos que participan en la simulación (en general con extensión **.log**) y el archivo con los valores de los puertos de salida (en general con extensión **.out**).

El archivo con el flujo de mensajes muestra en cada línea el tipo de mensaje, la hora a la que se produjo, quien lo emitió y el destinatario. Se puede encontrar una especificación detallada de este tipo de registro en [Wai03]. Estos archivos también son utilizados por herramientas de visualización para analizar de forma gráfica el flujo de la simulación.

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para life(02)
```

```
Mensaje I / 00:00:00:000 / life(02) para life(0,0,0)(03)
Mensaje I / 00:00:00:000 / life(02) para life(0,0,1)(04)
Mensaje D / 00:00:00:000 / life(0,0,0)(03) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,0,1)(04) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,0,2)(05) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,1,0)(06) / ... para life(02)
Mensaje * / 00:00:00:100 / Root(00) para top(01)
Mensaje * / 00:00:00:100 / top(01) para life(02)
Mensaje * / 00:00:00:100 / life(02) para life(0,0,0)(03)
Mensaje * / 00:00:00:100 / life(02) para life(0,0,1)(04)
Mensaje Y / 00:00:00:100 / life(0,0,0)(03) / out / 0.000 para life(02)
Mensaje D / 00:00:00:100 / life(0,0,0)(03) / ... para life(02)
Mensaje Y / 00:00:00:100 / life(0,0,1)(04) / out / 10.500 para life(02)
Mensaje D / 00:00:00:100 / life(0,0,1)(04) / ... para life(02)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,0,0)(03)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,1,0)(06)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,2,0)(09)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,9,0)(30)
...
```

Figura 28 - Ejemplo de archivo de flujo de mensajes

El archivo con los valores de los puertos de salida consiste de una secuencia de líneas, donde cada línea describe un evento con el mismo formato que el archivo de eventos de externo:

```
HH:MM:SS:MS PUERTO VALOR
```

Figura 29 - Formato del archivo de salida

A diferencia del archivo de eventos externo, en este caso los puertos son puertos de salida.

```
00:00:15:00 out 1
00:00:40:00 out 1.5
00:01:31:00 out 2.75
...
```

Figura 30 - Ejemplo de archivo de salida

En la siguiente sección se describirán las herramientas que simplifican la creación de modelos detallada previamente.

2.3.7 Herramientas gráficas para CD++

CD++ provee los algoritmos de simulación DEVS, clases que permiten extender el framework e intérpretes de alto nivel que permiten definir nuevos modelos. Se han desarrollado herramientas que facilitan el uso de CD++ para la creación de nuevos modelos, el análisis de los resultados de las simulaciones, etc. A continuación veremos algunas de estas herramientas.

Drawlog[Wai09] es una herramienta que permite mostrar gráficamente los resultados de la ejecución de un modelo celular. Conforma parte del paquete de distribución *CD++*. Se alimenta de un archivo con el resultado de la ejecución de una simulación, realiza un procesamiento de los mismos y devuelve un segundo archivo de texto con matrices que de valores temporales asumidos por las celdas del modelo a cada instante de tiempo.

Es un desarrollo hecho en C++. La interfaz con el usuario es por medio de línea de comandos como se muestra a continuación:

```
drawlog –[?hmtclwp0]
where:
```

```
? Show this message
h Show this message
m Specify file containing the model (.ma)
t Initial time
c Specify the coupled model to draw
l Log file containing the output generated by SIMU
w Width (in characters) used to represent numeric values
p Precision used to represent numeric values (in characters)
0 Don't print the zero value
```

Figura 31 - Parámetros de ejecución de Drawlog

El archivo de salida (por lo general de extensión **.drw**) muestra sucesivos instantes de tiempo y los valores que asumió cada una de las celdas del modelo en cada momento.

```
Line : 238 - Time: 00:00:00:000
   0    1    2    3    4    5    6    7    8    9
 +---------------------------------------------------+
0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
2| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
5| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
 +---------------------------------------------------+
Line : 358 - Time: 00:00:01:000
   0    1    2    3    4    5    6    7    8    9
 +---------------------------------------------------+
0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
2| 24.0 24.0 35.8 24.0 24.0 24.0 24.0 24.0 -6.3 24.0|
3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
5| 24.0 24.0 24.0 24.0 24.0 39.5 24.0 24.0 24.0 24.0|
6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 -4.0 24.0|
9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
 +---------------------------------------------------+
.....
```

Figura 32 - Ejemplo del archivo de salida Drawlog

Graflog [Wai09] es una herramienta que permite animar gráficamente la salida de *Drawlog*. Integra parte del paquete de distribución de *CD++*. Se alimenta con el archivo de salida de Drawlog y anima en pantalla las sucesivas matrices pintando cada celda según una escala de colores definida por el usuario.

Es un desarrollo hecho en JAVA y se encuentra disponible embebido como Applet en una página Web.

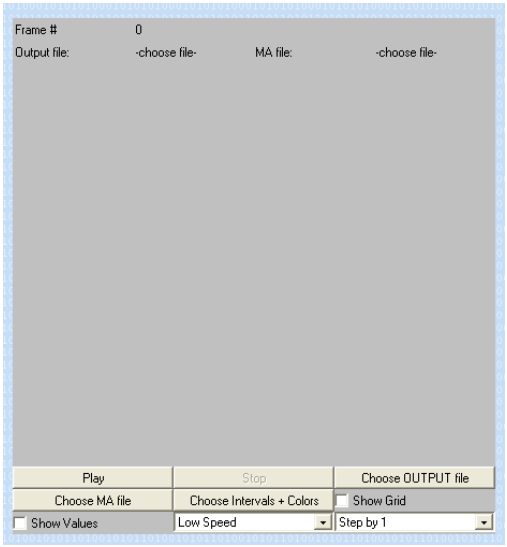


Figura 33 - Interfaz gráfica de Graflog

En la siguiente figura se pueden ver algunas capturas de pantalla del resultado de una animación de Graflog del modelo de FireSpread:

Figura 34 - Animación de un modelo celular en Graflog.

*GGADTool [*CDW04*]* es una herramienta que permite crear gráficamente modelos DEVS acoplados y modelos atómicos DEVS-Graphs. Es un desarrollo hecho en Visual Basic por lo que se encuentra solo disponible para la plataforma Windows.

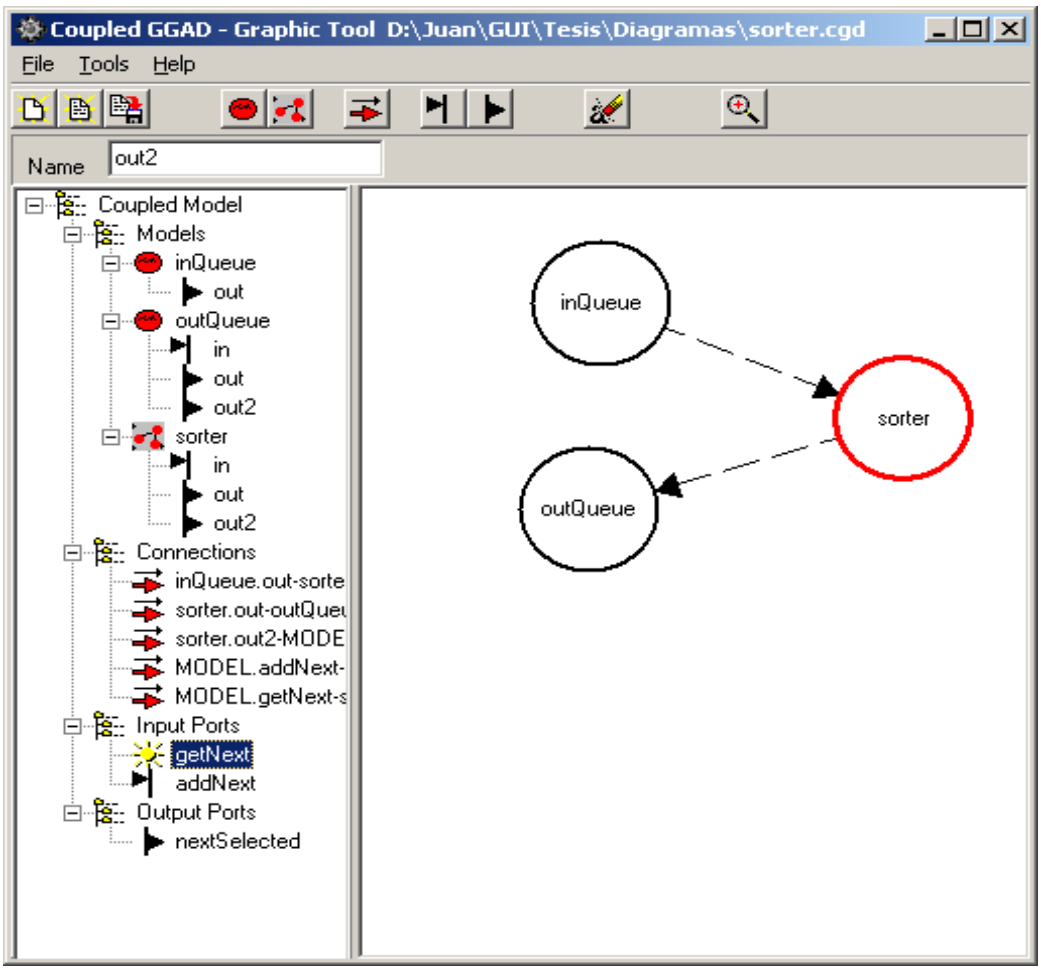


Figura 35 - Interfaz gráfica de GGADTool

Entre sus funciones podemos destacar:

- Diseño visual de modelos DEVS acoplados, incluyendo los modelos atómicos y/o acoplados que lo componen, la interconexión de los mismos, y sus puertos de entrada y puertos de salida.
- Diseño de la estructura y funcionalidad de un modelo atómico DEVS-Graphs, incluyendo los estados, transiciones internas y externas, los puertos de entrada y salida, las variables de estado y las acciones asociadas a las transiciones.
- Almacenamiento de los modelos generados y exportación de los mismos en formato GGADScript para ser simulados mediante la aplicación *CD++*.[Dob03]

CD++Modeler [CW06] es una herramienta con funcionalidades básicas similares a GGADTool, pero mejora y agrega nuevas características. Es un desarrollo hecho en JAVA por lo que puede correr tanto en entornos Windows como Linux.

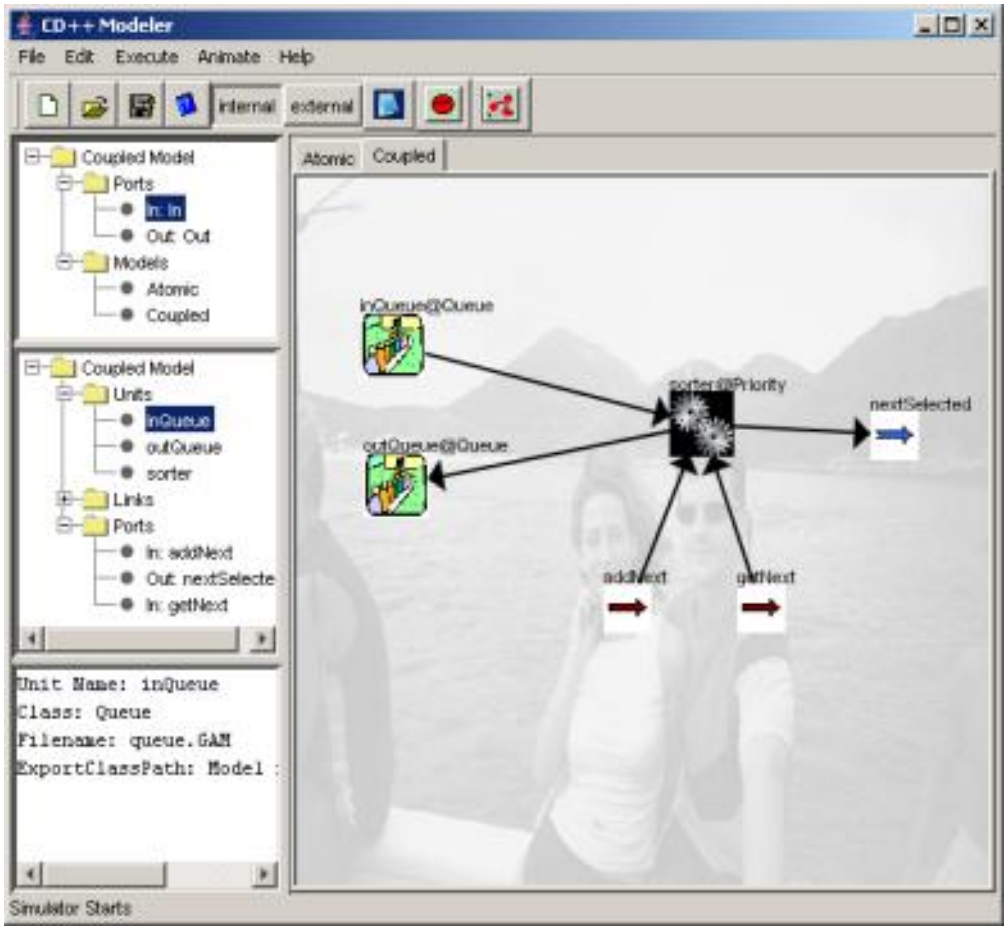


Figura 36 - Interfaz gráfica de CD++Modeler. [CW06]

Además de las funcionalidades básicas CD++Modeler incorpora nuevas herramientas para analizar los resultados de una simulación. CD++Modeler provee tres componentes que permiten animar modelos celulares, modelos atómicos y modelos acoplados.

La animación de modelos puede alimentarse directamente de la salida de ejecución haciendo innecesario el uso de *Drawlog*. La siguiente figura muestra su interfaz:

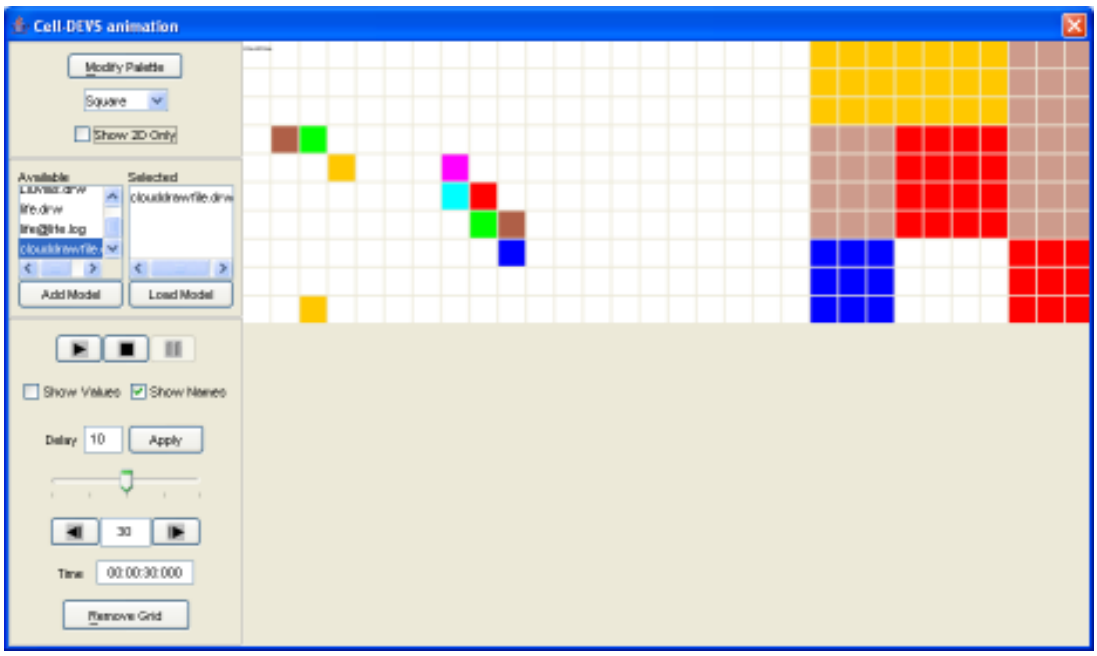


Figura 37 - Animación de un modelo celular con CD++Modeler. [CW06]

La animación de modelos atómicos muestra los valores que asumen los puertos a lo largo del tiempo. La siguiente figura muestra el resultado de una animación:

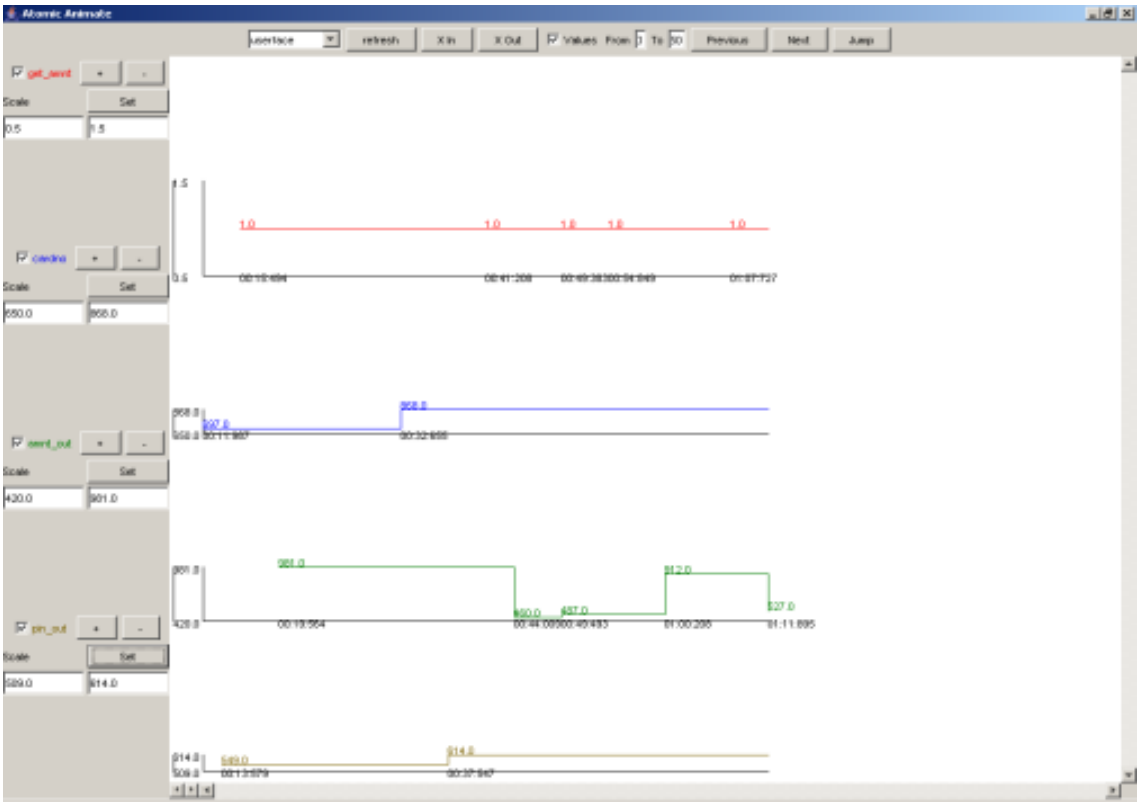


Figura 38 - Animación de un modelo atómico con CD++Modeler. [CW06]

La animación de modelos acoplados muestra el modelo y los valores que pasan por sus puertos. La siguiente figura muestra el resultado de una animación:

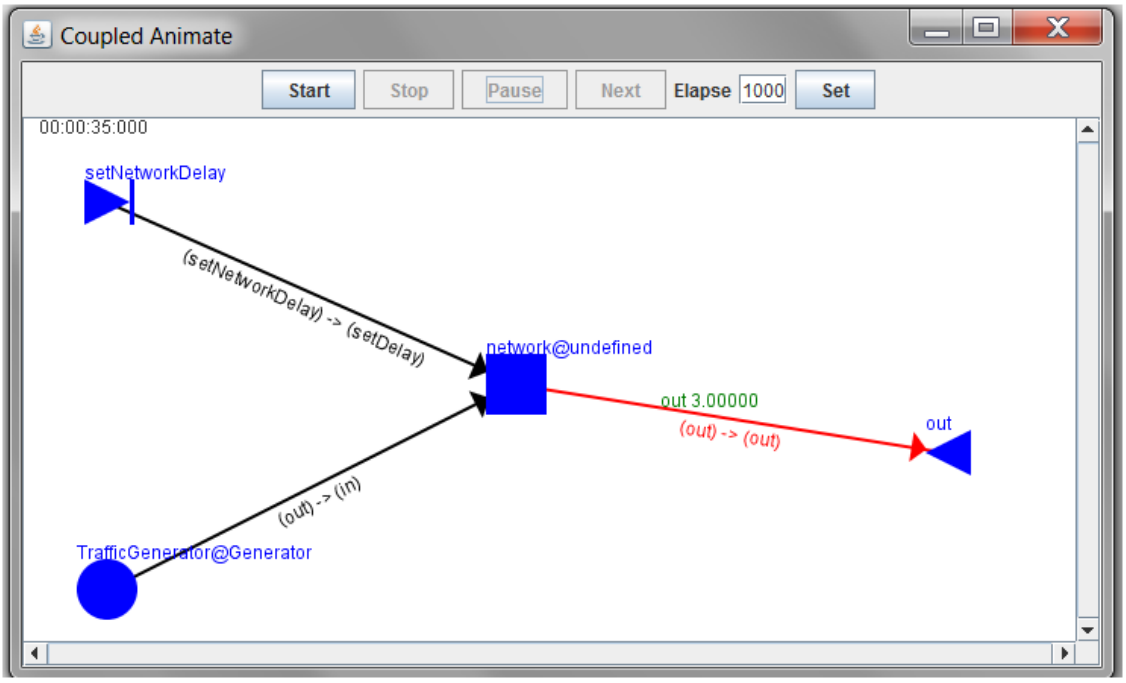


Figura 39 - Animación de un modelo acoplado con CD++Modeler. [CW06]

CD++Builder [CW07] es un plugin para Eclipse que permite realizar tareas de desarrollo en CD++ en forma integrada y simplificada. Originalmente fue desarrollado para Eclipse 2.1 y luego se migro para las nuevas versiones de Eclipse 3.1.x.

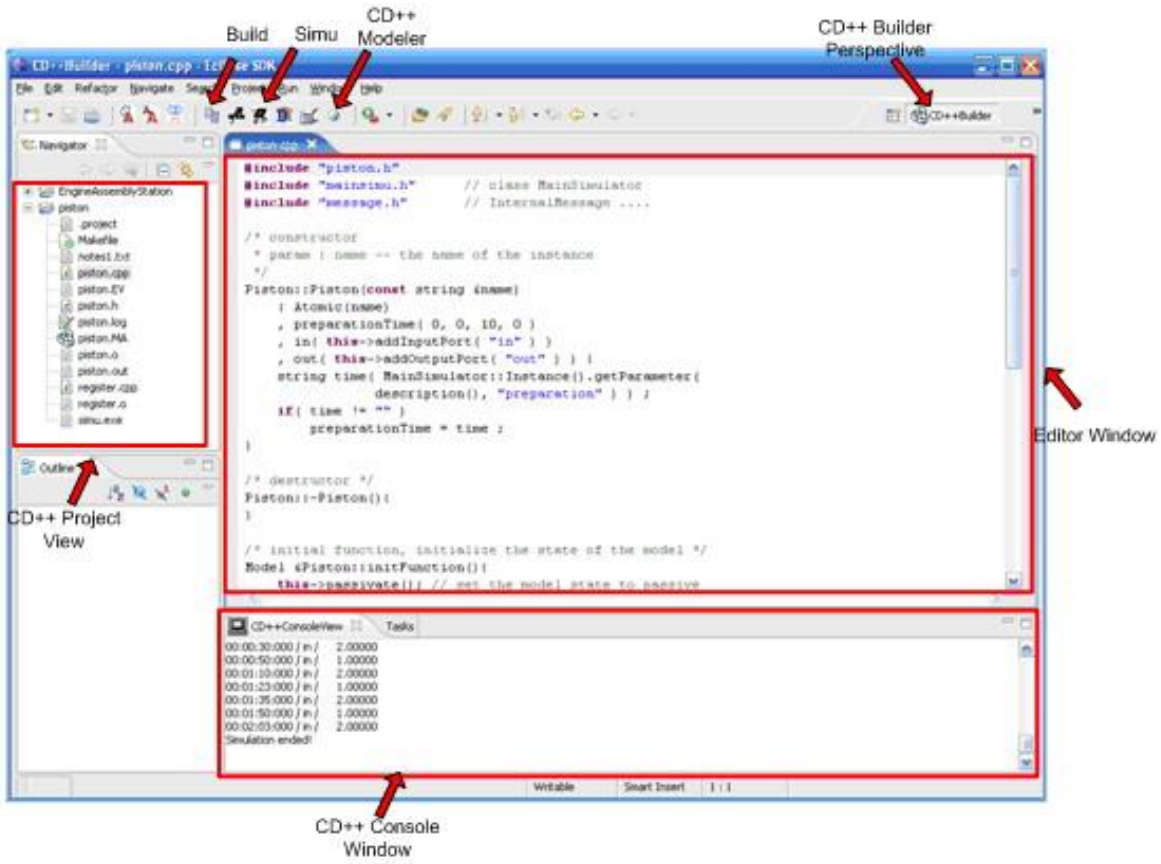


Figura 40– Interfaz CD++Builder

Provee la posibilidad de compilar y simular modelos fácilmente a través de botones de acción. También ofrece la posibilidad de ejecutar herramientas relacionadas con CD++ como por ejemplo Drawlog, CD++Modeler, GGADTool, etc.

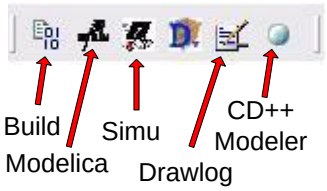


Figura 41– Botones de acción de CD++Builder

CD++Builder implementa un editor de texto para Eclipse que colorea el código de los archivos con la gramática de modelos acoplados de CD++ (con extensión .ma).

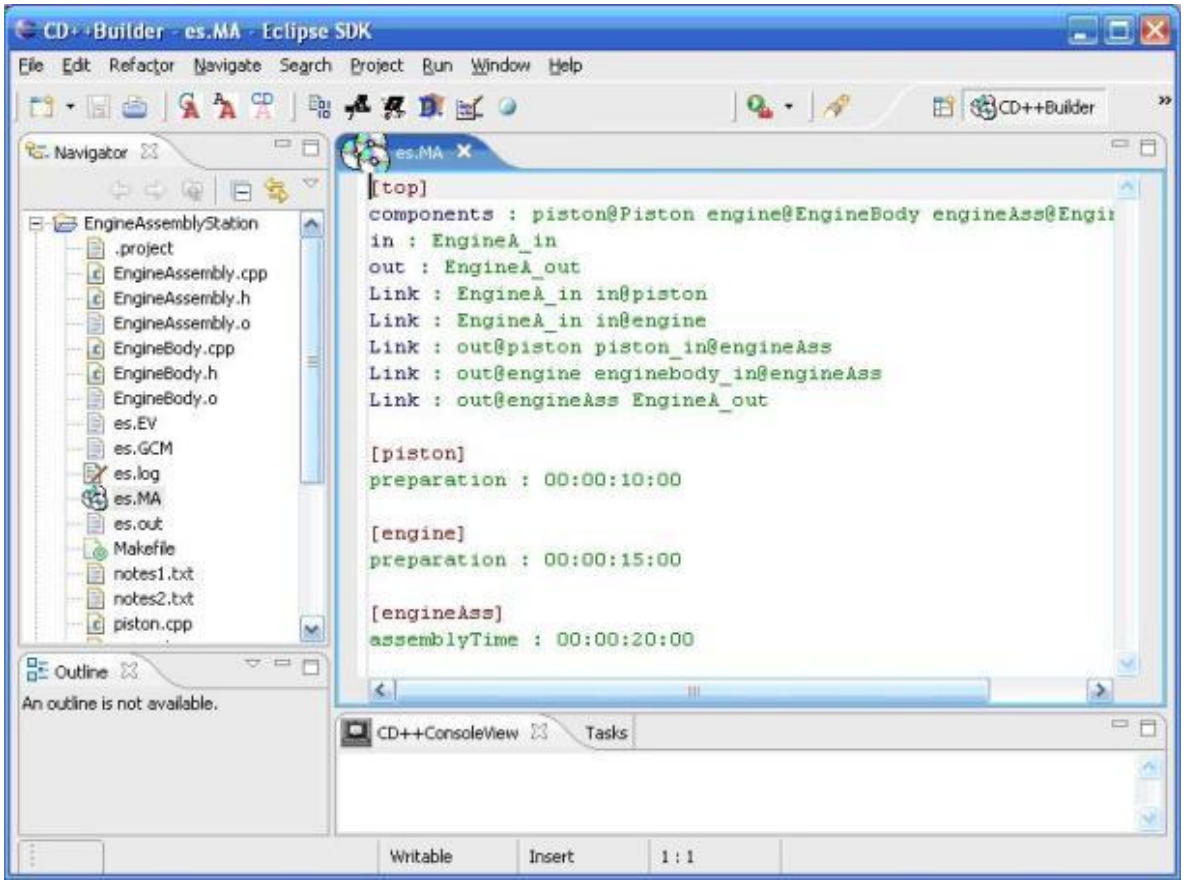


Figura 42– Editor de modelos acoplados de CD++Builder.

2.4 Limitaciones de las herramientas actuales

Un problema común a todas las herramientas vistas en la sección "2.2 Herramientas De Simulación Existentes" es que si bien soportan la creación y modificación visual de modelos DEVS, fueron desarrolladas desde cero a medida de cada simulador particular sin estar basadas en una interfaz estándar. Cada herramienta está implementada en diferentes tecnologías (Java, Visual Basic, etc.) y no utilizan frameworks o plataformas que permitan agregar nuevas funcionalidades. Para extender la funcionalidad de estas herramientas gráficas es necesario conocer los detalles de su implementación y en muchos casos incluso también los detalles del simulador que utilizan. En algunos casos, los editores gráficos están directamente vinculados al simulador, donde la definición del comportamiento del modelo se entremezcla con la definición gráfica del mismo.

Por otro lado, salvo para el simulador CD++, es necesario conocer algún lenguaje de programación (ya sea Java para DEVSJava o JDEVS, o bien C++) para desarrollar nuevos modelos atómicos. Los simuladores no proveen lenguajes descriptivos de alto nivel para describir el comportamiento de los modelos atómicos. Esto genera que las herramientas permitan describir en forma gráfica el acoplamiento entre diferentes modelos pero no el comportamiento de los modelos atómicos. Esta limitación imposibilita o hace muy limitada la utilización de los simuladores basados en DEVS a aquellos usuarios que tengan poca experiencia en el área de programación, ya que para crear

nuevos modelos atómicos es necesario escribir código y compilar en el lenguaje en que fue desarrollado el simulador.

Más aun, si bien los simuladores obligan a los usuarios a definir nuevos modelos atómicos mediante lenguajes de programación, el soporte que brindan las herramientas para esto es limitado. En algunos casos los editores gráficos permiten definir algunas propiedades de los modelos atómicos. En el caso de VLE, se pueden definir los nombres de los puertos de entrada y salida, pero estos no están asociados al código C++ por lo que deben mantenerse consistentes en forma manual. VLE no provee editores específicos para desarrollar código C++ que ayuden en la implementación del comportamiento de los modelos atómicos.

En el caso de CosMOs, se permite especificar las interfaces de los modelos atómicos en forma gráfica y luego generar código JAVA con la estructura básica para implementar el modelo para el simulador DEVSJava. La integración con editores específicos de C++ facilita el desarrollo con coloreado de código e intelliSense. De cualquier manera, una vez que la estructura fue generada, esta no puede ser modificada ya que las secciones de código que definen la interfaz del modelo quedan bloqueadas. Si bien la funcionalidad de generar la estructura de código para los modelos atómicos facilita mucho el modelado y provee un estándar de desarrollo, las secciones de código bloqueadas dificultan la actualización de los modelos, obligando a regenerar el código si por ejemplo se necesita agregar un nuevo puerto.

El editor gráfico de PowerDEVS permite definir nuevos modelos atómicos mediante drag&drop y brinda un editor específico para la editar modelos atómicos DEVS en C++. El editor de modelos atómicos posee coloreo para el código C++ y permite ver en diferentes solapas los métodos que deben implementarse (transición interna, transición externa, función de salida y función de avance de tiempo), con código y comentarios que ayudan en la implementación. Este editor es de gran ayuda para usuarios que no son expertos en el framework de PowerDEVS (ya que esconde detalles de implementación como el encabezado de los métodos, la estructura de los archivos de código C++, la declaración de la clase, etc.), aunque no provee ciertas funcionalidades como mostrar o ir a un determinado número de línea (dificultando la tarea de debug). Sin embargo, la debilidad más importante es que la interfaz del modelo atómico (cantidad de puertos de entrada, cantidad de puertos de salida y parámetros que recibe el modelo) no se identifica automáticamente a partir del código C++ sino que debe especificarse en forma independiente, pudiendo quedar la representación gráfica y la implementación del modelo inconsistentes.

Las herramientas asociadas a CD++ tienen la ventaja de permitir definir el comportamiento de los modelos atómicos utilizando DEVS-Graphs, sin utilizar lenguajes de programación. Sin embargo, también poseen limitaciones que describiremos a continuación.

Por un lado cada herramienta es un desarrollo independiente, implementadas utilizado diferentes tecnologías (CD++Modeler en java, GGADTool en Visual Basic, CD++ en C++ y los editores textuales bajo la plataforma de Eclipse, etc.). Cada una define sus propias interfaces, implementa diferentes infraestructuras, formatos para serializar los modelos, etc. Esto provoca que la interfaz de usuario

de cada una sea completamente diferente al resto, lo que obliga al usuario a tener que asimilar cada una por separado. Por otro lado, al estar desarrolladas en diferentes tecnologías no pueden ser utilizadas dentro de un mismo entorno (en el Capítulo 3 - Implementación: CD++Builder 2.0" varemos que CD++Modeler puede integrarse a Eclipse al correr ambos en Java). Para realizar distintas tareas (modelar, simular, analizar los resultados), las herramientas actuales obligan a los usuarios a tener que ejecutar diferentes aplicaciones, exportar modelos de una herramienta a la otra, etc. Esto hace que el proceso de modelado y simulación sea más lento, tenga una curva de aprendizaje mucho mayor y dé lugar a cometer errores.

Cada herramienta es una aplicación que se instala y ejecuta en forma separada. Esto provoca que los usuarios de CD++ tengan que realizar diversas instalaciones y mantener actualizadas cada una de las aplicaciones. Los usuarios no poseían un mecanismo de notificación de las actualizaciones de las herramientas de CD++, por lo que debían consultar periódicamente el sitio web en forma manual para mantener las herramientas actualizadas. Esto provocaba que los usuarios deban llevar cuenta de cuál fue la última versión instalada en sus máquinas y que la gran mayoría utilice herramientas desactualizadas. Por otro lado, para instalar una nueva versión de las herramientas era necesario primero remover las versiones viejas y luego proceder con la nueva instalación. Los problemas que esto trae aparejado se pueden hacer evidentes por ejemplo en la siguiente situación que se dio durante el transcurso de este trabajo: un usuario de China encontró un bug del simulador relacionado con las funciones de GGADScript. Para solucionar el bug modificamos el código del simulador, lo recompilamos y le enviamos al usuario el nuevo ejecutable con las instrucciones de cómo instalarlo. Si bien esto resolvió el problema para este usuario en particular, el resto de los usuarios de CD++ siguieron trabajando sobre el simulador que contenía el bug y no había forma de notificarles de la nueva versión.

Utilizando la versión anterior de CD++Builder y CD++Modeler para crear y simular un nuevo modelo gráficamente, se debían realizar a grandes rasgos los siguientes pasos:

1. Ejecutar CD++Modeler.
2. Crear el modelo en CD++Modeler.
3. Exportar el modelo (esto crea el archivo .ma necesario para CD++).
4. Ejecutar Eclipse.
5. En Eclipse, importar los archivos exportados previamente.
6. Ejecutar la simulación utilizando Eclipse.

Uno de los primeros problemas que surgen al revisar los pasos anteriores, es la necesidad de utilizar dos herramientas diferentes para modelar y simular. Esto trae aparejado los problemas de necesitar varias instalaciones, dificultad de mantener actualizadas ambas herramientas, y tener que adaptarse a las diferentes interfaces.

Otro de los problemas del escenario anterior es que la definición gráfica de los modelos y la definición utilizada para la simulación no se mantienen sincronizadas. Esto provoca que deba actualizarse el modelo gráficamente y luego realizarse los pasos 3, 5 y 6 repetidamente al

modificar iterativamente el modelo (ver Figura 43). Si se realizan cambios en los archivos de CD++ (los .ma de acoplados o .cdd de DEVS-Graphs), estas modificaciones se pierden en la especificación gráfica. Por otro lado, ciertos errores de la simulación se reportan indicando el número de línea del archivo CD++ donde se produjo, por lo que es difícil identificar cómo debe modificarse el modelo gráfico para solucionar los errores. Todo esto provoca que al utilizar las herramientas gráficas anteriores, el proceso de modelado se vuelva tedioso y complicado.

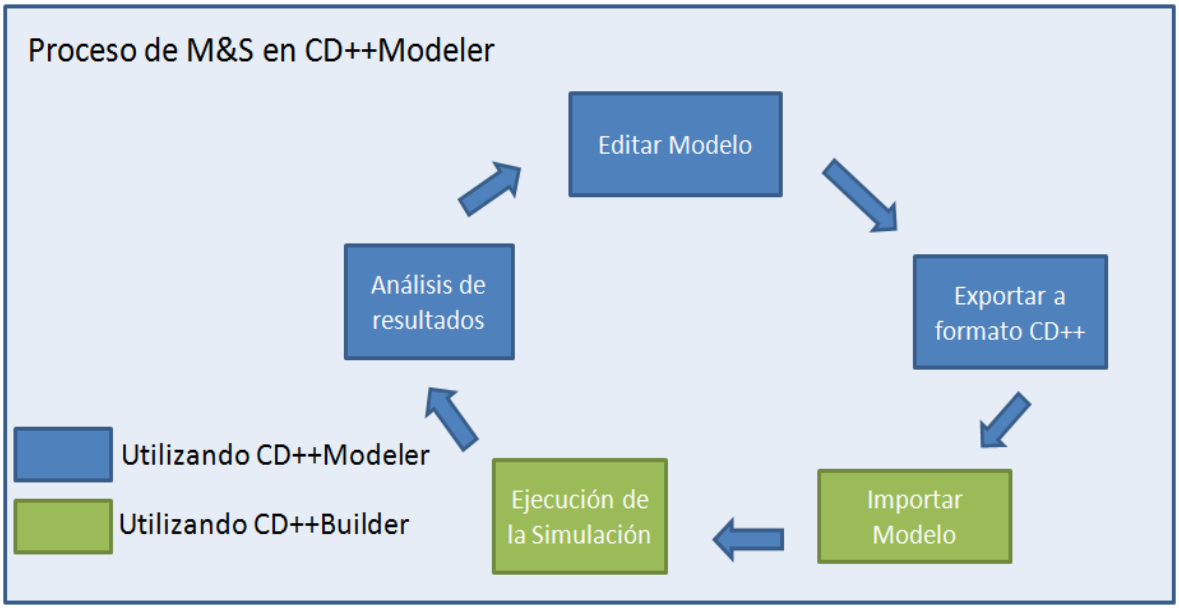


Figura 43 - Proceso de M&S con CD++Modeler.

Los lenguajes de alto nivel de modelos acoplados CD++ y DEVS-Graphs tienen el poder para describir por completo modelos DEVS y a su vez habilitan la representación gráfica del modelo sin restringirla a visualizaciones específicas. Es decir, la misma definición de un modelo puede ser tomada por distintos editores y graficarse de formas diferentes, ya que estos lenguajes de alto nivel no describen propiedades gráficas. De esta manera, la definición del comportamiento del modelo y su representación gráfica están claramente separadas. Las figuras, tamaños, posición, colores, y toda la información gráfica se guarda en un archivo diferente a la definición del modelo. Mientras esta separación es conceptualmente correcta, presenta algunos desafíos para implementar editores gráficos.

En el caso de CD++Modeler y GGADTool, que utilizan formatos propios para almacenar la información gráfica de los modelos DEVS-Graphs (.gcm y .cgd respectivamente), una vez que el modelo fue definido es necesario exportarlo para poder realizar la simulación ya que son incompatibles con el formato de CD++. Más allá de la incomodidad de tener que realizar una operación extra antes de poder simular los modelos, esto genera dos nuevas limitaciones. Por un lado es difícil mantener consistentes la descripción gráfica de los modelos con la definición de comportamiento utilizada para la simulación. Por ejemplo, si al realizarse la simulación se encuentra un error en el modelo, no puede modificarse su definición, sino que debe cambiarse su

descripción gráfica y reexportar el modelo. Por otro lado, si bien es posible extraer la definición del modelo a partir de su representación gráfica, la operación opuesta (generar una representación gráfica a partir de la definición del modelo) no estaba disponible en ninguna de las herramientas. CD++ posee un gran repositorio de modelos, de manera que es una gran limitación para utilizar estas herramientas: los modelos que ya están desarrollados y testeados no pueden ser abiertos por las herramientas gráficas anteriores. Por ejemplo, si uno quisiera editar el modelo ATM (se puede descargar del repositorio de modelos http://cell-devs.sce.carleton.ca/mediawiki/index.php/Model_Samples) no puede hacerlo de forma gráfica con estas herramientas y la única manera de hacerlo es a través del editor de texto.

CD++Modeler y GGADTool permiten crear modelos acoplados y modelos atómicos DEVS-Graphs. Sin embargo no es posible definir modelos acoplados que estén compuestos por modelos atómicos definidos en C++. Esta es una gran limitación para el modelador ya que no todos los modelos atómicos pueden ser definidos mediante DEVS-Graphs. Un modelo DEVS podría tener sus modelos atómicos simples definidos mediante DEVS-Graphs (ya que pueden ser definidos y editados gráficamente) y los modelos atómicos más complejos definidos en C++ extendiendo la clase **Atomic** del framework de CD++. Sin embargo estas herramientas no lo permiten, restringiendo la complejidad de los modelos que pueden generarse gráficamente, permitiendo crear sólo modelos relativamente simples.

Por otro lado, no existe ninguna herramienta para CD++ que facilite la creación de modelos atómicos definidos en C++. Como se describió en la sección "2.3 Herramienta CD++", el proceso para añadir nuevos modelos atómicos al framework de CD++ consta de varios pasos. Todos estos pasos deben ser realizados de forma completamente manual ya que las herramientas gráficas sólo permiten crear modelos atómicos DEVS-Graphs, y Eclipse sólo provee editores textuales que no son específicos para CD++.

Si bien algunos pasos no pueden ser automatizados (como la implementación particular de las funciones de transición), otros pasos pueden ser facilitados o realizados totalmente en forma automática. Por ejemplo la registración de los modelos atómicos en el archivo **register.cpp** puede automatizarse por completo y se puede facilitar la creación de los modelos al proveer una estructura estándar para implementar y extender la clase **Atomic** del framework CD++, similar a la generación de código de CosMOs. Esta estandarización para implementar modelos atómicos en C++ puede facilitar el aprendizaje, disminuir los errores y promover buenas prácticas de programación y modelado.

Por otro lado, para realizar este análisis se crearon una serie de modelos atómicos y acoplados utilizando las herramientas gráficas (específicamente en CD++Modeler, ya que es una evolución de GGADTool). Durante este proceso se identificaron algunas falencias de usabilidad general en las aplicaciones. Algunos ejemplos son:

- **Acciones poco intuitivas:** Selección con doble clic, drag&drop con doble clic y un sólo componente a la vez, imposibilidad de cambiar propiedades directamente, etc.

- **Falta de opciones habituales:** Deshacer, rehacer, copiar y pegar componentes, shortcuts, menús contextuales, etc.
- **Perdida de links al editar submodelos:** Al editar componentes que se encuentran dentro de un modelo acoplado se pierden los links del mismo.
- **Limitaciones al reutilizar modelos:** No es posible importar otros modelos de CD++Modeler. Los modelos importados de CD++ no son editables. Al importar modelos no detecta los puertos (es necesario crearlos a mano con el mismo nombre).
- **Visualización completa del modelo:** En los modelos DEVS-Graphs los puertos y variables no son visibles gráficamente y los estados iniciales son indistinguibles del resto. En modelos acoplados, los puertos no muestran los nombres y los componentes no muestran sus puertos.
- **Editor de expresiones limitado:** Para definir transiciones internas y externas de modelos DEVS-Graphs se utilizan expresiones. Estas expresiones no se validan, no existen herramientas que faciliten su creación (como por ejemplo ver los elementos del modelo y funciones que pueden utilizarse).
- **Limitaciones gráficas:** Los links son rectos por lo que se solapan, colores y tamaño de componentes, estilos y colores de letras estáticos.
- **No es posible crear varias instancias de un mismo modelo:** Para componer un modelo acoplado por dos modelos atómicos idénticos, es necesario crear por separado cada uno de los modelos atómicos, ya que no es posible reutilizar su definición.

Por último, dentro del proceso de análisis se estudió tanto la posibilidad de extender las herramientas existentes para resolver los problemas planteados como la posibilidad de reutilizar funcionalidades ya desarrolladas. Dado que estas herramientas fueron desarrolladas desde cero, su extensibilidad es limitada. Para extender el comportamiento es necesario conocer en profundidad su implementación, ya que no proveen infraestructura para agregar funcionalidad en forma desacoplada. Por otro lado, no poseen tests que verifiquen el correcto funcionamiento de las aplicaciones (ya sean tests manuales o automatizados), por lo que es difícil agregar nuevas funcionalidades y poder seguir asegurando el correcto desempeño de las anteriores funcionalidades.

Para resolver todas estas limitaciones, se decidió extender la versión anterior de CD++Builder a través de nuevos plugins de Eclipse. En el próximo capítulo se describirá en profundidad la implementación de CD++Builder 2.0, un plugin para Eclipse que integra todas las características y funcionalidades disponibles para el simulador CD++, facilitando la creación y simulación de los modelos. La incorporación de todas las funcionalidades existentes en el entorno de desarrollo Eclipse permitirá mejorarlas y potenciarlas. Como CD++Builder está basado en la infraestructura de Eclipse (desarrollada específicamente para soportar extensiones a la funcionalidad e interfaz de usuario), para agregar nuevas funcionalidades no es necesario conocer las implementaciones anteriores y es posible incluso agregar nuevos módulos sin tener el código fuente de CD++Builder. Eclipse también facilitará la implementación de nuevos mecanismos de actualización e instalación, permitiendo mantener actualizadas más fácilmente todas las herramientas desarrolladas en este

entorno. Mostraremos el desarrollo de nuevos editores gráficos que permitan editar y actualizar los modelos tanto en forma gráfica como textual, manteniendo ambas definiciones consistentes y sincronizadas automáticamente. Estos editores permitirán componer gráficamente modelos complejos tanto por medio de modelos atómicos DEVS-Graphs como por modelos atómicos definidos en C++. Para estos últimos, se proveerán generadores de código y estructuras estándar (templates) que ayuden en su implementación.

Capítulo 3 - Implementación: CD++Builder 2.0

En el presente capítulo se describirá la implementación desarrollada para dar solución a los problemas planteados en el capítulo anterior: el entorno de simulación CD++Builder 2.0. Se describirá brevemente dicho entorno y sus características más importantes. Luego se discutirá la arquitectura utilizada, las principales decisiones tecnológicas y detalles de su implementación.

3.1 Arquitectura y Decisiones Tecnológicas

El diagrama en capas de la Figura 44 muestra el rol de cada componente de CD++ y sus relaciones principales. En la capa inferior (situada por encima del sistema operativo) el simulador CD++ implementa los algoritmos de simulación basados en los formalismos DEVS y Cell-DEVS. CD++ provee diferentes versiones de los simuladores abstractos (por ejemplo, simuladores paralelos, achatados, en tiempo real, etc.).

En el nivel inmediato superior (Librerías) CD++ provee una librería básica de modelos atómicos que incluye un generador, un transductor, una cola, etc. que pueden ser utilizadas directamente para definir modelos acoplados. Por otro lado, la librería incluye intérpretes para lenguajes de alto nivel. Los algoritmos base de simulación junto con la capa de librerías suelen distribuirse integrados bajo el nombre de Simulador CD++. Los intérpretes aceptan archivos de entrada que provienen de la capa de modelado. Estos archivos pueden definir la composición de modelos acoplados, la especificación de modelos Cell-DEVS o de modelos atómicos DEVS-Graphs, sin necesidad de utilizar ningún lenguaje de programación (se utilizan especificaciones declarativas propias de cada definición). Existen también otros intérpretes en el nivel de Librerías, utilizados en dominios de modelado específicos tales como ATLAS (para describir tráfico urbano) o M/CD++ (para describir modelos continuos usando Bond Graphs y Modelica) y que son distribuidos en versiones especiales de CD++. Todos estos interpretes están implementados usando las clases base, por lo que pueden ser utilizados independientemente de la versión del simulador.

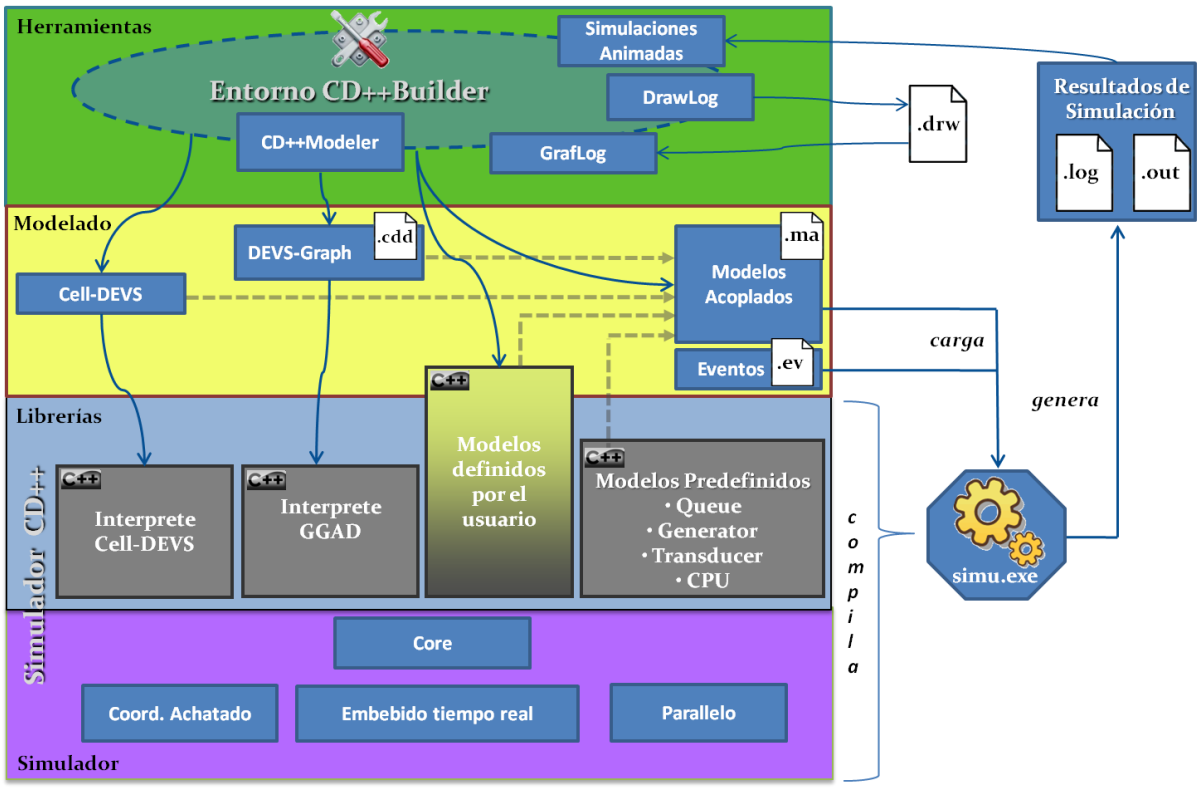


Figura 44 - Niveles de organización en la herramienta CD++.
Herramientas de productividad, lenguajes de **Modelado** de alto nivel, **Librerías** de intérpretes para modelos, y librerías base del **Simulador**. Ciclo de ejecución típico.

En el caso de que se desee definir el comportamiento de nuevos modelos atómicos cuya complejidad exceda la capacidad expresiva del paradigma DEVS-Graphs, es posible crear modelos atómicos genéricos que extiendan la clase Atomic de CD++. Para esto es necesario utilizar el lenguaje de programación C++, por lo que luego de programar un nuevo modelo se necesitará recompilar el simulador para que incluya el nuevo código introducido.
Para ejecutar una simulación, se debe especificar el archivo que contiene la definición de acoplamiento de modelos (.ma) y los eventos de entrada externos (.ev), entre otras opciones. La corrida de una simulación genera dos archivos de salida que pueden ser utilizados luego para analizar el flujo resultante de la simulación: el archivo *.out* contiene filas informando puerto, valor y tiempo de los eventos de salida del modelo, mientras que el archivo .log contiene toda la mensajería interna e información de sincronización intercambiada entre los modelos.
En la capa superior (Herramientas), diferentes aplicaciones fueron desarrolladas para facilitar la visualización de los archivos de salida, como por ejemplo Drawlog para simulaciones Cell-DEVS y CD++Modeler para animar el flujo de mensajes en las simulaciones de modelos acoplados DEVS y los valores de salida en modelos atómicos DEVS.

CD++Builder integra bajo una misma interfaz al resto de las herramientas, brindando un único punto de acceso para todas las funcionalidades relacionadas al simulador CD++. CD++Builder se encuentra por

encima de todas las capas anteriores y provee acceso a editores gráficos para especificar el comportamientos de los modelos, que a su vez generan especificaciones de alto nivel que son interpretadas (consumidas) por las capas inferiores.
Esta arquitectura en capas y su separación clara entre simulador, librerías intermedias, definición de modelos, y herramientas de productividad, permite modificar el simulador base sin afectar los modelos y visualizadores previamente desarrollados. Las mismas herramientas e interfaces de usuario pueden ser utilizadas para la definición de modelos, sin importar si los mismos serán luego simulados en un único procesador, en un ambiente distribuido o en un sistema embebido en tiempo real.

El entorno CD++Builder 2.0 está implementado basado en varios frameworks de Eclipse, que proveen la interfaz de usuario general y los servicios base para los plugins. Un requerimiento muy importante para CD++Builder es que fuese simple de extender, ya que nuevas funcionalidades y herramientas son desarrolladas continuamente para el simulador CD++ por distintos grupos de investigación geográficamente distantes. La arquitectura de Eclipse es extensible a través de plugins, y permite desarrollar nuevas funcionalidades independientes para CD++Builder e integrarlas fácilmente.

Un ejemplo de la extensibilidad flexible de CD++Builder es CD++Repository [cita], una base de datos de modelos CD++ que puede accederse a través de internet, que fue desarrollada en paralelo con el trabajo de esta tesis y en forma totalmente independiente, para luego ser integradas como parte del mismo software.

Para implementar los editores gráficos mencionados anteriormente, se consideraron y analizaron varios frameworks. Algunos de ellos incluyen librerías gráficas básicas como Standard Widget Toolkit (SWT), Abstract Window Toolkit (AWT) y Swing; y otras más específicas tales como Draw2D y Graphical Editing Framework (GEF). Las primeras tres librerías están basadas en Java y proveen controles generales para desarrollar formularios y ventanas. Sin embargo, no son prácticos para manipular figuras y no proveen ninguna infraestructura para el desarrollo de plugins de Eclipse. Las figuras en dos dimensiones son los elementos principales de Draw2D, que está basado en la librería SWT. GEF permite generar editores gráficos para modelos preexistentes, que deben cumplir ciertas características, y tiene algún soporte para correr bajo la plataforma de Eclipse [EGEF10]. Por esta razón se eligió finalmente utilizar el framework GMF (Eclipse Graphical Modeling Framework) [EGMF10], ya que esta librería integra y extiende GEF y Eclipse Modeling Framework (EMF) y está específicamente orientada a la creación de editores gráficos de Eclipse.

GMF utiliza el patrón de arquitectura Model-View-Controller (MVC) como herramienta para separar el *modelo* de su representación gráfica. Cabe destacar que en lo que resta de esta sección usaremos *modelo* para referirnos al conjunto de clases que representan el plugin desarrollado, y no debe confundirse con los modelos DEVS vistos anteriormente. El paradigma arquitectónico MVC fue utilizado satisfactoriamente para otros editores DEVS, y otros frameworks similares, para editores gráficos de lenguajes visuales [EEHT05].

La Figura 45 muestra una descripción conceptual de las diferentes tecnologías utilizadas en CD++Builder y sus relaciones.

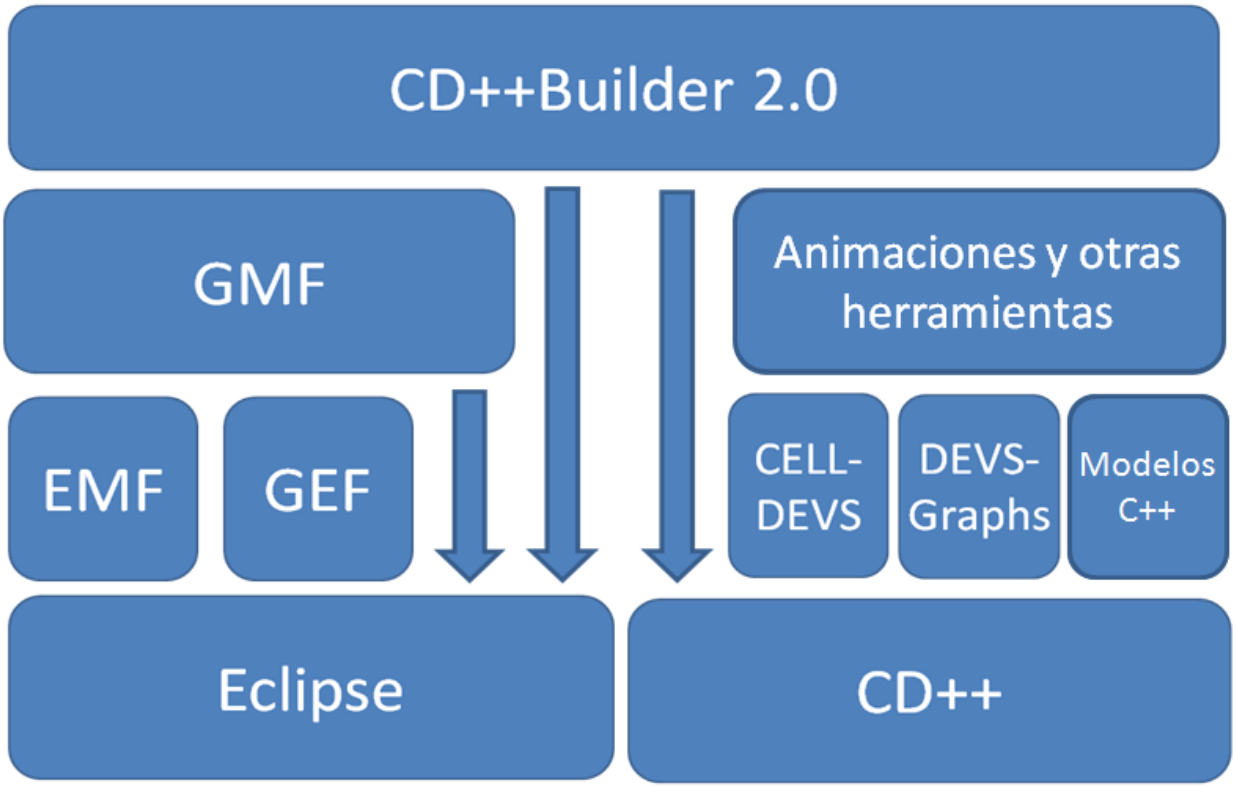


Figura 45 - Arquitectura tecnológica de CD++Builder.

EMF se sitúa por encima de Eclipse y es utilizado para definir el *modelo*. Si bien pueden utilizarse otras tecnologías o bien desarrollar el *modelo* del editor manualmente, se decidió utilizar EMF ya que es el framework utilizado por defecto por GMF y provee varios servicios para especificar y mantener el *modelo*. EMF es un framework que permite definir, generar y mantener el código asociado a un modelo de datos estructurado. A partir de la especificación de un *modelo* (descripta en el formato estándar XMI [XMI98] (XML Metadata Interchange) o a través de un editor gráfico), EMF provee librerías y herramientas que permiten crear sencillamente un conjunto de clases Java que implementan el modelo (como el editor gráfico de XMI, wizard para la generación de código, etc.) y una infraestructura con un editor básico. Las clases Java generadas implementan el patrón Observable, brindando métodos que notifican cuando las propiedades del *modelo* cambian, y que son utilizados por GEF y GMF para actualizar la *vista* (parte View en MVC). Esto también ayuda a mantener el *modelo* completamente desacoplado del resto de la implementación, permitiendo utilizarlo para otros desarrollos. Los métodos generados automáticamente por EMF pueden ser modificados y puede agregarse código propio utilizando comentarios específicos para evitar que sean sobrescritos cuando el *modelo* es regenerado. EMF además provee servicios de persistencia y validación sobre el modelo que genera. Dado que se espera que CD++Builder soporte transparentemente el desarrollo de nuevas funcionalidades, EMF puede utilizarse para proveer una abstracción conceptual del universo DEVS, la cual sea reutilizada por

todas las herramientas de CD++ que se sumen al entorno de desarrollo. En la sección "3.2 Detalles de Implementación" de este capítulo se discutirá en mayor detalle el modelo utilizado para representar las entidades DEVS en CD++Builder 2.0.

GEF y GMF proveen clases base que deben ser extendidas para implementar las partes de la *vista* y *controladores* del patrón MVC. GEF extiende Draw2D para facilitar la creación de representaciones gráficas de un modelo y provee varias clases para la implementación de editores basados en Eclipse. Los controladores de GEF necesitan ser provistos con un *modelo* que exponga sus propiedades y notifique cuando ocurren cambios. Es por esto que EMF es un complemento ideal para definir la parte del *modelo* para los *controladores* GEF.

El framework de Eclipse GMF provee un componente de generación de código e infraestructura de runtime para desarrollar editores gráficos basados en GEF y EMF [ST06]. En GMF se pueden distinguir dos módulos. Por un lado el runtime de GMF provee las librerías y clases base para desarrollar nuevos editores. En este sentido GMF es un framework de caja blanca ya que se deben crear nuevas clases, extendiendo las básicas de la librería, para crear nuevos editores. Por otro lado GMF brinda herramientas de generación de código que permiten, a partir de una determinada configuración, generar el código necesario para un nuevo editor. En este otro sentido GMF es un framework de caja negra. Para la construcción de CD++Builder se comenzó por configurar las herramientas para generar código a través de sucesivas iteraciones. Sin embargo, para muchas de las personalizaciones necesarias en los editores DEVS, fue necesario modificar manualmente el código generado y crear/extender nuevas clases.

Para generar el código de los editores gráficos basados en un modelo EMF, GEF necesita ser provisto con tres archivos con diferente información: definición gráfica del modelo (*.gmfgraph*), definición de las herramientas del modelo (*.gmftool*) e información de mapeo del modelo (*.gmfmap*). El archivo XML *.gmfgraph* describe las formas y figuras que van a ser utilizadas en los editores, junto con sus propiedades y como van a ser compuestas y posicionadas. El archivo *.gmftool* se utiliza para definir las herramientas que aparecen en el panel izquierdo del editor, como por ejemplo la herramienta de selección, de zoom, de creación de las entidades, etc. y su agrupación en secciones. Finalmente el archivo *.gmfmap* hace referencia a los dos archivos anteriores para asociar las entidades de EMF con la representación gráfica (definidas en *.gmfgraph*) y sus herramientas (definidas en *.gmftool*).

Para la implementación de los editores de CD++Builder, la generación de código de GMF fue utilizada al principio para definir la apariencia, diseño y comportamiento general. De cualquier manera, muchas de las funcionalidades fueron desarrolladas personalizando y extendiendo el código generado. GMF genera una infraestructura desacoplada, donde los controladores, las vistas y la implementación de los editores de Eclipse están separados del modelo, que se guarda en un proyecto separado. Esto se adapta a los requerimientos de CD++Builder, ya que el modelo puede ser reutilizado por otros plugins de CD++ o DEVS sin que dependan de la implementación de los editores.

3.2 Detalles de Implementación

3.2.1 Estructura de proyectos

Una de las decisiones tomadas para la implementación de CD++Builder 2.0 fue la descomposición del plugin en varias funcionalidades o *features* (proyectos instalables y actualizables en forma independiente). Esto no sólo simplifica la actualización de cada uno en forma independiente, sino que también simplifica la interdependencia de cada funcionalidad del plugin. Originalmente CD++Builder estaba compuesto por un único proyecto que contenía:

1. Código fuente del simulador CD++.
2. Código fuente de CD++Modeler.
3. Código que implementaba las extensiones a la interfaz de Eclipse (botones de ejecución).
4. Código para el editor de la gramática de modelos acoplados (con coloreo de las palabras clave).
5. Los ejecutables de las siguientes herramientas: GGADTool, DEVSView, CD++Modeler, SimService.

Para el desarrollo de las nuevas funcionalidades se separó cada uno de los ítems listados anteriormente en un nuevo proyecto. Las nuevas funcionalidades también fueron creadas en proyectos separados para que puedan ser reutilizadas, testeadas y actualizadas independientemente. En CD++Builder 2.0 se crearon los siguientes proyectos:

1. Proyecto **cdBuilder.model**. Representación en clases e interfaces de la problemática DEVS.
2. Proyectos **cdBuilder.model.edit** y **cdBuilder.model.editor.** Clases que proveen la infraestructura para la edición a través de comandos del modelo y un editor básico del mismo.
3. Proyecto **cdBuilder.editors.coupled.diagram**. Editor gráfico de modelos DEVS acoplados.
4. Proyecto **cdBuilder.editors.atomic.diagram**. Editor gráfico de modelos DEVS atómicos.

Esta separación de las funcionalidades en diferentes proyectos contribuyó además a facilitar el testeo de cada uno por separado. Por cada proyecto listado anteriormente, existe un proyecto asociado que implementa los tests automatizados. Por convención, los proyecto de test llevan el mismo nombre que el proyecto que testean, pero agregándole *.test* al final del nombre (por ejemplo el proyecto que realiza los test sobre el proyecto *cdBuilder.model* se llama *cdBuilder.model.tests*). En la parte final de este capítulo se describirán en detalle los test automatizados.

3.2.2 Modelo de MVC: Entidades DEVS

GMF fomenta la utilización del patrón MVC para implementar cada elemento que compone los editores gráficos. Para diseñar las entidades DEVS (parte *modelo* en MVC) se utilizaron las herramientas de visualización y generación de código de EMF. Como se mencionó anteriormente, EMF provee un editor que permite crear un modelo utilizando el formato XMI. El modelo completo desarrollado para CD++Builder 2.0 cuenta con 59 clases, que representan cada uno de los elementos de los diferentes modelos DEVS (transiciones, puertos, modelos atómicos, modelos acoplados, variables, estados, expresiones, etc.). En el archivo **DEVS.ecore** del proyecto **cdBuilder.model** puede verse el modelo

completo. Este modelo y las clases que lo implementan se encuentran en un proyecto independiente, para facilitar su reutilización por cualquier otro plugin. A continuación se muestran y detallan las clases y jerarquías más importantes del modelo.

En la Figura 46 se muestra las relaciones de los distintos modelos DEVS y su jerarquía de clases.

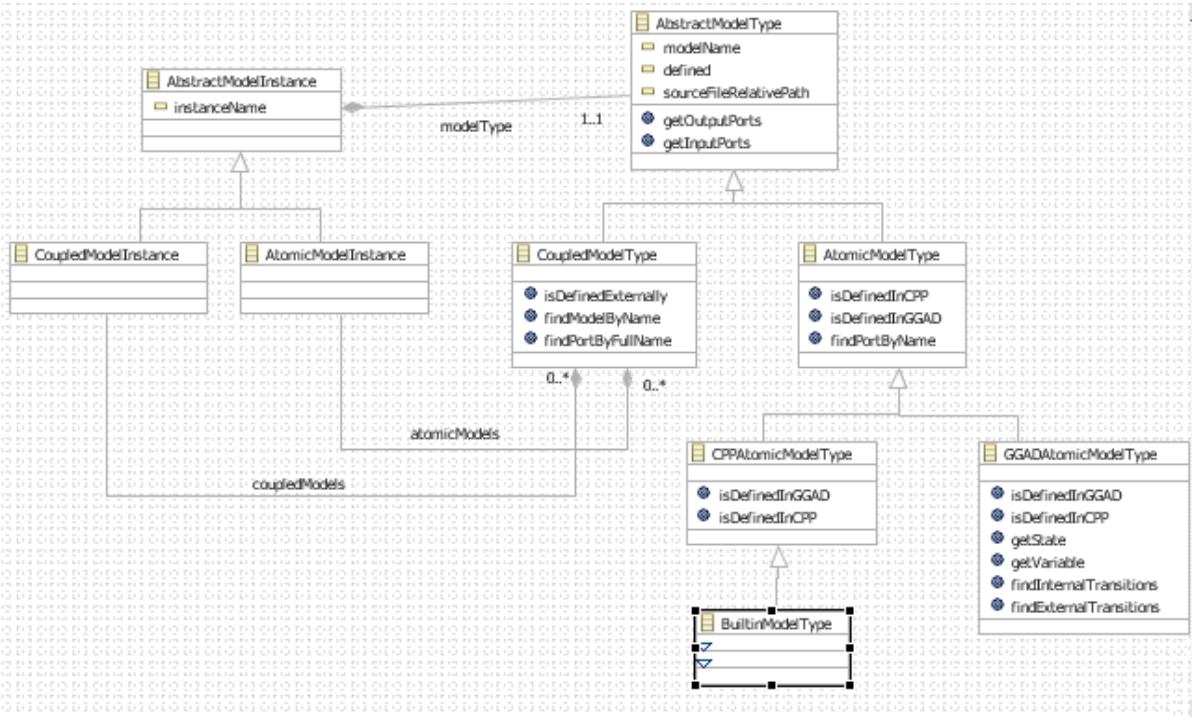


Figura 46 - DEVSModel.ecore, jerarquía de clases y relaciones de modelos DEVS

Como puede verse están representados los modelos atómicos y acoplados, tanto como tipos y como instancias. Todas las instancias tienen uno y sólo un tipo de modelo. Los modelos acoplados están compuestos por 0 o más instancias de modelos acoplados o atómicos, lo que permitirá luego crear varias instancias de un mismo modelo que no era posible en las herramientas anteriores. Por ejemplo, un modelo acoplado puede contener 2 colas (queue1@Queue y queue2@Queue) donde queue1 y queue2 son instancias del mismo tipo Queue.

En el diagrama también pueden verse los diferentes tipos de modelos atómicos. Los modelos atómicos DEVS-Graphs están representados por la clase **GGADAtomicModelType**, mientras que los modelos atómicos definidos en C++ están representados por la clase **CPPAtomicModelType**. Una especialización de los modelos atómicos definidos en C++ son aquellos que ya están incluidos en el simulador CD++. Estos últimos están representados por la clase **BuiltinModelType**.

En la siguiente figura se muestran las relaciones y jerarquías de puertos, links y transiciones:

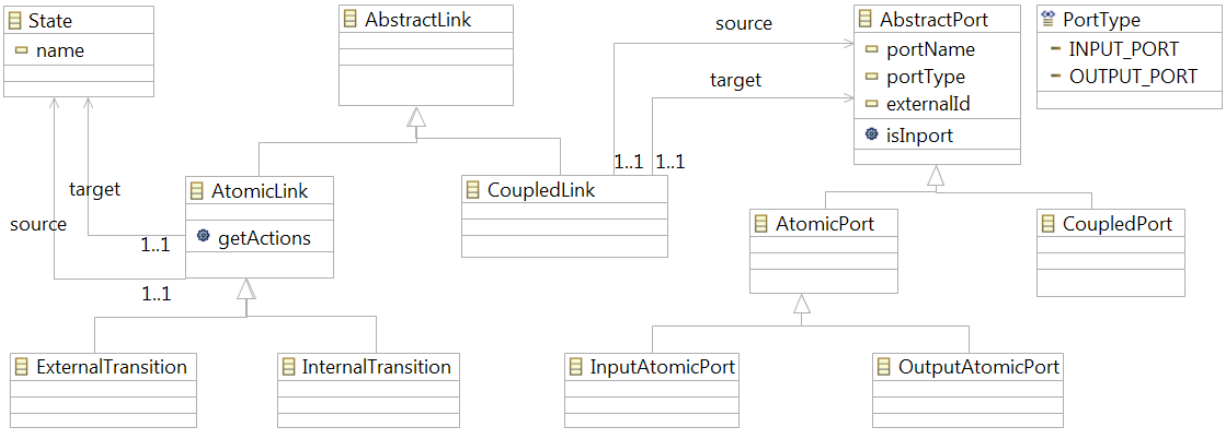


Figura 47 - DEVSModel.ecore. Jerarquía de clases y relaciones de links y puertos

Como se puede observar, se distingue entre puertos y links, siendo estos de modelos atómicos o de modelos acoplados. Los links atómicos a su vez pueden ser transiciones internas o externas, respetando el formalismo DEVS. Un detalle importante a tener en cuenta es que en CD++Builder 2.0 los links de modelos acoplados conectan únicamente puertos entre sí, a diferencia con el modelo que se había propuesto en CD++Modeler donde los links conectan componentes. En CD++Modeler esto dificultó la validación (se pueden crear links entre modelos que no tienen puertos y entre dos puertos de entrada de un mismo modelo) y la creación de links (la creación se hace en dos pasos: primero se crea el link entre los modelos, y luego se seleccionan los puertos). En CD++Builder el modelo no permite conectar componentes, y los controladores validan que los puertos que se conectan sean correctos.

En la siguiente figura puede verse la jerarquía de clases y las relaciones que se utilizan para la representación de los modelos atómicos DEVS-Graphs y atómicos C++. Las clases que heredan de Expression se muestran separadas para simplificar el diagrama, pero forman parte del mismo modelo:

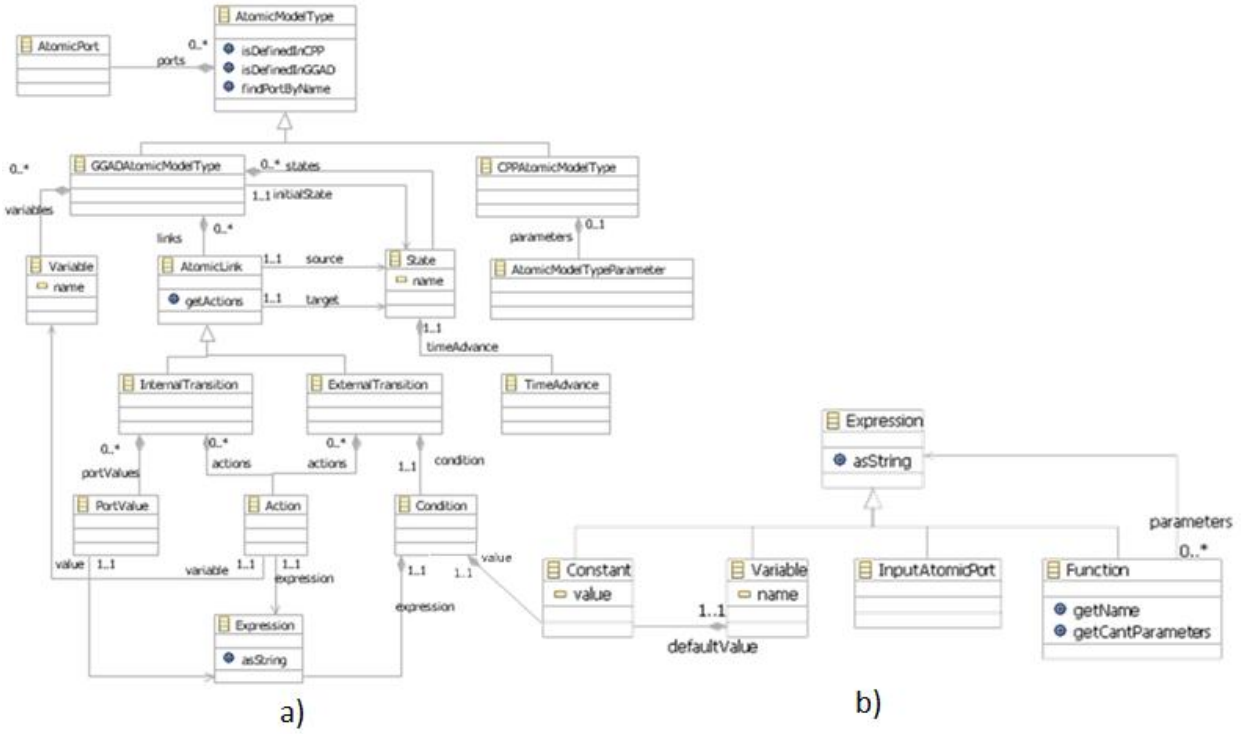


Figura 48 - DEVSModel.ecore. a) Jerarquía de clases de los modelos atómicos b) jerarquía de la clase Expression.

Como puede verse, la jerarquía y composición de los modelos atómicos definidos en C++ es más simple, ya que sólo se distinguen sus puertos y parámetros. Los puertos y parámetros de un modelo C++ son identificados por los editores a través de un parser (descripto más adelante) que interpreta el código C++ que implementa modelo atómico.
La jerarquía de modelos DEVS-Graphs respeta la sintaxis de GGADScript, pudiendo estar compuestos por puertos de entrada y salida, estados, variables, transiciones internas y externas. Tienen también necesariamente un estado inicial. Las transiciones externas poseen una condición de ejecución y acciones a realizar, mientras que las transiciones internas contienen un conjunto de pares puerto-valor, respetando también la sintaxis de GGADScript. Se decidió representar también la estructura de las expresiones (a diferencia de los modelos de CD++Modeler y GGADTool). En la Figura 48 b. se ven todas las entidades que pueden representar expresiones. A través de la composición de funciones se pueden formar expresiones complejas. Este modelo permite construir sólo expresiones válidas, y validar las expresiones dentro de los editores mismos.

Es importante resaltar dos características de los diagramas vistos anteriormente. Por un lado los diagramas son la representación real del modelo ejecutable, ya que a partir de ellos se generan las clases Java. Por otro lado, el modelo no tiene referencia alguna a su representación gráfica. La primer característica facilita la comprensión del modelo y hace muy sencilla su modificación. La segunda característica posibilita a futuros desarrolladores a reutilizar el modelo (y todos los servicios asociados

como persistencia, traducciones, comandos de edición, etc.) para cualquier funcionalidad relacionada con modelos DEVS, sin depender de la implementación de los editores gráficos.

No todas las instancias del modelo presentado son válidas. Por ejemplo, en la Figura 47 puede verse que el modelo permite en principio crear links que posean cualquier par de puertos como origen y destino, que no siempre es válido (por ejemplo un link de un puerto de entrada a otro de entrada del mismo modelo no es válido). Otro ejemplo pueden ser los nombres de los componentes que no deben repetirse dentro de un mismo modelo. Este tipo de validación se realiza tanto en runtime, como al instanciar/modificar los objetos del modelo como en los editores gráficos (mediante los controladores). Al realizar la validación en los controladores, es posible brindar información gráfica al usuario de las operaciones inválidas (cambiando la imagen del puntero del mouse), y no permitir crear o modificar el modelo incorrectamente (esto puede verse por ejemplo durante la creación links). Las validaciones se implementaron utilizando la infraestructura provista por EMF Validation, y en algunos casos usando código personalizado.

3.2.3 Vistas y Controladores de MVC: Editores Gráficos

El framework GMF permite partir de un modelo y una configuración para luego generar la infraestructura de un plugin que implementa un editor gráfico para dicho modelo. Para personalizar el código generado, las herramientas de GMF permiten colocar comentarios especiales para evitar perder los cambios al regenerar el código. En esta sección nos concentraremos en el código personalizado, ya que el código generado puede encontrarse documentado en el sitio web de GMF [EGMF10].

En el caso de CD++Builder 2.0 se utilizó el *modelo* descripto anteriormente y luego se realizaron modificaciones iterativas sobre las configuraciones para lograr la *vista* deseada. La configuración utilizada para CD++Builder 2.0 y puede consultarse en el proyecto **cdBuilder.model**. El generador de código de GMF genera editores bastante completos, pero no es lo suficientemente poderoso para adaptarse a todos los requerimientos de editores de modelos DEVS. Por ello, fue necesario realizar varias personalizaciones, entre las cuales se distinguen:

1. **Editores con múltiples páginas.** Se utilizan para mostrar dentro del mismo editor una página para el modelo gráfico y otra el modelo textual (ver Figura 59 de la sección "3.3 Entorno CD++Builder 2.0").
2. **Mecanismo de sincronización** entre ambas páginas para mantener el modelo gráfico coherente con el modelo textual.
3. **Traductores** para los diferentes formatos de persistencia de los modelos. Como se vió antes la persistencia del modelo gráfico es implementada por EMF, pero fue necesario desarrollar traducciones y parsers para los formatos de CD++ (archivos con extensión *.cdd* y *.ma*), para los modelos atómicos C++ y para los formatos de CD++Modeler (archivos con extensión *.gcm* y *.gam*).
4. **Visualización personalizada** de los puertos. Fue necesario re implementar la vista generada por GMF para distinguir los puertos que poseen los submodelos y posicionarlos adecuadamente.

5. **Navegación jerárquica** de modelos. Se re implementó la acción que se ejecuta al hacer doble clic sobre los modelos. En el caso de modelos acoplados se abrirá un nuevo editor, en el caso de modelos atómicos se créanlos archivos necesarios (editores en el caso de DEVS-Graphs y código C++ en el otro caso).
6. **Extensión del panel de herramientas** para incluir modelos CD++. El panel de herramientas carga dinámicamente los modelos atómicos declarados en el register.cpp de CD++ para poder ser reutilizados al modelar. Al arrastrarlos al panel de edición, crea una nueva instancia del tipo de modelo seleccionado.

Para la implementación de los editores con múltiples páginas, se modificaron las clases generadas por GMF para que los editores extiendan la clase **MultiPageEditorPart** provista por Eclipse. La página gráfica utiliza el editor generado por GMF mientras que la página con el modelo textual reutiliza el editor de CD++Builder que identifica las palabras clave. Estos editores también fueron modificados para actuar como controlador, ejecutando el mecanismo de sincronización entre ambas páginas al realizarse cambios.

Los editores gráficos fueron extendidos para permitir dos tipos de sincronización: automática y manual. El tipo de sincronización a utilizar se selecciona de un menú contextual que se hace visible al hacer clic derecho sobre el editor (ver Figura 49).

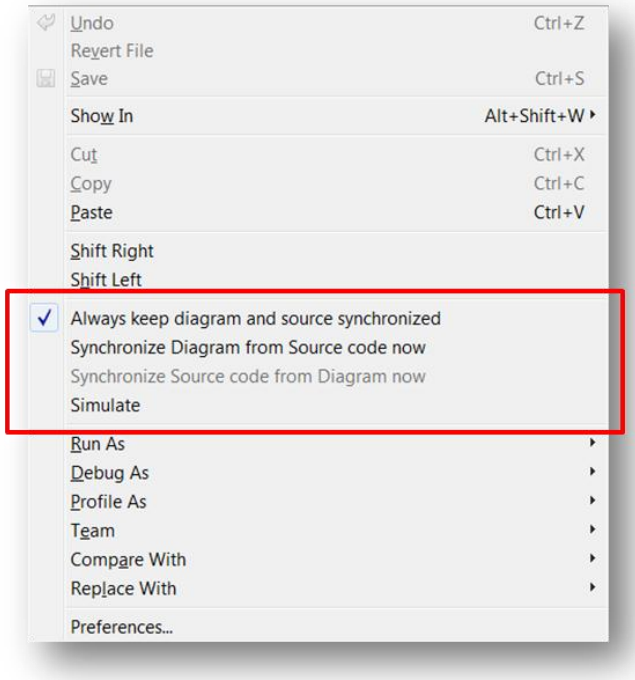


Figura 49 - Menú contextual para seleccionar el modo de sincronización

En caso que la actualización sea automática, el mecanismo de sincronización actualiza el modelo gráfico y el modelo textual en base a la página que esta activa al guardar los cambios (o sea, toma como modelo válido el que estaba visible al guardar). En caso que se haya guardado el modelo textual, es siempre necesario sincronizarlo con el modelo gráfico regenerando el diagrama. Al regenerar el diagrama a partir de la especificación textual del modelo, se pierde la información de las figuras (posiciones, tamaños, etc.) por la misma razón anterior (CD++ no guarda información visual del modelo).

Como se vio, el proceso de sincronización genera partir de la especificación de un modelo, el mismo modelo en otro formato (por ejemplo, a partir de la representación gráfica genera el modelo utilizando la gramática de CD++). Estos procesos fueron implementados mediante traductores. No solo la sincronización utiliza este tipo de traducciones, sino que se implementaron traductores para reconocer puertos y parámetros en el código C++ de los modelos atómicos, para identificar los modelos registrados en archivos register.cpp y para generar el formato CD++Modeler necesario para las animaciones. En CD++Builder se desarrollaron los siguientes traductores:

1. **DEVSGGADAtomicModelSimpleTranslator.** Implementa la traducción de la gramática GGADScript al modelo conceptual (representado por la clase GGADAtomicModelType) y viceversa.
2. **FromCPPAtomicModelToCDModelerTranslator.** Implementa la traducción del modelo conceptual de CPPAtomicModelType al formato de modelos atómicos de CD++Modeler.
3. **FromGGADAtomicModelToCDModelerTranslator.** Implementa la traducción del modelo conceptual de GGADAtomicModelType al formato de modelos atómicos de CD++Modeler.
4. **DEVSCoupledModelSimpleTranslator.** Implementa la traducción de la gramática de modelos acoplados de CD++ al modelo conceptual (representado por la clase **CoupledModelType**) y viceversa.
5. **FromCoupledModelToCDModelerTranslator.** Implementa la traducción del modelo conceptual de CoupledModelType al formato de modelos acoplados de CD++Modeler.

Estos traductores fueron desarrollados desde cero, basándose en algunos casos en la estructura de los parsers de CD++Modeler preexistentes. En particular, el parser que interpreta la gramática GGADScript está desarrollado utilizando JFlex y JCup. Esta decisión fue tomada ya que la implementación de DEVS-Graphs en CD++ utiliza Flex y Bison para interpretar los archivos GGADScript y ya se contaba con la especificación de la gramática. Por lo tanto el parser de DEVS-Graphs en CD++Builder 2.0 está basado en el parser de DEVS-Graphs de CD++.

En cuanto a los parsers de modelos atómicos C++, no reconocen la sintaxis completa de C++, sino que buscan características específicas dentro del código. Por ejemplo para identificar los puertos se buscan las líneas de código que contengan llamadas al método **addOutputPort** o **addInputPort** y para identificar los parámetros se buscan llamadas al método **getParameter**. Esta decisión se tomó por un lado porque no se encontró un parser de la sintaxis completa de C++ implementado en Java y por el otro porque no era necesario reconocer todo el código sino sólo extraer algunas características (puertos, parámetros y nombre del modelo). De cualquier manera, esta decisión implica que si bien el parser

funciona bien cuando se utiliza la sintaxis recomendada, no se adapta a todos los casos posibles. Un simple ejemplo donde el parser no funciona consistentemente es si la lectura de los parámetros se encapsula en una función y la función nunca es ejecutada (en vez de hacerse directamente como es habitual). En este caso, el parser reconoce la lectura del parámetro por lo que el modelo gráfico siempre aceptará este parametro (se ejecute o no el código de lectura del parámetro).

```
string getParameter(const string &parameterName ) const
{
   // If this method is never called, the parameter will appear in the editor anyway
   return MainSimulator::Instance().getParameter( description(),parameterName);
}
```

Figura 50 - Ejemplo donde el parser C++ puede funcionar inconsistentemente.

En la práctica esta limitación no pareció ser muy relevante, ya que el parser funcionó de manera correcta para todos los modelos preexistentes utilizados en los tests.

3.2.4 Actualizaciones

Para el desarrollo de las actualizaciones automáticas se crearon dos proyectos que implementan esta funcionalidad. Por un lado el proyecto **cdBuilder.installation.features** encapsula todos los plugins relacionados a CD++Builder 2.0 en un único feature instalable. Por otro lado el proyecto **cdBuilder.installation.updateSite** contiene toda la información necesaria para publicar el feature y los plugins de CD++Builder 2.0 en un sitio de actualizaciones.

En cuanto a los detalles de implementación de los proyectos de actualizaciones se desarrollaron en su generalidad siguiendo las prácticas recomendadas por Eclipse [ECL10]. Se realizaron mejoras iterativas en la configuración de los proyectos para minimizar la dependencia con plugins externos que incrementen el tiempo de instalación. Es importante notar que el proyecto ***cdBuilder.installation.updateSite*** debe estar alojado en un servidor accesible por todos los usuarios para que pueda utilizarse. Con este fin, se creó un sitio utilizando la infraestructura que provee SourceForge (www.sourceforge.net). Se decidió utilizar este servicio ya que el proyecto de CD++ está también publicado allí, es gratuito y provee buen soporte. Una desventaja es que el setup no es sencillo. En **Apéndice C: Manual de desarrollo de CD++Builder** se encuentran detallados los pasos necesarios para generar una nueva versión del plugin y modificar el sitio de actualizaciones. Dado que para instalar el plugin se requieren descargar aproximadamente 10MB y estos sitios no están pensados para soportar grandes cantidades de descargas, se hicieron pruebas para verificar la velocidad del sitio. Se comprobó que si bien la velocidad de descarga no es excelente, es aceptable para la instalación de CD++Builder (se completa en aproximadamente 4-5 minutos utilizando una conexión de Internet de 1MB).

3.2.5 Tests automatizados

Para el desarrollo de los tests automatizados se creó un proyecto de test por cada uno de los nuevos features implementados. En total CD++Builder incluye los siguientes proyectos de tests:

1. **cdBuilder.tests**. Incluye test unitarios y funcionales para las acciones que se ejecutan al utilizar los botones de la barra de herramientas de CD++Builder. En particular los tests se centran en las acciones de animación que son las que fueron modificadas en este trabajo.
2. **DEVSModeler.atomic.diagram.tests.** Incluye test unitarios y funcionales que verifican las funcionalidades del editor de modelos DEVS-Graphs.
3. **DEVSModeler.diagram.tests.** Incluye test unitarios y funcionales que verifican las funcionalidades del editor de modelos acoplados DEVS.
4. **DEVSModeler.tests.** Incluye test unitarios que verifican el comportamiento de las entidades. Este es autogenerado por EMF.

En el caso de los tests para los editores gráficos, se centran en verificar el funcionamiento de los features personalizados. No es necesario verificar el funcionamiento del código generado automáticamente dado que ya está garantizado por GMF. De cualquier manera sería interesante que GMF provea la posibilidad de generar tests, para corroborar que las personalizaciones que se realizan en el código no corrompan el funcionamiento original. Lamentablemente esto aún no es posible en la versión actual de GMF.

El comportamiento de los traductores depende en gran medida del input (el archivo a parsear o el modelo a escribir), por lo que realizar únicamente tests unitarios sería insuficiente. Es por esto que se decidió implementar tests funcionales para cada uno de los traductores.

Los test funcionales para traductores toman como entrada modelos ya desarrollados y conocidos, que pueden descargarse del repositorio de modelos (http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples). En total se verifican las traducciones utilizando los siguientes modelos:

1. Alternate bit protocol: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/alternatebitprot.zip
2. ATM: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/atm.zip
3. Auto: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/auto.zip
4. Bridge: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/bridge.zip
5. Bus Vending Machine: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/BusVending.zip
6. Clock: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/clock.zip
7. Clock2: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/clock_2.zip
8. Cruise Control System: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/cruisecontrolsystem.zip
9. Elevator: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/Elevator.zip
10. Vending: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/Vending.zip
11. Virtual Pet: http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/virtualpet.zip

También se utilizaron los siguientes modelos para testear los traductores de modelos DEVS-Graphs:

1. Automatic Queue.
2. Alternate bit protocol.
3. Queueing System.
4. Tic-Toc 1.
5. Tic-Toc2.
6. Timer.

Tomando como entrada cada uno de estos modelos los test verifican primero que el traductor lea correctamente los archivos. O sea que los archivos sean parseados sin generar errores y que el modelo conceptual que se interpretó a partir de la sintaxis del archivo es correcto. Luego se verifica que el traductor escriba correctamente el modelo conceptual en el formato de CD++. O sea, que a partir del modelo conceptual se utiliza el traductor para generar nuevamente un archivo con la sintaxis de CD++ y luego se verifica que el archivo original y el nuevo generado sean equivalentes.

La siguiente figura muestra la ejecución de todos los tests de JUnit para CD++Builder 2.0:

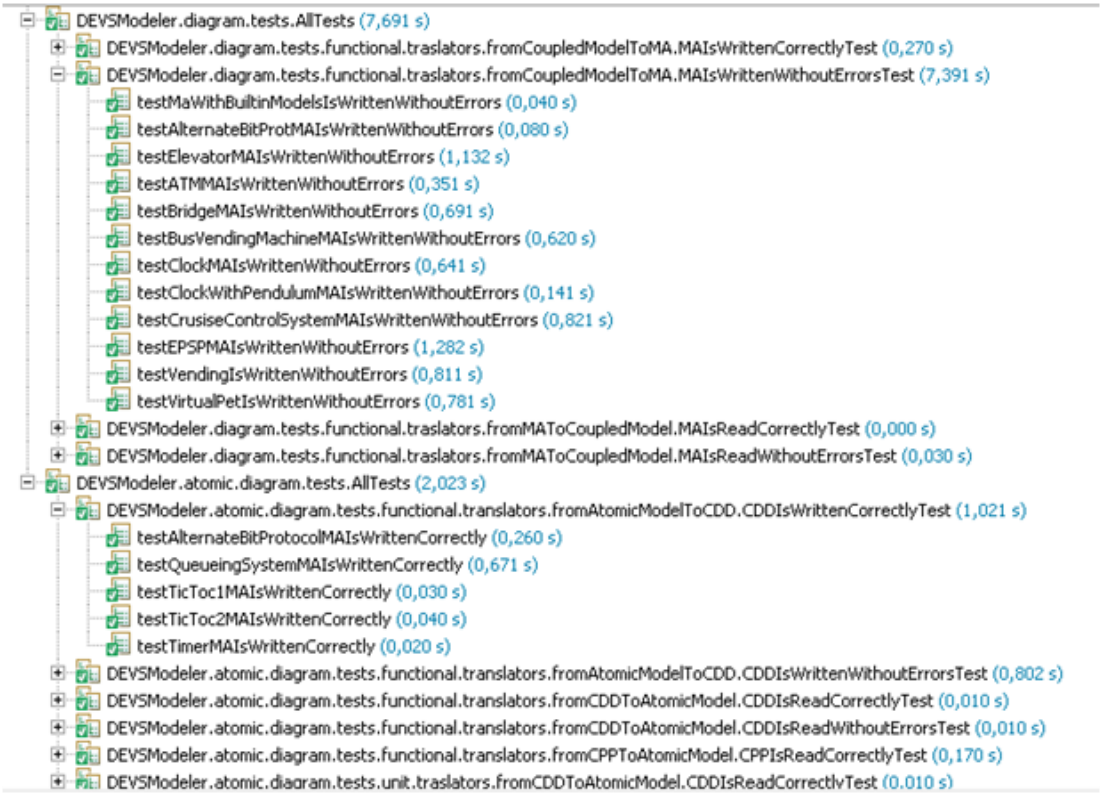


Figura 51 - Ejecución de los tests de traductores en CD++Builder 2.0

Para realizar tests sobre proyectos y funcionalidades que están implementadas en plugins de Eclipse, es necesario ejecutar una nueva instancia de Eclipse, con los plugins instalados especialmente para realizar

tests en ese ambiente. Es por eso que cuando se ejecutan los tests, aparecerá una nueva ventana de Eclipse donde se ejecutarán automáticamente diferentes comandos. JUnit provee la infraestructura y las librerías para realizar esta tarea; sin embargo, fue necesario realizar modificaciones menores que permitan incluir un proyecto con los recursos necesarios para realizar tests funcionales en la nueva instancia de Eclipse.

3.3 Entorno CD++Builder 2.0

CD++Builder 2.0 es un plugin para Eclipse que integra todas las características y funcionalidades disponibles para el simulador CD++, facilitando la creación y simulación de modelos DEVS. La incorporación de todas las características en el entorno de desarrollo Eclipse permitió mejorar y potenciar funcionalidades preexistentes basándolas en la infraestructura de Eclipse.

La siguiente figura muestra el entorno de simulación CD++Builder.

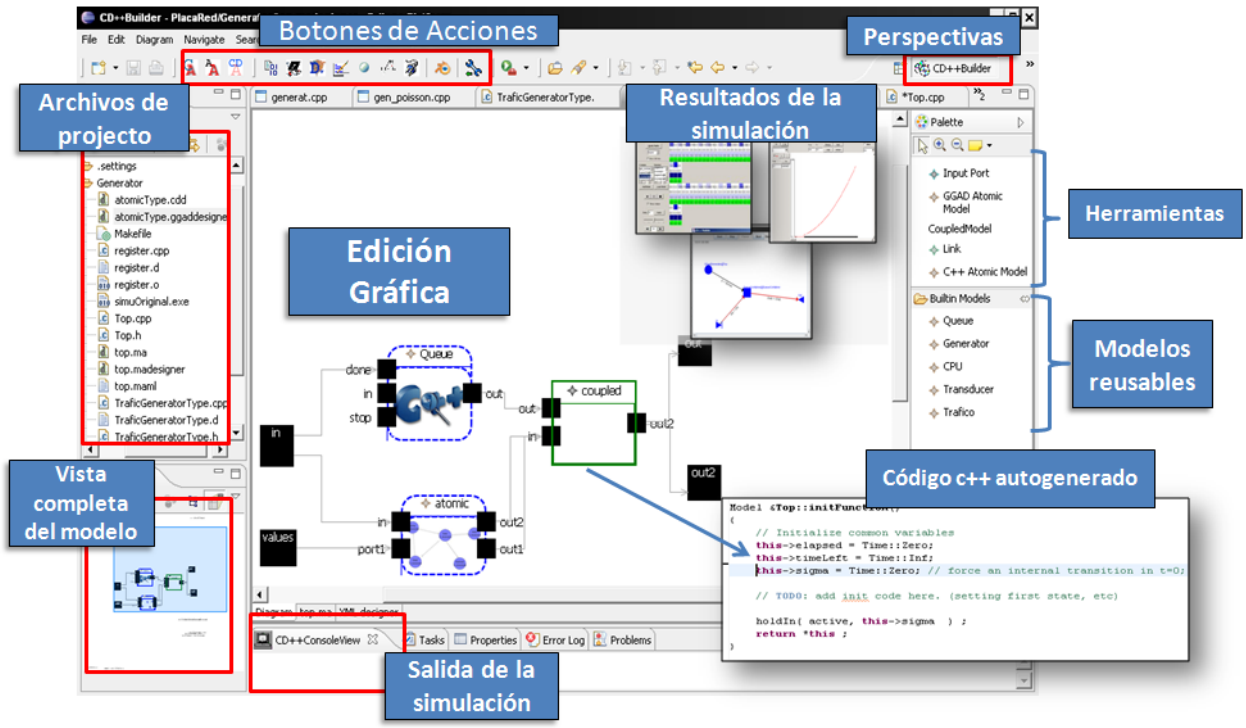


Figura 52 - Entorno de simulación CD++Builder.

Los *botones de acción* en la parte superior permiten la ejecución de herramientas externas. En la parte izquierda, una sección especial está dedicada a las animaciones de CD++Modeler que permiten animar los resultados de simulaciones de modelos DEVS atómicos, DEVS acoplados y Cell-DEVS. También incluyen botones para ejecutar las herramientas CD++Modeler, GGADTool y Drawlog. El botón de *Build* crea un archivo makefile y compila el código fuente de todos los modelos atómicos en el proyecto, generando un archivo ejecutable para la simulación. El botón *Simulate* muestra un

formulario que permite especificar los parámetros necesarios para correr una simulación. Para crear nuevos modelos o proyectos es posible utilizar los menús para nuevos archivos de Eclipse (Figura 53), que serán agregados al árbol de navegación junto al resto de los archivos del proyecto. La edición de modelos también se realiza a través de la interfaz de Eclipse utilizando editores gráficos. Eclipse permite reorganizar las ventanas para personalizar la interfaz y las Perspectivas de CD++Builder se pueden utilizar para mostrar los botones y paneles específicos de distintos escenarios. Los paneles inferiores CD++ConsoleView y ErrorLog son utilizados para mostrar los resultados de la compilación y los errores que se producen en los editores, respectivamente.

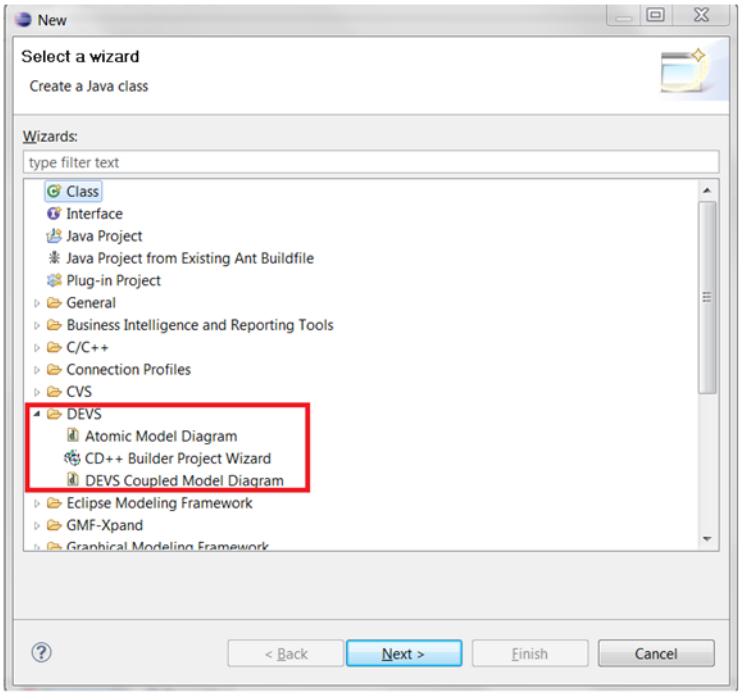


Figura 53 - Creación de nuevos modelos y proyectos.

El editor de modelos acoplados (Figura 54 y Figura 55) permite definir modelos atómicos C++ y genera la estructura de código necesaria para extender la clase *Atomic* de CD++. La integración con *Eclipse C/C++ Development Tools* permite una mejor visualización y manejo de los archivos C++. El editor de modelos acoplados también permite la definición de modelos DEVS-Graphs, para los cuales CD++Builder provee otro editor gráfico (Figura 56).

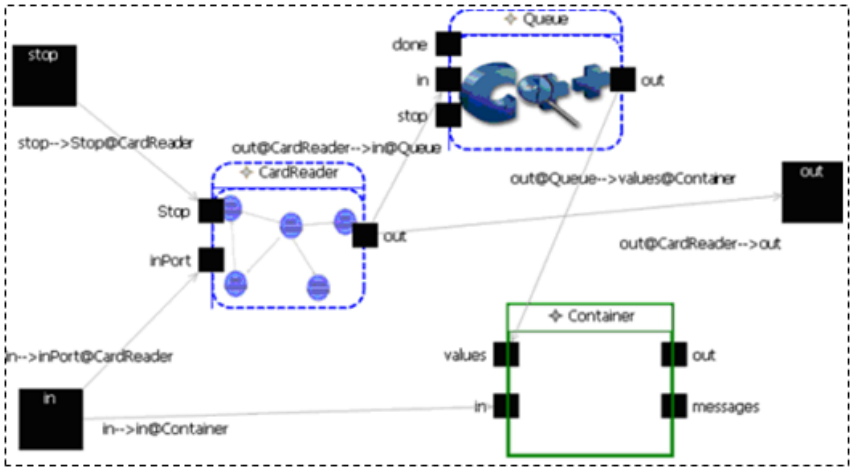


Figura 54 - Editor de modelos acoplados

El panel *Properties* de Eclipse se utiliza para mostrar y editar las propiedades de las diferentes entidades y el panel *Outline* muestra una visión completa del modelo. Cada editor se muestra en una solapa separada (permitiendo tener varios modelos abiertos al mismo tiempo), y los submodelos (modelos contenidos en otros modelos acoplados) se mantienen consistentemente relacionados.

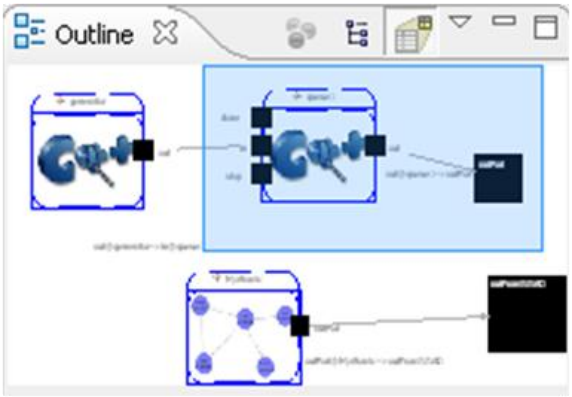


Figura 55 - Overview de modelos en CD++Builder 2.0.

Otras características que facilitan el modelado gráfico fueron incorporadas a ambos editores: posibilidad de acercar/alejar la vista del modelo, comandos para copiar y pegar entidades, configuración de tipo y tamaños de las letras y diferentes estilos para las conexiones. Tanto el editor para modelos acoplados como el editor de modelos atómicos proveen paneles especiales con herramientas para crear fácilmente las entidades disponibles (modelos atómicos/acoplados, links entre modelos, puertos, transiciones, estados, variables, etc.). En particular el editor de modelos acoplados también incluye en este panel una sección con otros modelos acoplados que pueden incluirse y reutilizarse para componer nuevos modelos acoplados.

En el editor de modelos acoplados, los modelos están gráficamente representados por rectángulos que contienen el nombre del modelo. Los modelos atómicos y acoplados que componen al modelo principal

se diferencian por color (azules y verdes, respectivamente) y por su forma (rectángulos y rectángulos con puntas ovaladas, respectivamente). Los modelos atómicos DEVS-Graphs y C++ a su vez se distinguen entre sí por la imagen que se encuentra en su interior. Los puertos se muestran como cajas negras con el nombre del puerto y las flechas representan las conexiones entre los diferentes puertos de los modelos.

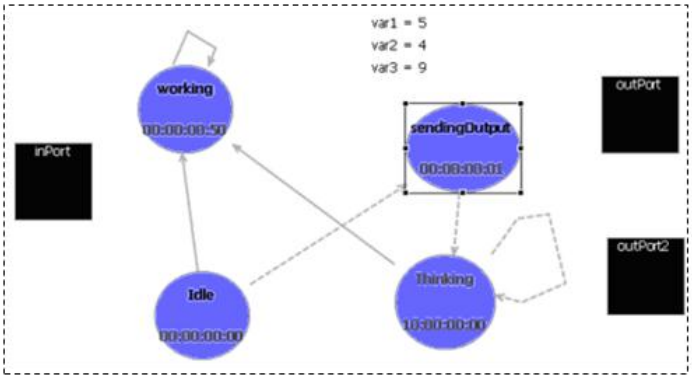


Figura 56 - Editor de modelos atómicos DEVS-Graphs.

El editor de modelos atómicos (Figura 56) utiliza la notación DEVS-Graphs para representar los modelos: los estados del modelo atómico se muestran como círculos con su identificador y el valor del *time advance.* Las transiciones internas están representadas por flechas punteadas y las externas por flechas completas que unen los estados origen y destino.

Para desarrollar nuevos modelos atómicos C++ en CD++ es necesario crear una nueva clase que herede de la clase abstracta *Atomic*, y modificar el archivo *register.cpp* para registrar el modelo en el framework. Para simplificar y agilizar la definición de estos modelos, se agregó la capacidad de generar código C++ al nuevo editor de modelos acoplados. Al hacer doble clic sobre un nuevo modelo atómico C++, utilizando el nombre del modelo y templates de código predefinidos (Figura 57), se generan los archivos necesarios (de cabecera y código C++) y se actualiza el archivo *register.cpp*. Los templates proveen la estructura de código para extender la clase *Atomic* y contienen comentarios de ayuda con ejemplos de código. Todos los métodos que se encuentran implementados (*initFunction*, *externalFunction*, *internalFunction* and *outputFunction*) poseen un breve comentario explicando su uso y función, que facilita la implementación a nuevos usuarios de DEVS y C++. Los comentarios también muestran ejemplos de cómo agregar puertos de entrada/salida y diferentes parámetros al modelo.

Al desarrollar modelos atómicos directamente en C++, la representación gráfica del modelo se mantiene consistente con el código C++ por medio de un nuevo parser. Cuando los archivos C++ son modificados y grabados, el parser reconoce estructuras de código específicas para identificar el nombre del modelo, puertos de entrada/salida y los parámetros (valores constantes para configurar los modelos atómicos). De esta manera la representación gráfica del modelo y el código están siempre sincronizados, sin imponer restricciones sobre el código.

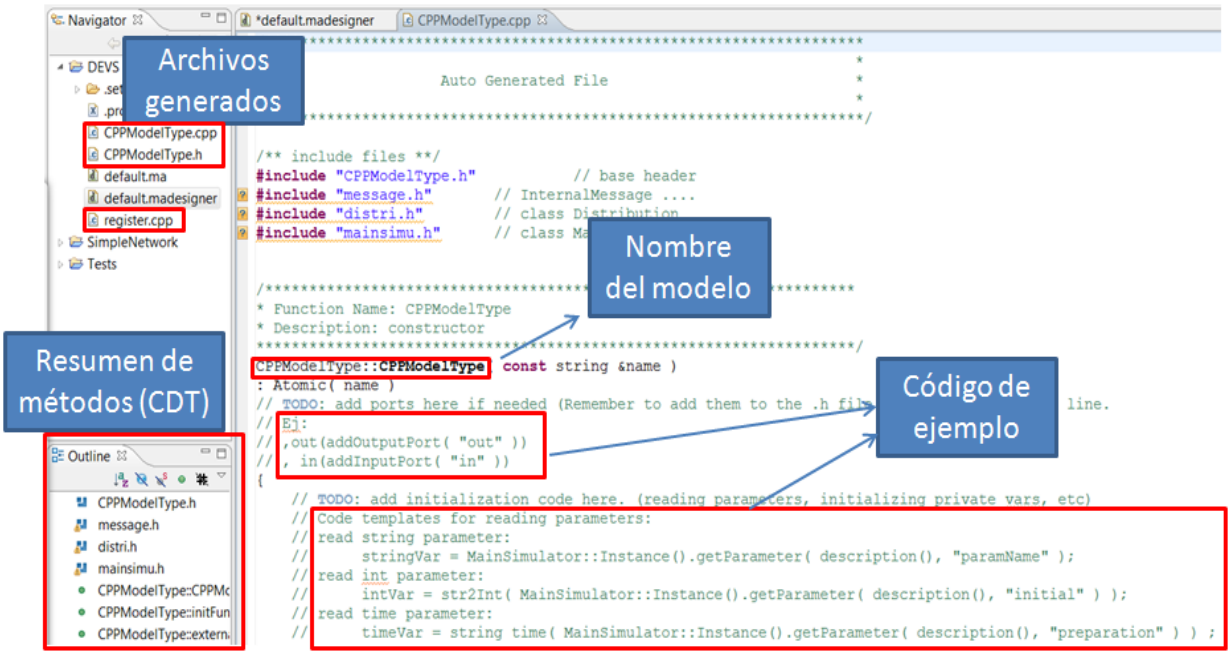


Figura 57 - Código generado para los modelos atómicos C++.

Los nuevos editores fueron adaptados para reusar las animaciones disponibles anteriormente en CD++Modeler, ya que en el pasado han sido utilizadas satisfactoriamente para visualizar los resultados y la evolución de las simulaciones. Se proveen controles para manejar el avance del tiempo y las conexiones se colorean dinámicamente para representar mensajes que van de un modelo al otro. Para los modelos acoplados se utiliza una representación en bloque, mientras que para los modelos atómicos las trayectorias entrada/salida se muestran en diferentes puertos a lo largo del tiempo

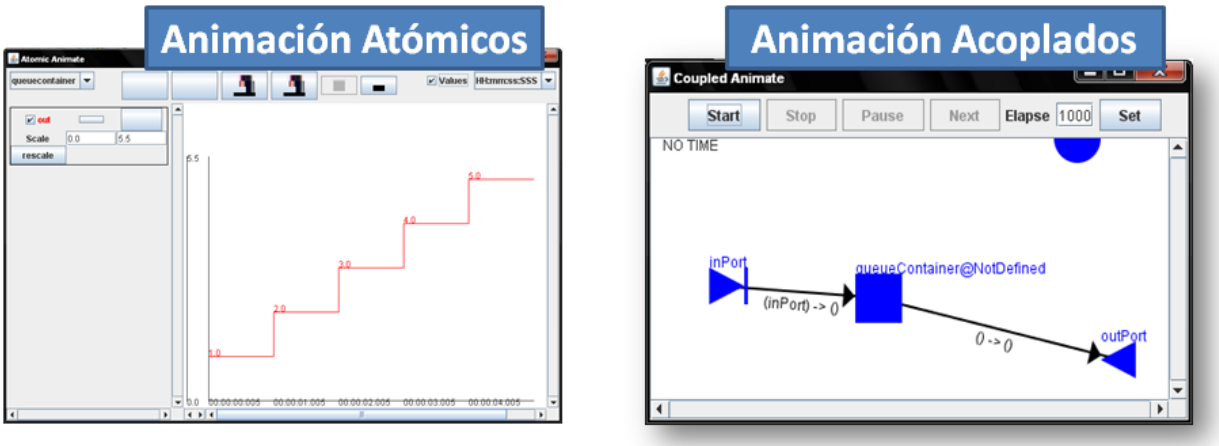


Figura 58 - Animaciones para modelos atómicos y acoplados.

En CD++Builder, la información gráfica se guarda en formato XMI y la persistencia de y hacia este formato es manejada por los servicios provistos por el framework EMF. Se han desarrollado nuevos

parsers para implementar traductores, de la gramática de CD++ a su representación gráfica y vice versa. De esta manera, los nuevos editores de modelos acoplados y modelos atómicos DEVS-Graphs pueden mostrar ambas vistas de los modelos (la representación gráfica y la textual en formato CD++). Para cambiar de una vista a la otra se utilizan las solapas en la parte inferior de los editores (Figura 59). Cuando cualquiera de las vistas es actualizada y guardada, ambos archivos son sincronizados utilizando los traductores para mantenerlos consistentes.

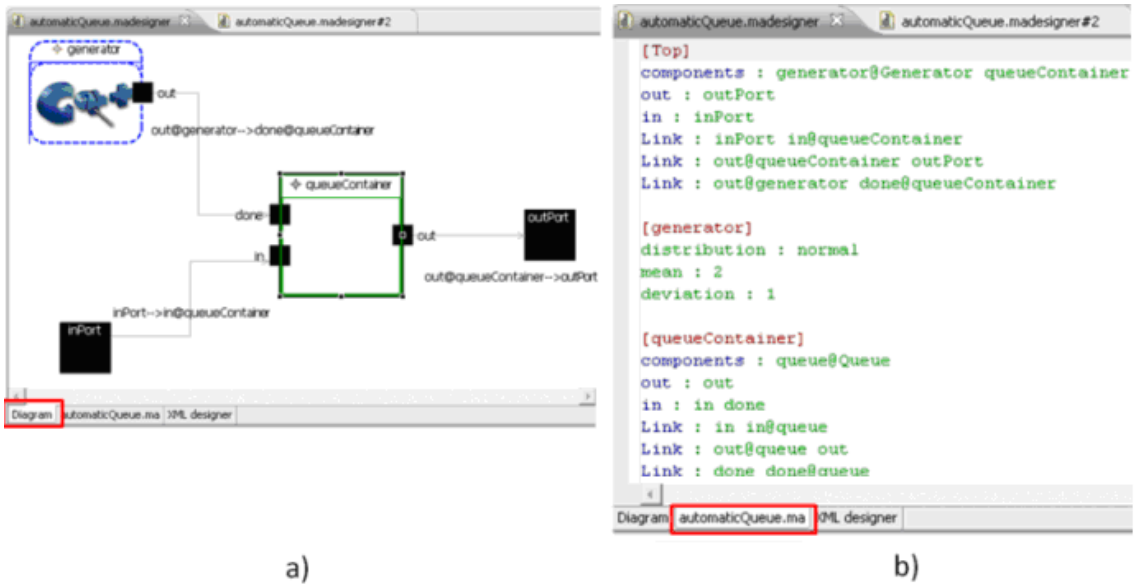


Figura 59 - Vista a) gráfica b) textual de un mismo modelo.

Si bien los archivos gráficos contienen toda la información para reescribir por completo la definición en CD++ del modelo, lo opuesto no es posible. Por lo tanto, cuando el archivo en formato textual es grabado, el diagrama puede ser actualizado consistentemente, pero se pierde toda la información gráfica. De la misma manera, al abrir modelos que no fueron construidos utilizando los editores gráficos, los traductores pueden generar una nueva representación gráfica a partir de la definición textual, solucionando la limitación de los editores anteriores. En estos casos los valores predefinidos son utilizados para la información gráfica faltante.

La posibilidad de visualizar todos los modelos (incluso los que no fueron desarrollados gráficamente) ayuda a una mejor comprension de los modelos, facilitando la reutilización de modelos entre la comunidad. En este sentido, el editor de modelos acoplados posee un panel (Figura 60) en donde se incluye una sección con modelos que pueden ser reutilizados. Esto le permite a los usuarios que no están familiarizados con la librería de modelos de CD++ saber qué modelos ya fueron creados previamente. Para componer el modelo que se está editando, el editor permite arrastrar del panel los modelos que se desean utilizar.

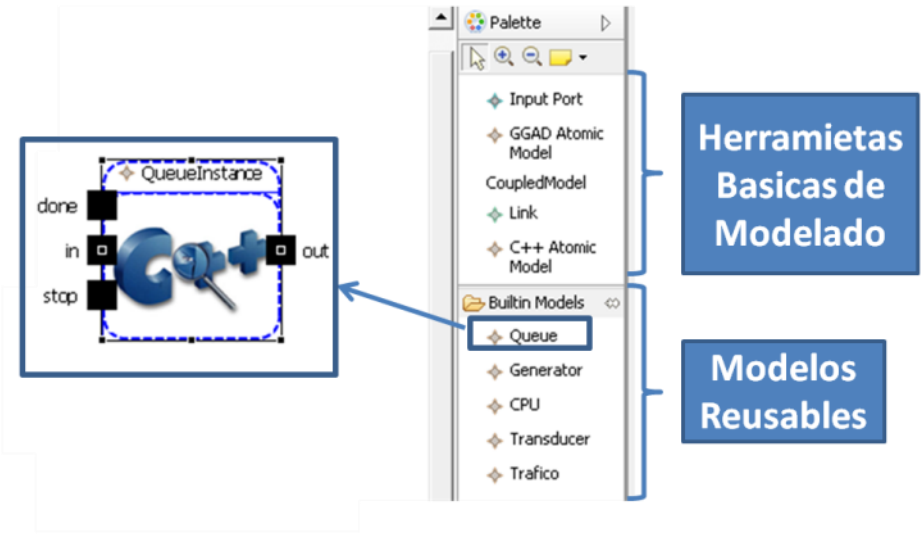


Figura 60 - Panel con modelos reutilizables.

Las instalación de CD++Builder en las versiones anteriores se realizaba descargando una versión completa de Eclipse y otras herramientas. Las instalación y actualización de las nuevas versiones de CD++Builder utiliza un esquema centralizado, integrado con el sistema de actualización *Eclipse Update Manager* provisto por la infraestructura de Eclipse. Este sistema permite colocar el código compilado del plugin y su metadata en un único sitio web de publicación, al que todos los usuarios pueden acceder para realizar una nueva instalación o actualizaciones periódicas. La siguiente figura muestra esta arquitectura:

Figura 61 - Esquema centralizado de instalación y actualizaciones.

Este esquema permite una instalación más sencilla, asistida por formularios estándar de instancias ya instaladas de Eclipse. Es decir, que no es necesario descargar la plataforma completa de Eclipse para instalar el plugin de CD++Builder, permitiendo que este corra junto a otros plugins. Más importante aún, este sistema resuelve los problemas de versionado: las soluciones a bugs y las últimas funcionalidades incorporadas no tienen que ser distribuidas a cada usuario individualmente. Por el contrario, las actualizaciones a CD++Builder son subidas a un punto centralizado al que luego los usuarios consultarán. La integración con *Eclipse Update Manager* permite a los clientes lanzar los chequeos de actualizaciones manualmente o configurar actualizaciones periódicas automáticas. En el **Apéndice A: Manual de Instalación de CD++Builder 2.0** se encuentran detallados los pasos necesarios para instalar y actualizar CD++Builder 2.0.

Para el desarrollo de los componentes más importantes de CD++Builder 2.0 se utilizó la técnica de desarrollo *Test Driven Development* (TDD) [Bec03]. Los test automatizados provistos ayudan a mejorar la calidad del software desarrollado y facilitan la extensibilidad de la plataforma. Los test proveen un nivel más alto de seguridad que una cierta funcionalidad esta correctamente implementada. Por otro lado, al poder ser ejecutados automáticamente en cualquier momento, pueden ser utilizados para verificar que una refactorización del código o una nueva funcionalidad no provoquen un mal funcionamiento en el

resto del sistema. Para el desarrollo de estos tests se ha utilizado el framework JUnit, ya que provee soporte para testear plugins de Eclipse.

CD++Builder cuenta con 138 tests automatizados, que pueden ejecutarse para verificar que el plugin se comporta de la forma esperada. Al correr los test, se ejecuta una instancia especial de la interfaz de Eclipse ya que en muchos casos es necesaria la infraestructura de Eclipse para la ejecución de los tests. En la siguiente figure puede verse la ejecución de todos los tests de CD++Builder, junto con sus tiempos de ejecución.

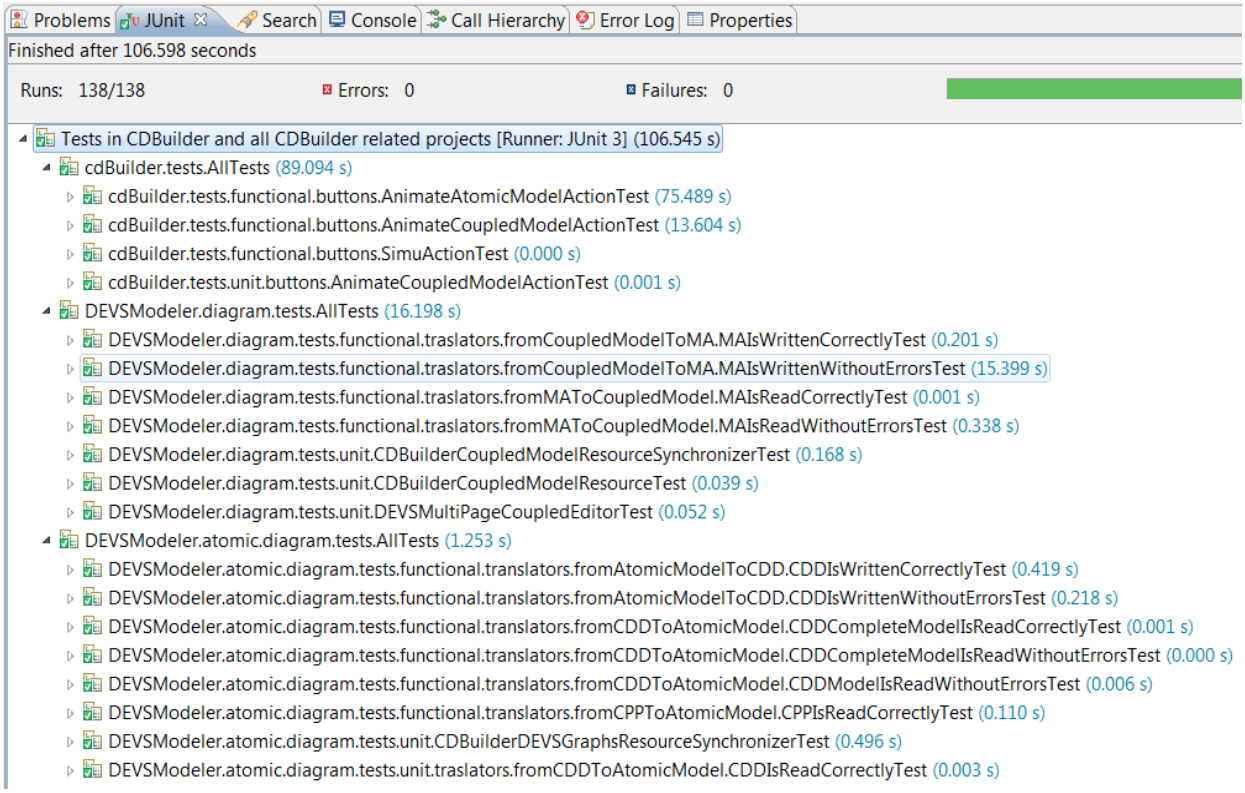


Figura 62 - Ejecución de todos los tests de CD++Builder 2.0.

Capítulo 4 - Resultados

4.1 Proceso completo de M&S con CD++Builder 2.0

En esta sección se describirán los pasos necesarios para obtener un modelo completo, simularlo y analizar los resultados utilizando el entorno CD++Builder 2.0. A lo largo de este proceso destacaremos los beneficios que brinda la herramienta en el proceso de M&S de modelos DEVS para CD++ y así poder contrastarlas con las herramientas anteriores.

Crearemos un modelo muy simple de una red, que genere un determinado retraso en los paquetes y que aleatoriamente pierda algunos paquetes. Comenzaremos creando un modelo que genera valores aleatorios utilizando el generador que ya provee CD++ para poder ejecutar la primera simulación. Este generador representará el tráfico de paquetes entrante a la red. Luego iremos modificando el modelo iterativamente, componiéndolo con nuevos modelos atómicos (primero un modelo DEVS-Graphs, luego desarrollaremos uno en C++) hasta obtener el modelo final. Finalmente veremos los resultados de la simulación del modelo final. En el **Apéndice B: Manual de usuario de CD++Builder** se encuentran los pasos detallados para crear cada elemento, editor, etc. y donde se muestra la creación del modelo de una cola.

Como primer paso, es necesario crear un nuevo proyecto en Eclipse y su modelo acoplado global (que llamaremos *simpleNetwork)*. Para esto se pueden utilizar los asistentes que se encuentran dentro de la categoría *DEVS* como muestra la siguiente figura:

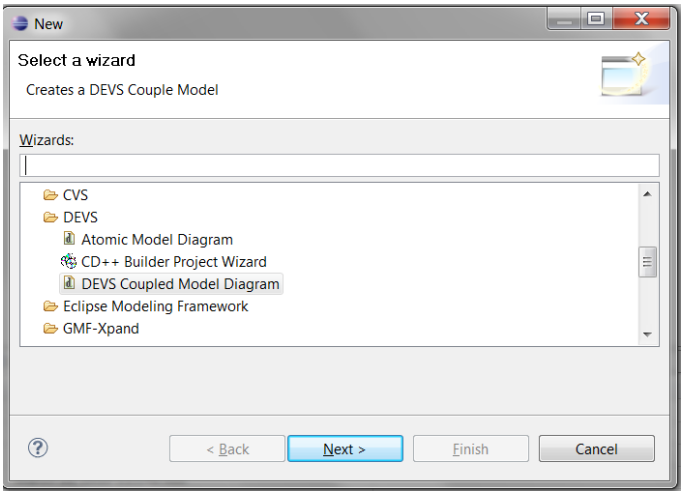


Figura 62 - Creación del proyecto en Eclipse y modelo acoplado.

Para poder ejecutar la primer simulación rápidamente, crearemos un modelo acoplado simple compuesto por un generador de paquetes que envíe su salida al puerto como muestra la Figura 64 a.

Para crear los elementos del modelo (en este ejemplo un puerto de salida, un generador y un link que los conecte), se utilizan los iconos del panel de herramientas que se encuentra a la derecha del editor.

Por ejemplo, para crear el puerto de salida se debe seleccionar el icono *Output Port* y luego hacer click en el área del editor donde se desee colocar el puerto. Inmediatamente después de crear el puerto, se puede modificar su nombre, ya sea directamente sobre la figura del puerto o bien desde la vista de propiedades. De la misma manera, puede crearse el modelo atómico del generador utilizando el icono *Generator* que se encuentra en la sección de modelos reusables del panel de herramientas. Para crear el link, debemos seleccionar el icono *Link*, luego hacer click sobre el puerto de origen (en nuestro caso el puerto *out* del generador) y manteniendo el botón del mouse presionado arrastrarlo hasta el puerto destino (en nuestro caso el puerto *out* del modelo global). Todos los elementos se pueden mover a cualquier lado del editor, seleccionándolos y arrastrando como muestra la siguiente Figura:

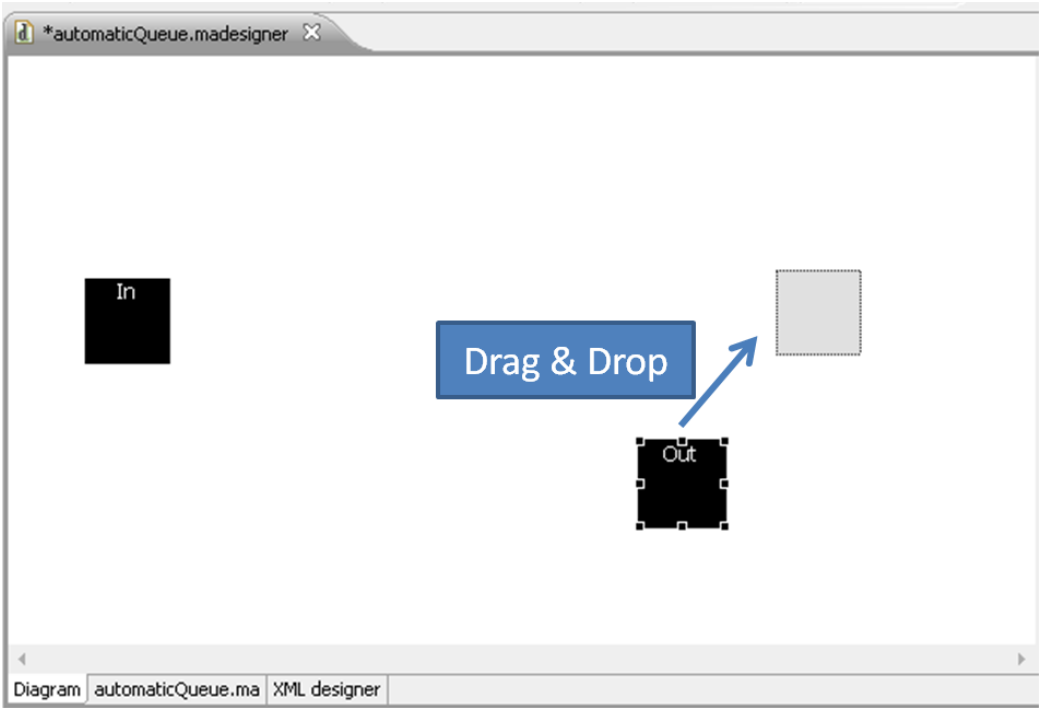
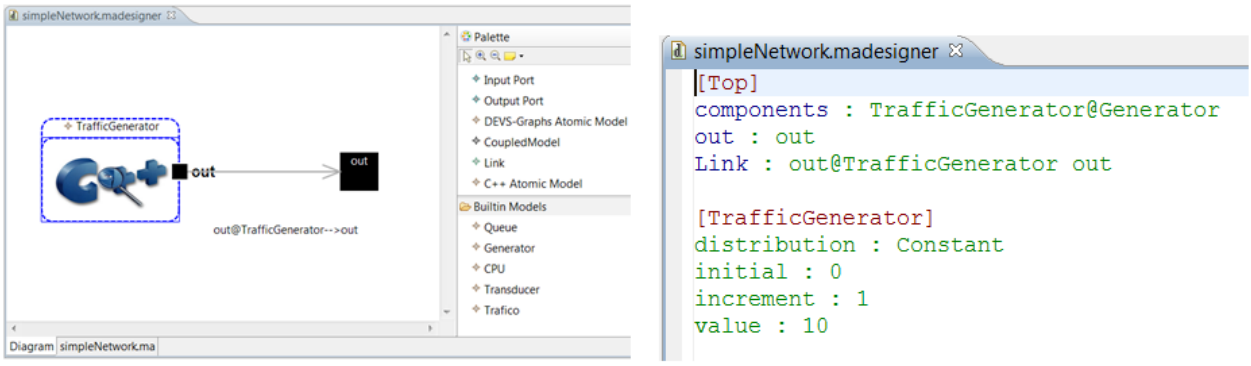


Figura 63 - Posicionamiento de las figuras mediante Drag&Drop.

Los valores de los parámetros para el generador también pueden colocarse en la vista de propiedades. Para este ejemplo usaremos una distribución denominada “constante” (determinista), que genere paquetes cada exactamente 10 segundos, para simplificar la comprensión del ejemplo (en un modelo real, se puede modificar la distribución para introducir aleatoriedad).



a) b)

Figura 64 - Modelo acoplado simple a) vista gráfica b) vista textual.

Luego grabamos el modelo (puede realizarse tanto presionando *Cntrl+s*, utilizando los iconos de acción o utilizando la opción de menú *File--> Save*) para que la especificación gráfica se sincronice con la especificación textual del modelo. Una vez que hayamos grabado el modelo, podemos utilizar la solapa inferior para cambiar a la vista textual como muestra la Figura 64 b.

Con esto ya estamos en condiciones de ejecutar la primer simulación. Para esto se puede utilizar el botón de acción de simulación  o bien haciendo click derecho sobre el editor seleccionando la opción *simulate*. Esto abrirá el formulario para especificar los parámetros de la simulación que deben completarse como la Figura 65, y luego presionar S*tart Simulation.*

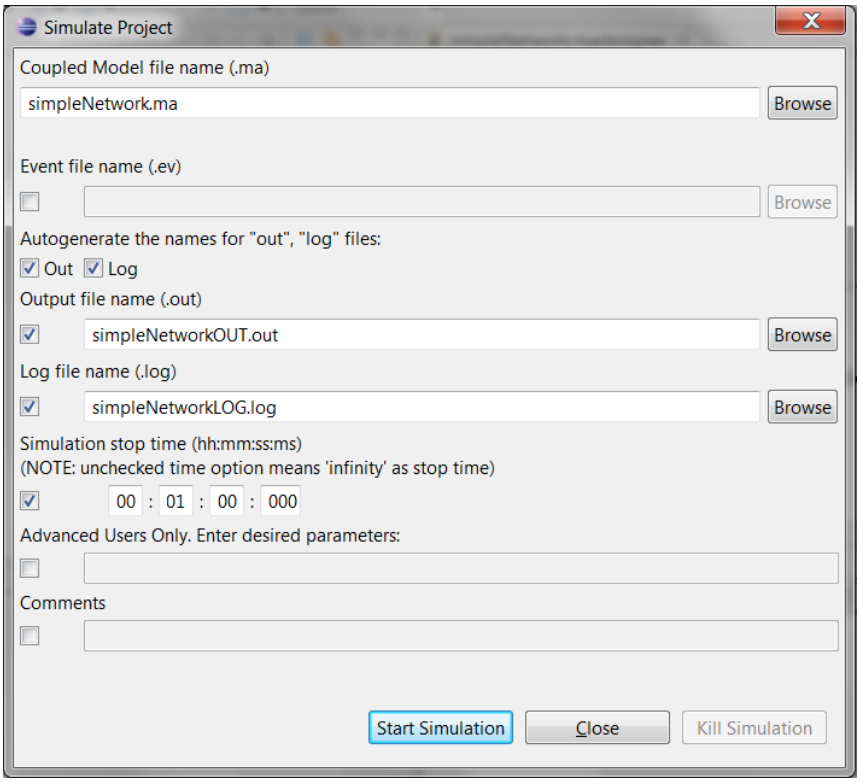


Figura 65 - Ejecución de la simulación.

En la vista de consola de CD++Builder podremos ver como avanza la simulación, los parámetros utilizados, versión del simulador, etc. (Figura 66 a). Al finalizar la ejecución, se generarán los dos archivos de salida especificados (*simpleNetwork.out* y *simpleNetwork.log*). En la Figura 66 b se pueden ver los eventos de salida del archivo *simpleNetwork.out.*

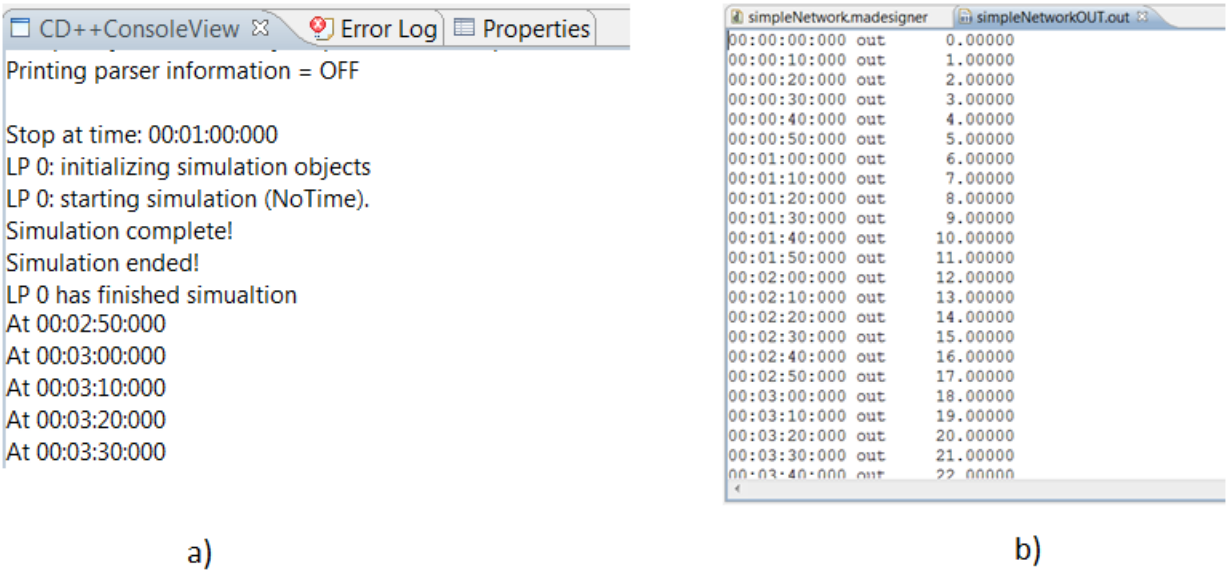


Figura 66 - a) Consola de CD++Builder b) archivo simpleNetwork.out luego de la primer simulación

Como podemos observar, por un lado pueden utilizarse para modelar en forma gráfica los modelos preexistentes en CD++ (en este ejemplo el *Generator),* donde se muestran los puertos y parámetros que estos utilizan. Por otro lado, para generar la definición textual del modelo (que es la que utiliza CD++ para realizar la simulación) no es necesario realizar un paso extra para exportar el modelo, sino que se realiza en forma automática al guardar los cambios. Finalmente, se vio como se puede ejecutar la simulación y ver los resultados desde la misma interfaz, sin tener que ejecutar otra aplicación.

Este modelo simplemente genera valores aleatorios y luego los envía por su puerto de salida. En nuestra segunda iteración, vamos agregaremos un nuevo modelo acoplado que represente a la red, con un puerto de entrada y un puerto de salida para los paquetes, que genere un retraso de 5 segundos y pierda el 50% de los paquetes. Dado que este es un comportamiento relativamente simple, utilizaremos DEVS-Graphs para representarlo.

Dado que el modelo de la red puede hacerse arbitrariamente complejo según el problema que se quiera modelar, utilizaremos un modelo acoplado para encapsular su comportamiento, previendo que en futuras iteraciones complejizaremos el funcionamiento y requeriremos de más de un modelo atómico para representarlo.

Utilizando el icono *Coupled Model* del panel de herramientas del editor, creamos el modelo acoplado y le colocamos el nombre *network* como muestra la siguiente Figura:

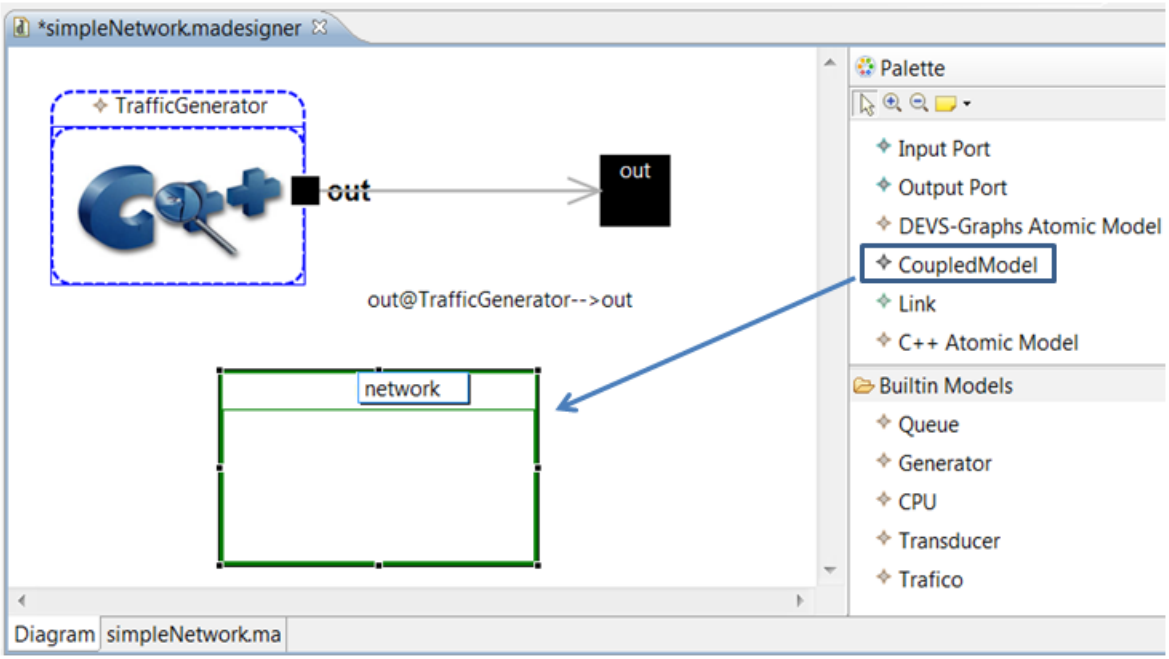


Figura 67 - Creación del modelo acoplado para representar la red.

El modelo acoplado se crea vacío (sin puertos ni modelos internos), por lo que debemos editarlo para crear el modelo atómico DEVS-Graphs (que llamaremos *networkDelay*) que le definirá el comportamiento. Para editar este nuevo modelo acoplado debemos hacer doble click sobre su figura. Se abrirá una nueva solapa en Eclipse con el editor para el modelo *network*, donde podremos agregarle los puertos (*Input Port* y *Output Port*) y el modelo DEVS-Graphs (*DEVS-Graphs Atomic Model*) utilizando el panel de herramientas como muestra la siguiente Figura:

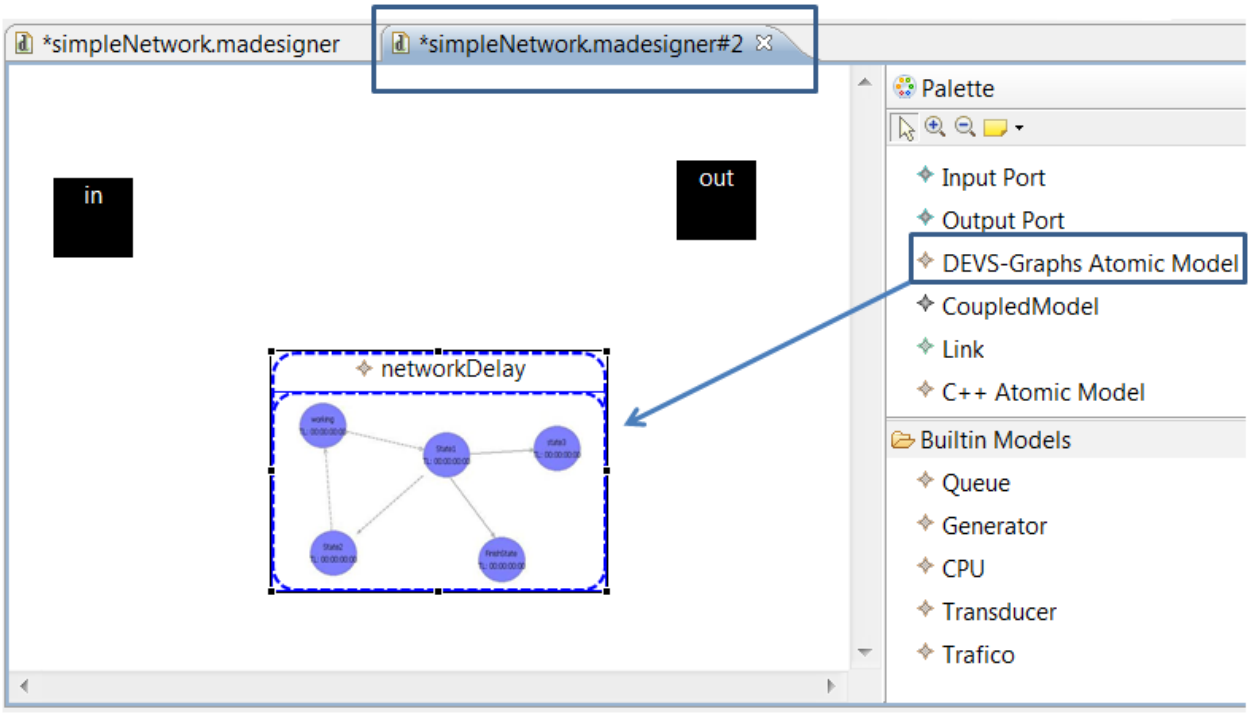


Figura 68 - Composición del modelo acoplado network.

Para definir internamente al modelo DEVS-Graphs, hacemos doble click sobre su figura. En este caso se abrirá una nueva solapa con el editor gráfico de modelos atómicos. La definición del modelo se puede ver en la siguiente Figura:

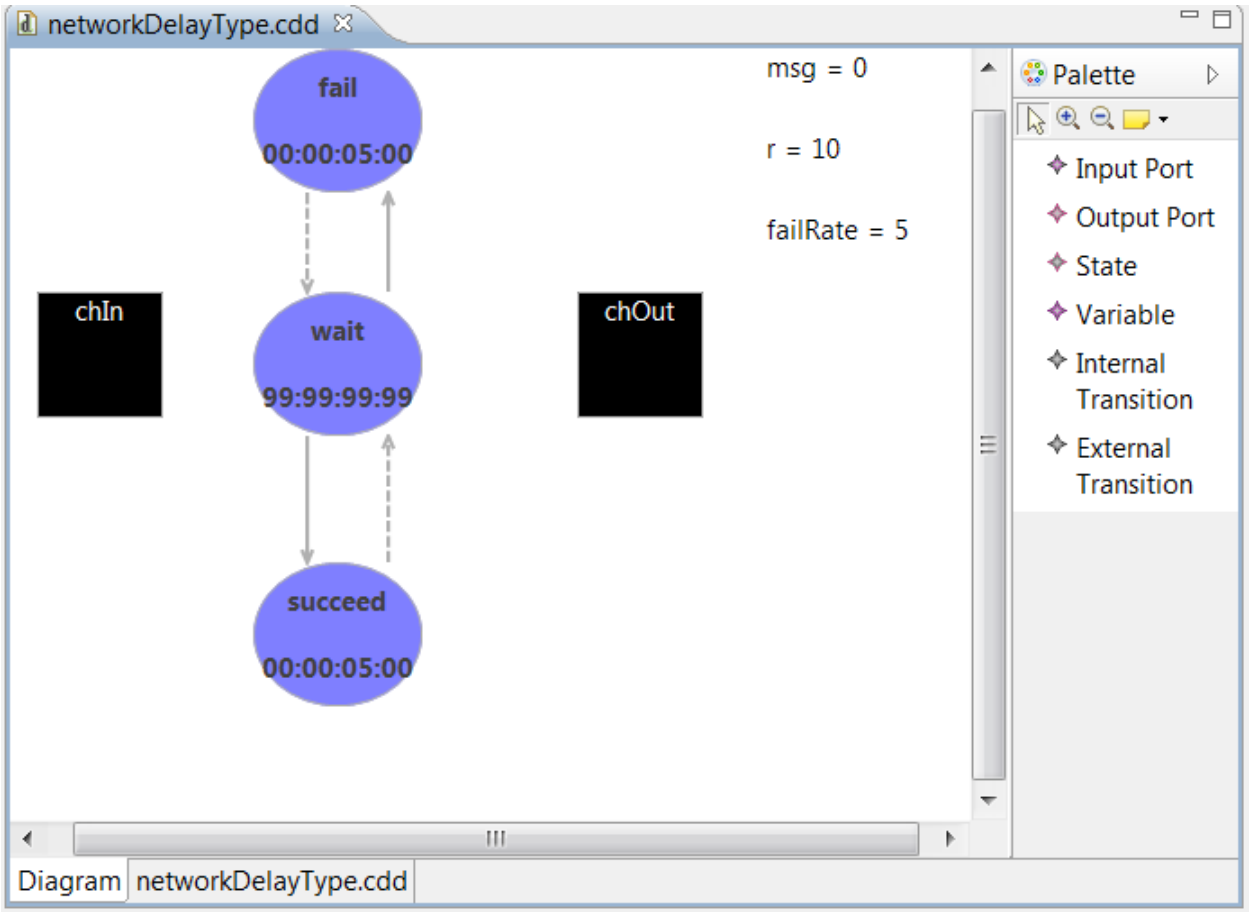


Figura 69 - Modelo atómico DEVS-Graphs networkDelay.

Para este modelo se utilizan tres estados: *wait, succeed y fail*. En el estado *wait*, el modelo se quedará esperando la llegada de nuevos paquetes por el puerto *chIn*. Cuando esto sucede, aleatoriamente (con cierta probabilidad dada por la variable *failRate*) cambiará al estado *succeed* o *fail.* En el estado *succeed* al transcurrir cinco segundos se enviará el último valor recibido (almacenado en la variable *msg*) por el puerto *chOut* y se volverá al estado *wait*. En el estado *fail*, simplemente se espera 5 segundos para volver al estado *wait* inmediatamente (perdiendo el último paquete recibido).

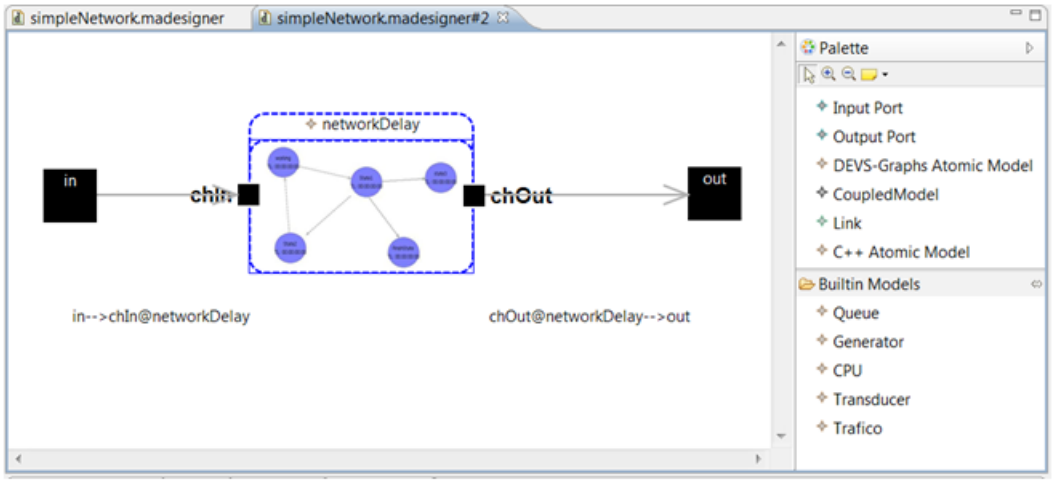
Para definir el modelo se puede utilizar los iconos del panel de herramientas, arrastrando cada elemento (estados, variables, puertos, etc). Sin embargo dado que son mucho elementos es más simple para este ejemplo copiar la definición de la Figura 70 en el editor textual (solapa inferior *networkDelayType.cdd*). Al grabar el modelo también podremos ver en la solapa *Diagram* la definición gráfica que se genera automáticamente a partir de la definición textual que acabamos de introducir.

```
[Top]
in:chIn
out:chOut
var:msg r failRate
state:succeed wait fail
initial:wait
```

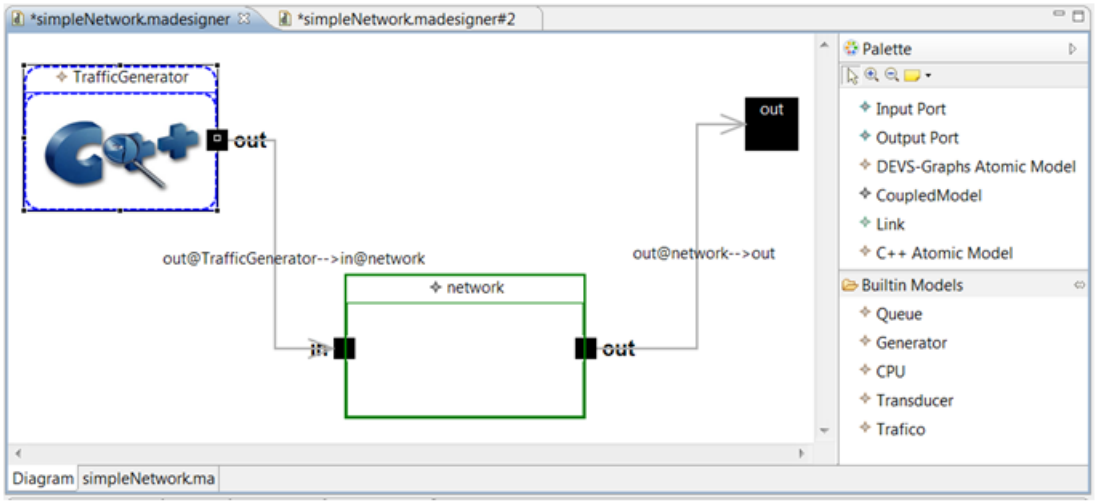
```
int:succeed wait chOut!msg
int:fail wait
ext:wait                                                                    succeed
And(Any(Value(chIn)),Greater(r,failRate))?1{msg=Value(chIn);r=Rand(1,10);}
ext:wait fail And(Any(Value(chIn)),Less(r,failRate))?1{r=Rand(1,10);}
succeed:00:00:05:00
wait:99:99:99:99
fail:00:00:05:00
msg:0
r:10
failRate:5
```

Figura 70 - Definición textual del modelo atómico DEVS-Graphs networkDelay.

Con el modelo DEVS-Graphs ya definido, sólo resta completar el acoplamiento de los modelos acoplados uniendo los puertos apropiadamente. Si volvemos al editor del modelo acoplado *network*, veremos que ahora el modelo DEVS-Graphs posee un puerto de entrada y uno de salida, tal como lo definimos anteriormente, por lo que ahora podemos crear los links como muestra la Figura 71 a. En el editor del modelo global *simpleNetwork,* debemos remover el link previo y crear dos nuevos links que conecten con el modelo acoplado *network,* como muestra la Figura 71 b.



a)



b)

Figura 71 - Acoplamiento de los modelos acoplados a) network b) simpleNetwork.

Grabando todos los editores, estamos en condiciones ejecutar nuevamente la simulación (presionando el botón de acción de simulación  o bien haciendo click derecho sobre el editor seleccionando la opción *simulate*.) utilizando los mismos parámetros que antes. En este caso podemos observar en el archivo *.out (*Figura 72*)*, que los eventos de salida (transmisión de paquetes) se producen con un retraso de 5 segundos, y que aleatoriamente algunos no se producen.

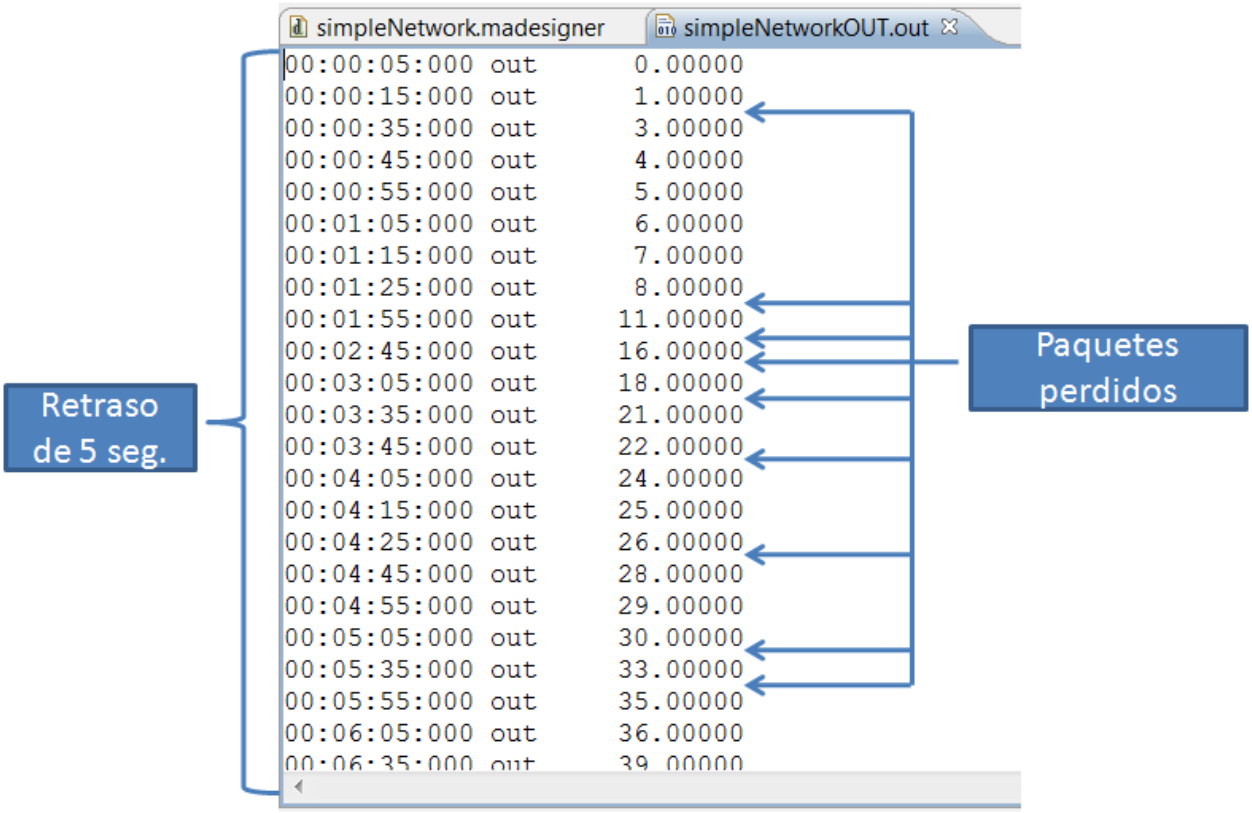


Figura 72 - Eventos de salida luego de la segunda simulación.

Se puede observar como en CD++Builder 2.0 se ha mejorado la experiencia en cuanto a la navegación jerárquica de los modelos manteniendo la consistencia de los mismos. Al hacer doble click sobre un modelo, se abre una nueva sola donde se puede editar la definición de este modelo, mientras que su representación gráfica se mantiene consistente en el modelo padre. Por ejemplo, cuando agregamos nuevos puertos en el modelo acoplado *Network*, estos se muestran automáticamente en el modelo *simpleNetwork.* En las herramientas anteriores, al explotar un modelo se pedían todos los links que estaban conectados al mismo, haciendo muy tediosa su definición.

Por otro lado, se vio como puede definirse un modelo completamente en forma textual (en nuestro ejemplo el modelo DEVS-Graphs *networkDelay)* y se generará automáticamente una vista gráfica del mismo. Esto no era posible en las herramientas anteriores, por lo que era imposible editar gráficamente modelos preexistentes.

Si bien este modelo funciona bien para el ejemplo que propusimos aquí, no puede modificarse el tiempo de retraso dinámicamente y puede tener problemas si se generan paquetes mientras el modelo atómico se encuentra en el estado *fail* o *succeed* (ya que estos estados no poseen ninguna transición externa definida). El primer problema se debe a que los modelos DEVS-Graphs especifican un tiempo para cada estado, pero este no puede modificarse una vez que la simulación se está ejecutando.

Para solucionar el segundo problema, se deberían incluir transiciones externas en los estados *fail* y *succeed* para poder encolar en una variable los mensajes que llegan mientras el modelo se encuentra en alguno de estos estados. Sin embargo, DEVS-Graphs solo permite crear variables de tipo entero (sería útil contar con una variable de array para encolar los paquetes).

Luego, en nuestra tercer iteración extenderemos el modelo para poder modificar el tiempo de retraso mientras la simulación está corriendo y a su vez poder encolarlos valores de los paquetes que lleguen mientras se está produciendo el retardo. Como dijimos, DEVS-Graphs no permite definir este comportamiento, desarrollaremos un modelo atómico en C++ que reemplace al modelo *networkDelay* anterior.

Para esto, comenzamos abriendo el editor del modelo acoplado *network* y borramos el modelo atómico DEVS-Graphs. Luego,utilizando el panel de herramientas, creamos un modelo atómico C++ (icono *C++ Atomic Model)* con nombre *networkDelay.*

Para configurar dinámicamente el retraso, agregaremos un nuevo puerto de entrada con nombre *setDelay* al modelo acoplado. La siguiente figura muestra el modelo resultante:

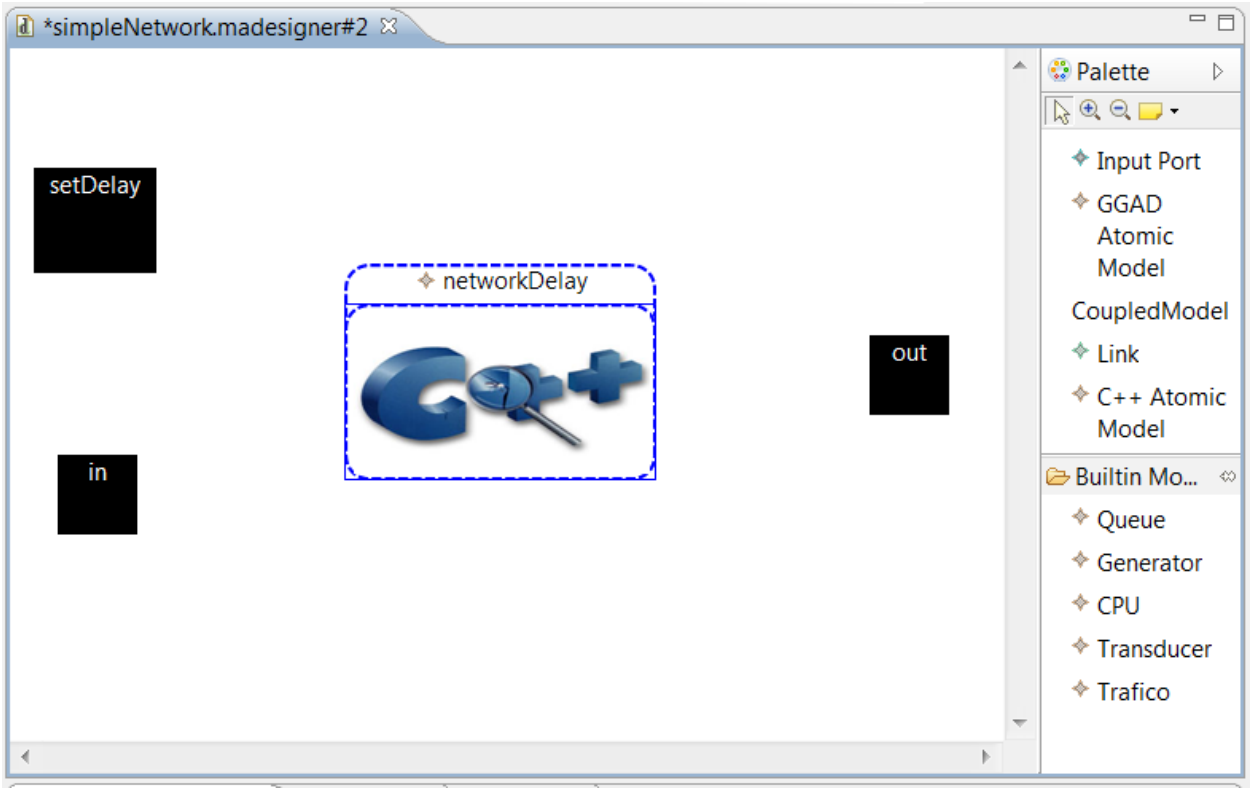


Figura 73 - Modelo acoplado network, luego de reemplazar el modelo DEVS-Graphs por uno C++.

Al hacer doble click sobre el nuevo modelo atómico, se crearán automáticamente los archivos necesarios para implementar un nuevo modelo atómico de CD++. También se abrirá el editor de CDT

con un código C++ tipo template pre-generado para el nuevo modelo, sobre el cual podremos incorporar nuevas funcionalidades.

El archivo *register.cpp* no necesita modificación manual, e incorpora automáticamente las referencias necesarias al nuevo atómico C++.

Debemos modificar los archivos generados *networkDelayType.cpp* y *networkDelayType.h* con el siguiente código (en fondo gris se encuentran las líneas de código que introducimos manualmente y no forman parte del template):

```
/*******************************************************************
*                              Auto Generated File                 *
*******************************************************************/
#ifndef __networkDelayType_H
#define __networkDelayType_H

/** include files **/
#include "atomic.h"  // class Atomic

/** forward declarations **/
//TODO: add distribution class declaration here if needed
// Ej: class Distribution ;

/** declarations **/
class networkDelayType: public Atomic
{
public:
      networkDelayType( const string &name = "networkDelayType" );// Default
constructor
      ~networkDelayType();    // Destructor
      virtual string className() const { return "networkDelayType"; }

protected:
      Model &initFunction();
      Model &externalFunction( const ExternalMessage & );
      Model &internalFunction( const InternalMessage & );
      Model &outputFunction( const InternalMessage & );

private:
//[(!) TODO: declare ports, distributions and other private varibles here]
      /*******************  Example declarations   *********************/
       const Port &in;  // this is an input port named 'in'
       Port &out ;      // this is an output port named 'out'
       const Port &setDelay;  // this is an input port named 'setDelay'

      // Distribution *myDistribution ;
      // Distribution &distribution()     { return *myDistribution; }
      /*****************************************************************/

      // [(!) declare common variables]
      // Lifetime programmed since the last state transition to the next
planned internal transition.
      Time sigma;
```

```
	// Time elapsed since the last state transition until now
	Time elapsed;

	// Time remaining to complete the last programmed Lifetime
	Time timeLeft;

	int delay;

	typedef list<Value> ElementList ;
	ElementList elements ;
};	// class networkDelayType
#endif	//__networkDelayType_H
```

Figura 74 - networkDelayType.h - Implementación del modelo atómico C++ networkDelay.

```
/*******************************************************************
*                              Auto Generated File                 *
*******************************************************************/

/** include files **/
#include "networkDelayType.h"           // base header
#include "message.h"        // InternalMessage ....
#include "distri.h"         // class Distribution
#include "mainsimu.h"       // class MainSimulator
#include <stdlib.h>
#include <time.h>

/*******************************************************************
* Function Name: networkDelayType
* Description: constructor
*******************************************************************/
networkDelayType::networkDelayType( const string &name )
: Atomic( name )
// TODO: add ports here if needed. Each in a new line. Ej:
 , out(addOutputPort( "out" ))
 , in(addInputPort( "in" ))
 , setDelay(addInputPort( "setDelay" ))
{
	// TODO: add initialization code here. (reading parameters, initializing
private vars, etc)
	// Code templates for reading parameters:
	// read string parameter:
	delay =     str2Int(MainSimulator::Instance().getParameter(
description(), "initialDelay" ));
	// read int parameter:
	//          str2Int(             MainSimulator::Instance().getParameter(
description(), "initial" ) );
	// read time parameter:
	//          string     time(     MainSimulator::Instance().getParameter(
description(), "preparation" ) ) ;
	// read distribution parameters:
	//          dist                 =                 Distribution::create(
MainSimulator::Instance().getParameter( description(), "distribution" ) );
	//          MASSERT( dist ) ;
	//          for ( register int i = 0; i < dist->varCount(); i++ )
	//          {
```

```
	//		string parameter( MainSimulator::Instance().getParameter( description(), dist->getVar( i ) ) ) ;
	//		dist->setVar( i, str2Value( parameter ) ) ;
	//	}
}

/*******************************************************************
* Function Name: initFunction
*******************************************************************/
Model &networkDelayType::initFunction()
{
	// [(!) Initialize common variables]
	this->elapsed = Time::Zero;
	this->timeLeft = Time::Inf;
	this->sigma = Time::Inf; // stays in active state until an external event occurs;
	//this->sigma = Time::Zero; // force an internal transition in t=0;

	// TODO: add init code here. (setting first state, etc)
	elements.erase( elements.begin(), elements.end() ) ;

	/* initialize random seed: */
	srand ( time(NULL) );

	// set next transition
	holdIn( active, this->sigma ) ;
	return *this ;
}

/*******************************************************************
* Function Name: externalFunction
* Description: Add the description for the external function here
*******************************************************************/
Model &networkDelayType::externalFunction( const ExternalMessage &msg )
{
	//[(!) update common variables]
	this->sigma = nextChange();
	this->elapsed = msg.time()-lastChange();
	this->timeLeft = this->sigma - this->elapsed;

	if( msg.port() == setDelay )
	{
		// set new delay and
		delay = msg.value();
	}
	else if( msg.port() == in )
	{
		// Queue the value of the message
		elements.push_back( msg.value() ) ;
	}

	Time nextChange = Time(0,0,delay,0);
	if(this->timeLeft < nextChange)
		nextChange =  this->timeLeft;
```

```
	// continue (leaving TA unchanged, so that new delay will take effect
after next output)
	holdIn( active, nextChange );
	return *this ;
}

/*******************************************************************
* Function Name: internalFunction
*******************************************************************/
Model &networkDelayType::internalFunction( const InternalMessage &msg )
{
	if( elements.size() == 0 )
		this->sigma =  Time::Inf;
	else
		this->sigma = Time(0,0,delay,0);

	//this->sigma = Time::Inf; // stays in active state until an external
event occurs;
	holdIn( active, this->sigma  );
	return *this;
}

/*******************************************************************
* Function Name: outputFunction
*******************************************************************/
Model &networkDelayType::outputFunction( const InternalMessage &msg )
{
	//TODO: implement the output function here
	// remember you can use sendOutput(time, outputPort, value) function.
	// sendOutput( msg.time(), out, elements.front() ) ;

	Value val = elements.front();
	elements.pop_front();

	float v = rand() % 10; // random between 0-10
	if(v < 5)
	{
		sendOutput( msg.time(), out, val ) ;
	}

	return *this;
}

networkDelayType::~networkDelayType()
{
	//TODO: add destruction code here. Free distribution memory, etc.
}
```

Figura 75 - networkDelayType.cpp - Implementación del modelo atómico C++ networkDelay.

En contraste con las versión anterior de la herramienta donde no existía ningún soporte para desarrollar modelos atómicos C++, para el desarrollo del código anterior el usuario cuenta con ayudas que simplifican la implementación, tanto a nivel editor de código (plugin CDT: código coloreado, resumen de los métodos, etc.) como también en el código template generado (líneas comentadas con explicaciones orientativas, sugerencias de usos frecuentes, buenas prácticas de modelado, etc.). No es necesario crear

y/o modificar el archivo de registración register.cpp, ya que se genera en forma automática. El editor gráfico de modelos acoplados reconoce una sintaxis específica de CD++ que permite representar los puertos y los parámetros que reciben todos los modelos atómicos que compongan al modelo acoplado. Esta representación además se mantiene consistente, por lo que si se realiza una modificación en el código, la misma se reflejará en la representación gráfica del modelo acoplado.

Una vez que guardemos los dos archivos que definen el modelo *networkDelay,* podemos volver al editor de modelos acoplados para definir la unión de los puertos como muestra la Figura 76. Como se puede observar en la figura ahora el editor muestra los puertos que fueron definidos en el archivo C++, y la vista de propiedades permite colocar un valor para el parámetro *initialDelay.*

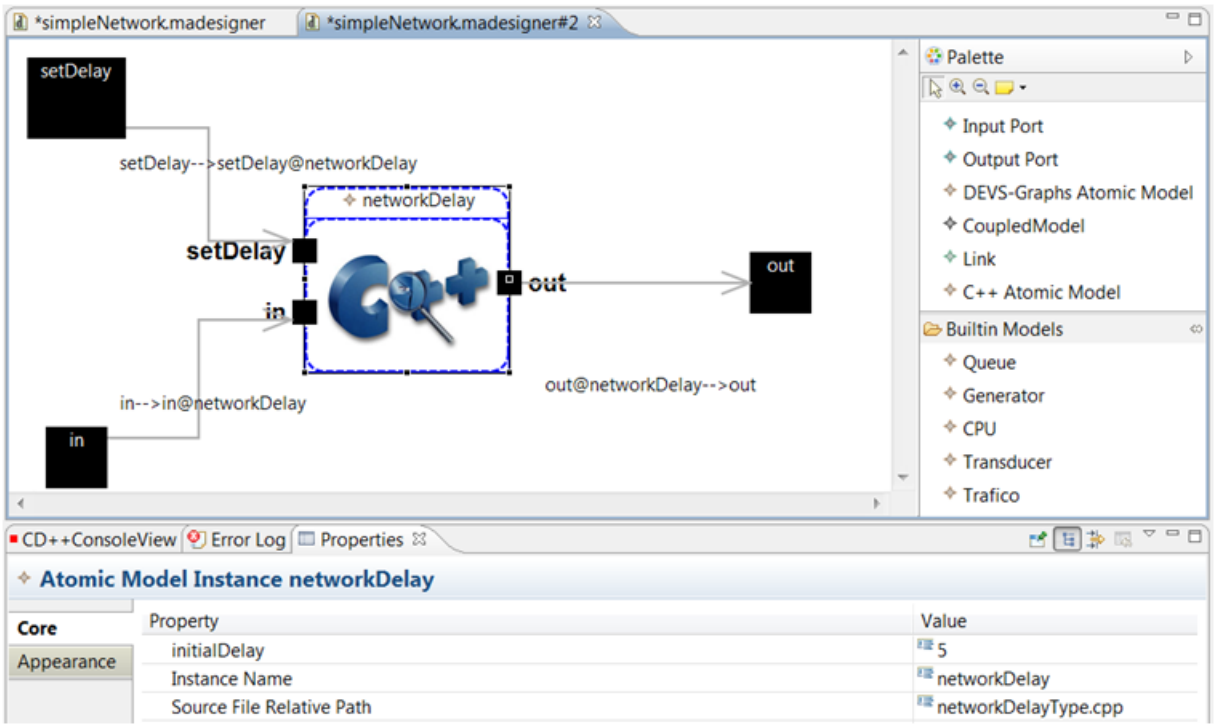


Figura 76 - Modelo acoplado network con modelo C++.

Para poder modificar dinámicamente el tiempo de retraso, a través de eventos de entrada, en el modelo global *simpleNetwork* también tendremos que definir un puerto de entrada setNetworkDelay y unirlo al puerto del modelo acoplado network como muestra la siguiente figura:

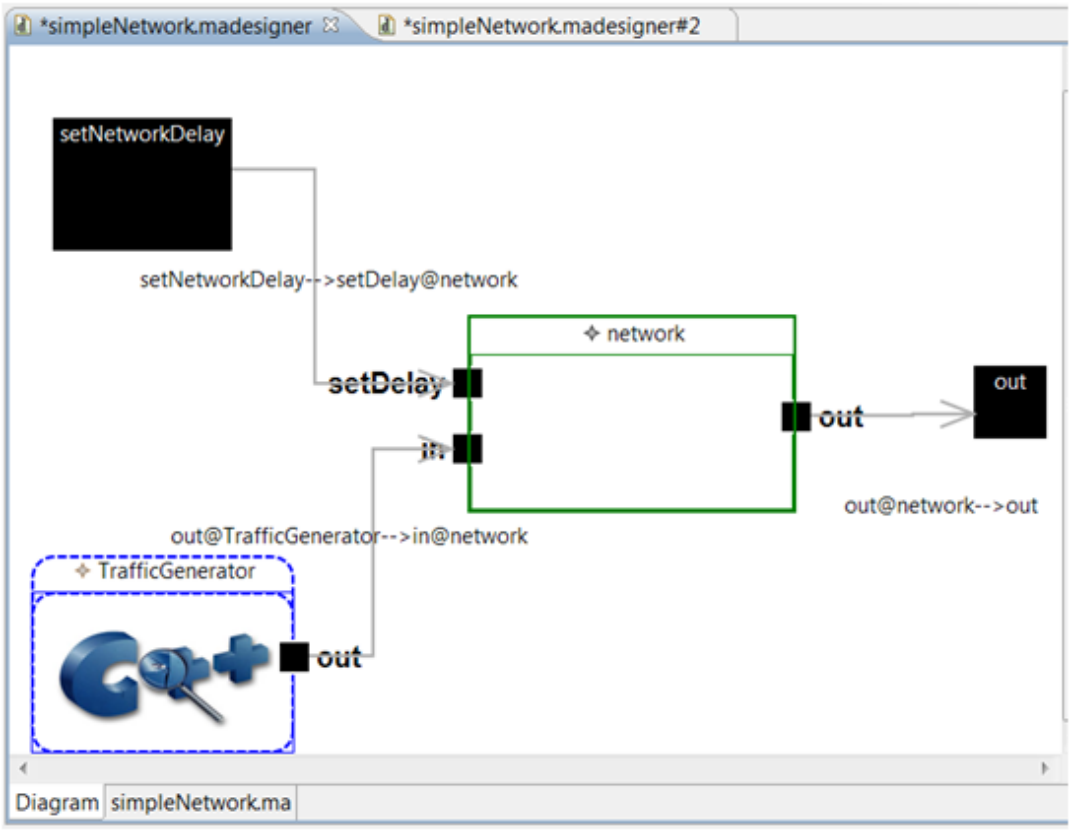


Figura 77 - Modelo simpleNetwork con puerto de entrada setNetworkDelay.

Dado que el modelo ahora contiene modelos atómicos desarrollados en C++ que no vienen incluidos en el simulador CD++, es necesario recompilar el simulador para que incluya el nuevo modelo atómico. Para realizar esta tarea simplemente seleccionamos la carpeta donde están todos los archivos que describen al modelo (en la vista de navegación del panel izquierdo) y presionamos el botón de acción *Build* . Se creará un nuevo ejecutable dentro de la carpeta seleccionada. En la consola de CD++Builder se pueden ver los comandos ejecutados, resultado de la compilación y los errores en caso de que existan. La siguiente figura muestra el resultado de una compilación correcta del modelo:

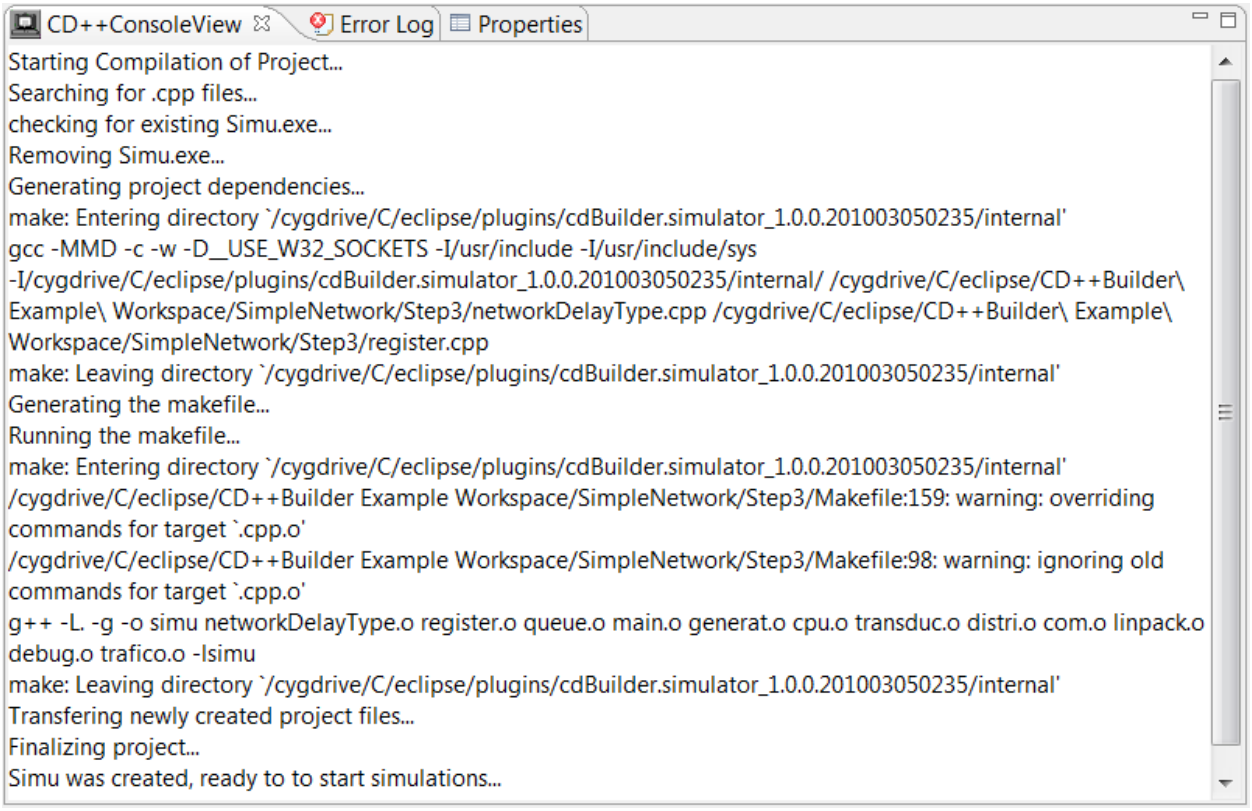


Figura 78 - Compilación del modelo simpleNetwork en la consola de CD++Builder.

Dodo que este modelo recibe eventos externos para configurar dinámicamente el retraso, es necesario crear un archivo que describa estos eventos. Para esto creamos un archivo utilizando los asistentes de Eclipse. En general para este archivo se suele utilizar el mismo nombre del modelo pero con extensión *.ev*. Para este ejemplo utilizaremos los siguientes eventos externos, que especifican que a los tres minutos de comenzada la simulación el tiempo de retraso de la red aumentará 8 segundos y a los 6 minutos disminuirá a 2 segundos.

```
00:03:00:00 setNetworkDelay 8
00:06:00:00 setNetworkDelay 2
```

Figura 79 - simpleNetwork.ev - Eventos de entrada para el modelo simpleNetwork.

Para correr la simulación utilizamos el botón de acción de simulación  o bien haciendo click derecho sobre el editor seleccionando la opción *simulate*, en este caso para correr la simulación utilizando el archivo de eventos, debemos completar los parámetros como muestra la siguiente Figura:

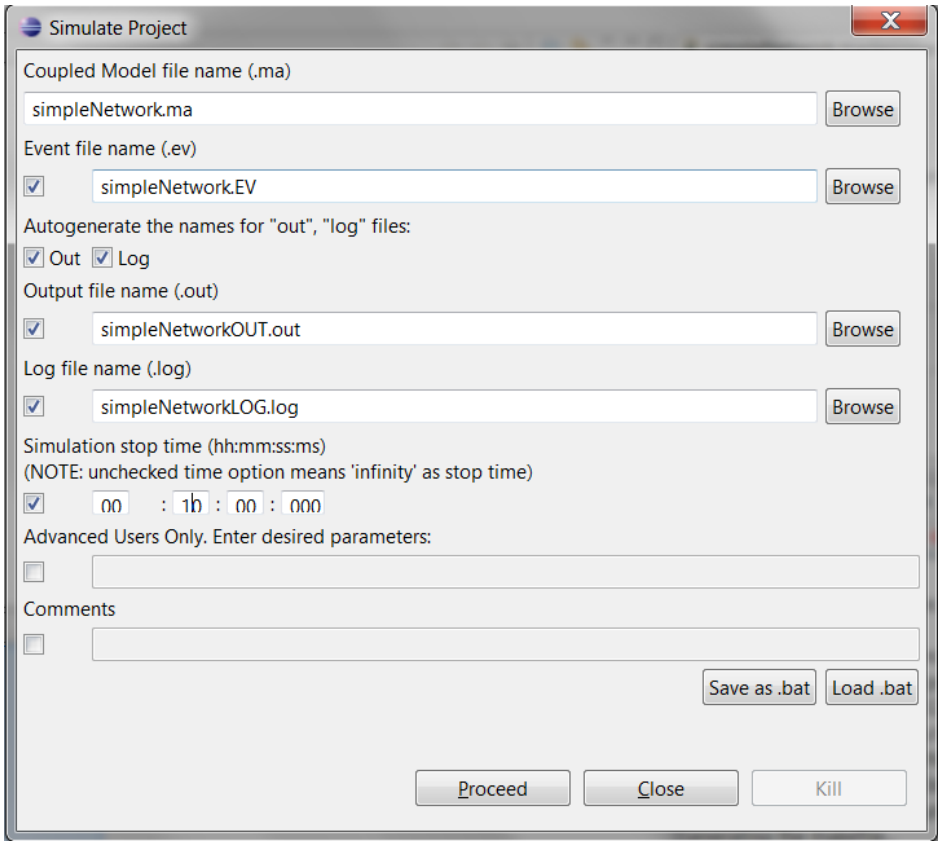


Figura 80 - Parámetros para utilizar un archivo de eventos en la simulación.

Al finalizar la simulación se puede observar, en el archivo de salida *simpleNetwork.out* (Figura 81), como se modifica el retraso de los mensajes en los minutos tres y seis.

```
00:00:15:000 out 1
............
00:02:35:000 out 15
00:02:55:000 out 17
00:03:08:000 out 18
00:03:18:000 out 19
00:03:28:000                                    out                                    20
............
00:05:48:000 out 34
00:05:58:000 out 35
00:06:02:000 out 36
00:06:12:000 out 37
00:06:32:000                                    out                                    39
...........
00:09:52:000 out 59
```

Figura 81 - simpleNetwork.out - Eventos externos del modelo simpleNetwork modificando dinámicamente el tiempo de retraso en los minutos 3 y 6.

Además del archivo de eventos de salida, el simulador también crea archivos de log donde se detalla en orden temporal todo el flujo de la simulación, los mensajes que se intercambian entre los diferentes

modelos, etc. Como se vio en el Capítulo 2 - Background, este es un archivo de texto que puede ser utilizado para crear animaciones 2D y 3D de la simulación.

Para visualizar la ejecución de la simulación del modelo *simpleNetwork* que desarrollamos anteriormente podemos utilizar el botón de acción *Animate Coupled* , debiendo especificar los siguientes parámetros:

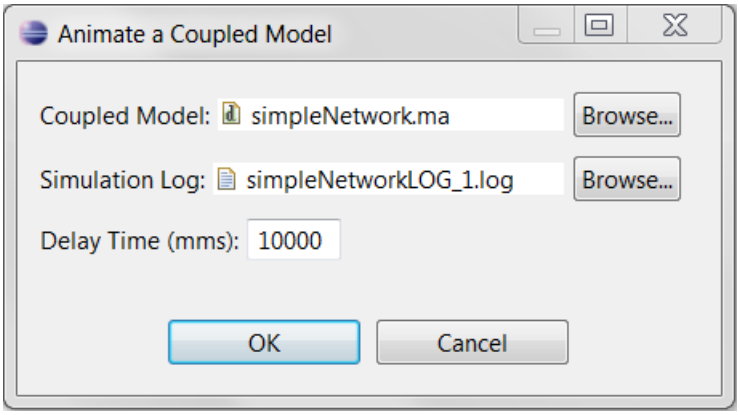


Figura 82 - Parámetros para la animación del modelo acoplado simpleNetwork.

Se abrirá una nueva ventana mostrado el modelo acoplado seleccionado. Se pueden utilizar los botones de la barra superior para controlar el avance y retroceso de la animación. Los links se tornan color rojo mostrando el traspaso de mensajes y su valor. La siguiente Figura muestra algunas tomas de la animación de la simulación de del modelo desarrollado anteriormente.

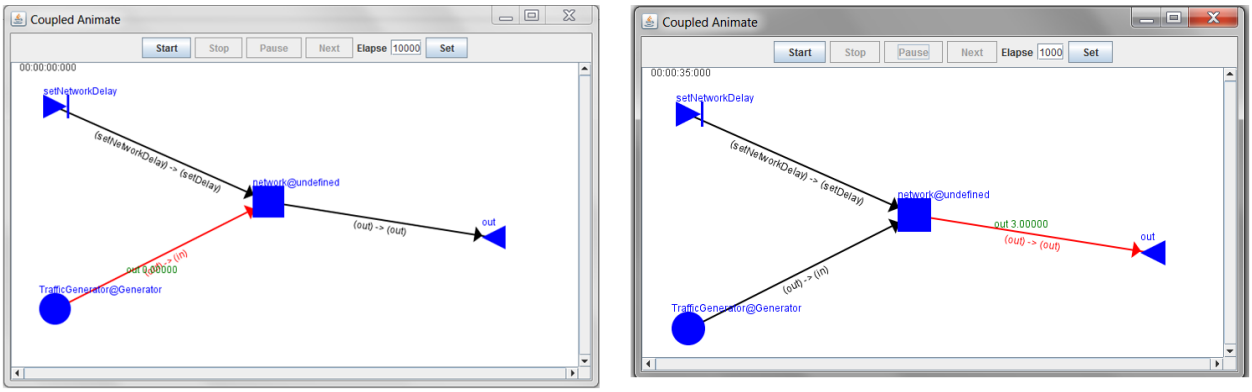


Figura 83 - Animación del modelo acoplado simpleNetwork.

También se puede analizar visualmente el archivo de eventos utilizando la animación de modelos atómicos, que muestra el valor que toman los diferentes puertos en un eje de coordenadas puerto vs. tiempo. Para esto se utiliza el botón de acción de *Animate Atomic* , con los mismos parámetros que la animación anterior.

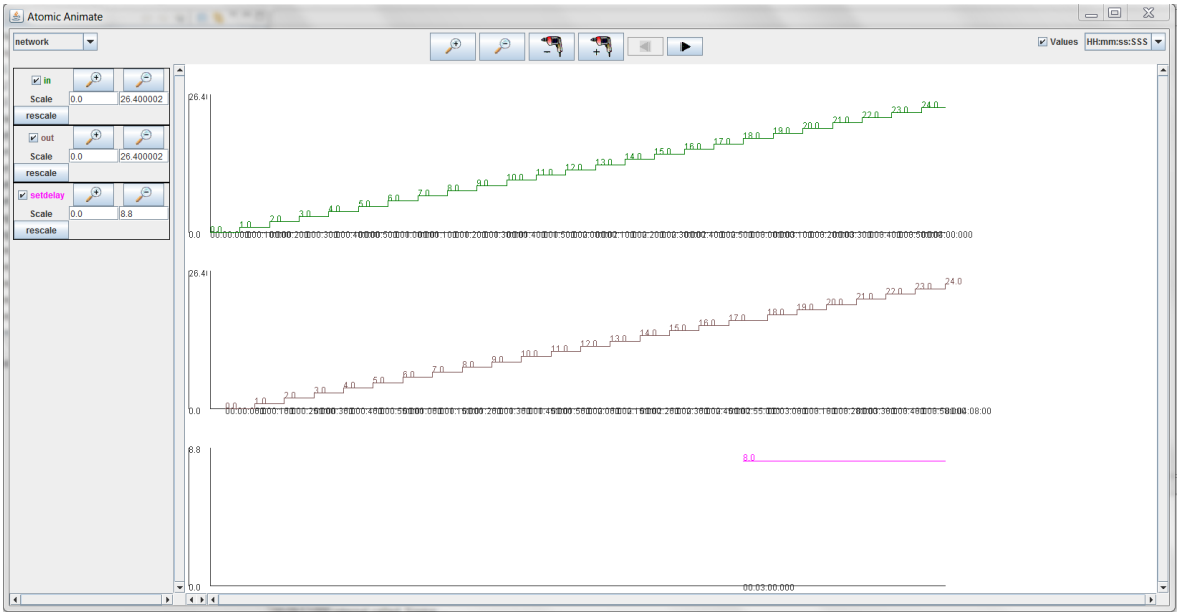


Figura 84 - Animación atómica del modelo acoplado simpleNetwork.

En esta figura se puede observar la evolución de los valores de los puertos del modelo acoplado *network.* Los dos primeros gráficos muestran los valores de los puertos *in* y *out* respectivamente, donde se puede ver el retraso de 5 segundos. El tercer gráfico muestra el valor del puerto *setDelay que se* mantiene constante en su valor inicial 0, hasta el minuto 3 en donde toma el valor 8 acorde al evento externo que llega vía el archivo *simpleNetwork.ev*.

4.2 Comparación con herramientas anteriores

4.2.1 Entorno de simulación

La versión anterior de CD++Builder brindaba un entorno de trabajo basado en Eclipse que permitía crear y simular modelos DEVS de manera no gráfica. Para crear modelos gráficamente se debían ejecutar aplicaciones externas. En el presente trabajo se mejoraron las capacidades de modelado gráfico de CD++Modeler y GGADTool, y se integraron dentro de la plataforma de Eclipse.

La integración de las herramientas facilita el proceso de modelado ya que evita pasos innecesarios de exportar los modelos de una aplicación a otra. Las capacidades y usabilidad de los editores gráficos se han mejorado (a través de comandos y opciones habituales de Eclipse, edición de estilos, etc.), y permitiendo complementarlos y extenderlos fácilmente con otros desarrollos (como por ejemplo CD++Repository).

En CD++Builder 2.0 para crear y simular un nuevo modelo gráficamente, es necesario utilizar una única herramienta, contenida en el propio entorno Eclipse, y no es necesario manipular (importar, exportar) archivos externos. Por otro lado, las modificaciones que se introducen en los modelos son bidireccionales, es decir, pueden hacerse tanto en los archivos de CD++ como en los modelos gráficos ya que se mantienen sincronizadas. Por esto, no es necesario realizar ningún paso extra simplificando el proceso de modelado notablemente. Esto puede verse comparando la Figura 43 del Capítulo 2-

Background (proceso de uso con las herramientas anteriores) con la siguiente figura (proceso con CD++Builder2.0):

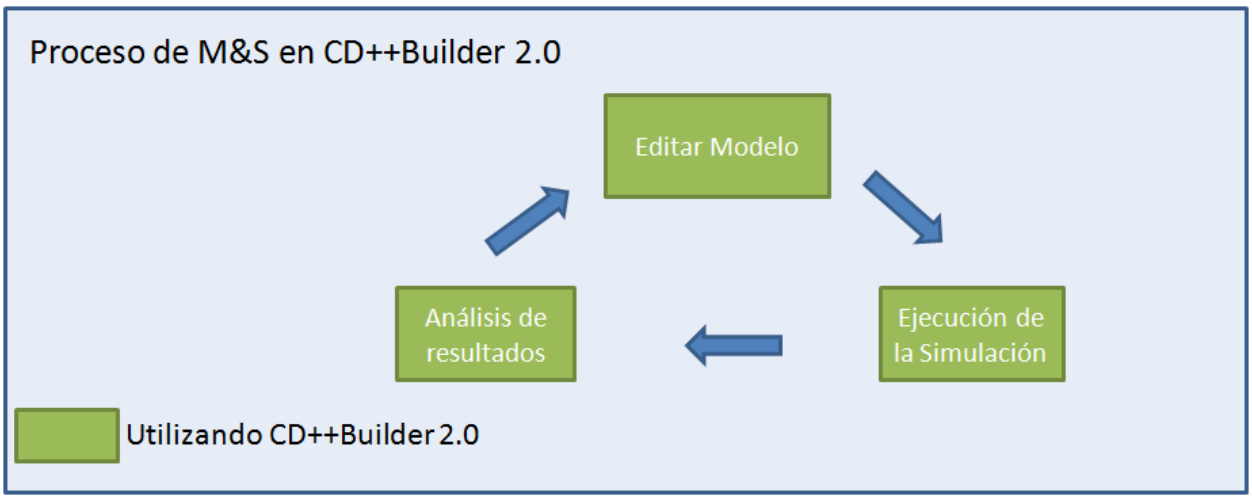


Figura 85 - Proceso de M&S con CD++Builder 2.0.

Por otro lado, la integración de herramientas permitió desarrollar funcionalidades que anteriormente no eran posibles. En CD++Modeler no era posible componer los modelos acoplados con modelos atómicos desarrollados en C++. Para desarrollar este tipo de modelos atómicos, debía programarse manualmente en Eclipse, y no existía ningún soporte gráfico. El editor de modelos acoplados de CD++Builder 2.0 permite crear modelos atómicos C++ gráficamente, mostrando tanto los parámetros que puede recibir el modelo como los puertos de entrada y salida (ver Figura 86). Esto se realiza mediante drag&drop utilizando en panel de herramientas del editor, que también permite utilizar modelos atómicos C++ que ya vienen incluidos en CD++.

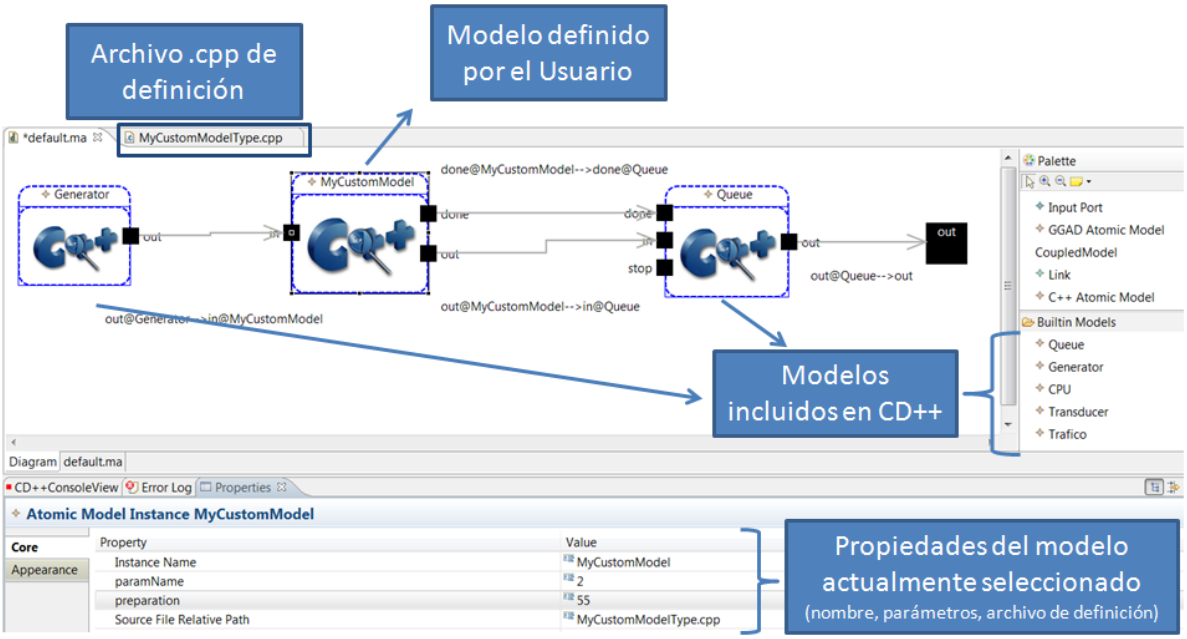


Figura 86 - Modelo acoplado compuesto por modelos atómicos C++.

Además del soporte gráfico, CD++Builder 2.0 también brinda herramientas que facilitan la implementación de modelos atómicos C++, simplificando el proceso completo de M&S (ver Figura 87), en contraste con las versiones anteriores donde sólo había soporte para algunas tareas (ver Figura 22 del capítulo de Background). Por un lado el código base para implementar la clase Atomic de CD++ se genera automáticamente junto con el archivo de registración *register.cpp*. Los comentarios y ejemplos del código generado y el plugin CDT facilitan la implementación del comportamiento (mediante editores especializados para C++, con coloreo de código, intelisense, etc.). Las animaciones simplifican en gran medida el análisis de los resultados, permitiendo ver el flujo de la simulación en forma gráfica. La automatización de las tareas de compilación y ejecución de la simulación ya estaban presentes en la versión anterior de CD++Builder.

El siguiente gráfico muestra el proceso que debe realizarse para modelar, simular y analizar los resultados para un modelo atómico C++

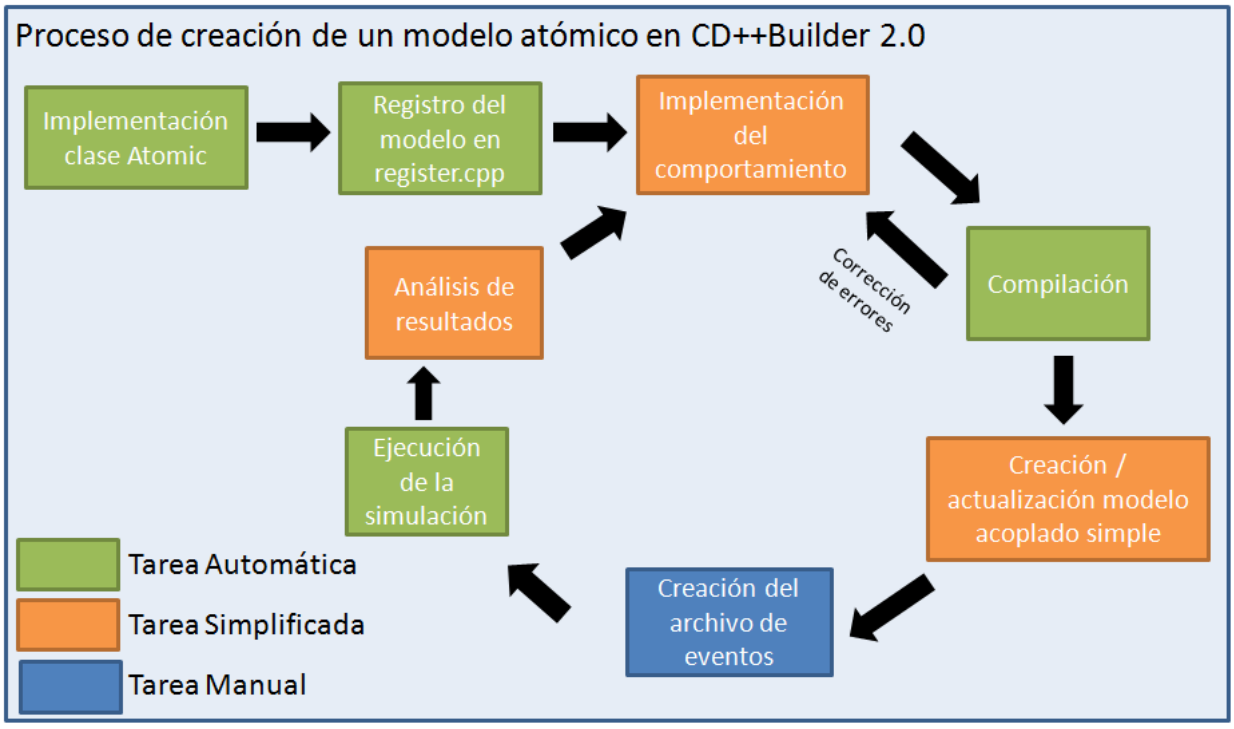


Figura 87 - Proceso de creación de un modelo atómico utilizando CD++Builder 2.0

Al comparar el proceso de la figura anterior (utilizando CD++Builder) con el proceso de la Figura 22 del capítulo de Background (sin utilizar CD++Builder), puede observarse que la gran mayoría de las tareas de M&S se ven simplificadas o automatizadas al utilizar CD++Builder. Si bien la creación del archivo de eventos de entrada debe realizarse en forma manual, es una tarea simple.

En cuanto a la extensibilidad de la plataforma, se han construido nuevas funcionalidades que no habían sido consideradas al comenzar este trabajo y de manera totalmente independiente por parte de otros grupos de investigación. Un ejemplo de esto fue el desarrollo realizado en forma paralela con CD++Builder 2.0 que implementa un repositorio de modelos para CD++. CD++Repository fue desarrollado en Carleton Univerity, Canadá, y luego se integró a CD++Builder como muestra la siguiente Figura:

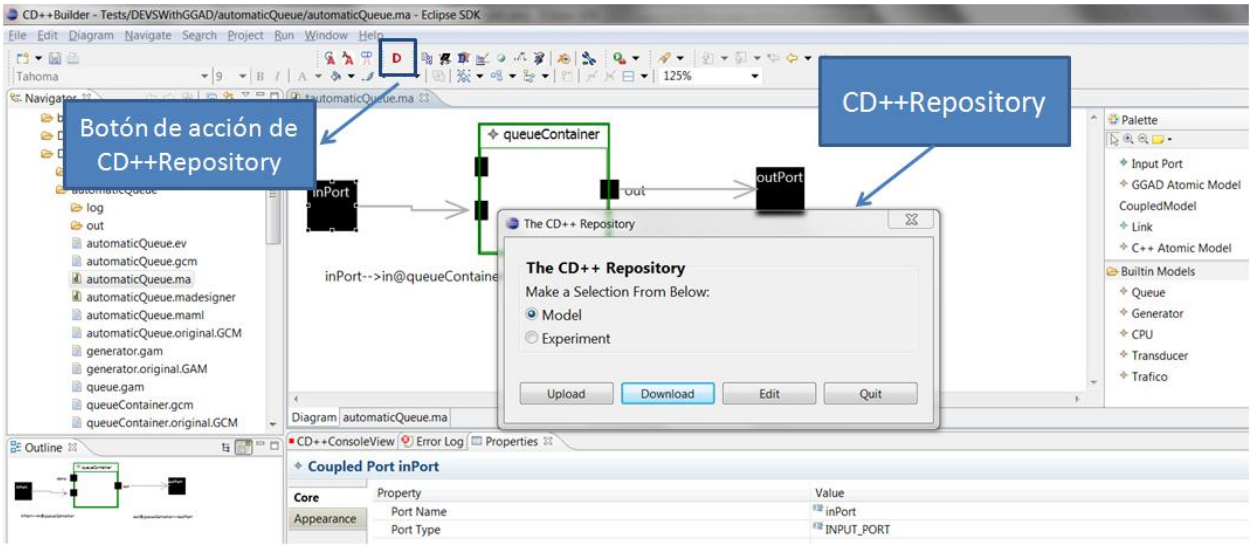


Figura 88 - Integración de CD++Repository en CD++Builder.

4.2.2 Editores gráficos

El objetivo original de los nuevos editores gráficos era proveer las mismas funcionalidades que las herramientas de modelado CD++Modeler y GGADTool, pero integradas dentro de la plataforma de Eclipse. Otro objetivo principal en cuanto a los editores era resolver los problemas de usabilidad general que poseían las herramientas anteriores. A continuación se verán los resultados del desarrollo de CD++Builder 2.0 y cómo estos cumplen con los objetivos planteados.

En la siguiente figura puede verse el nuevo editor de modelos acoplados (Figura 89 a.) y el nuevo editor de modelos atómicos DEVS-Graphs (Figura 89 b.) integrados dentro del entorno de Eclipse:

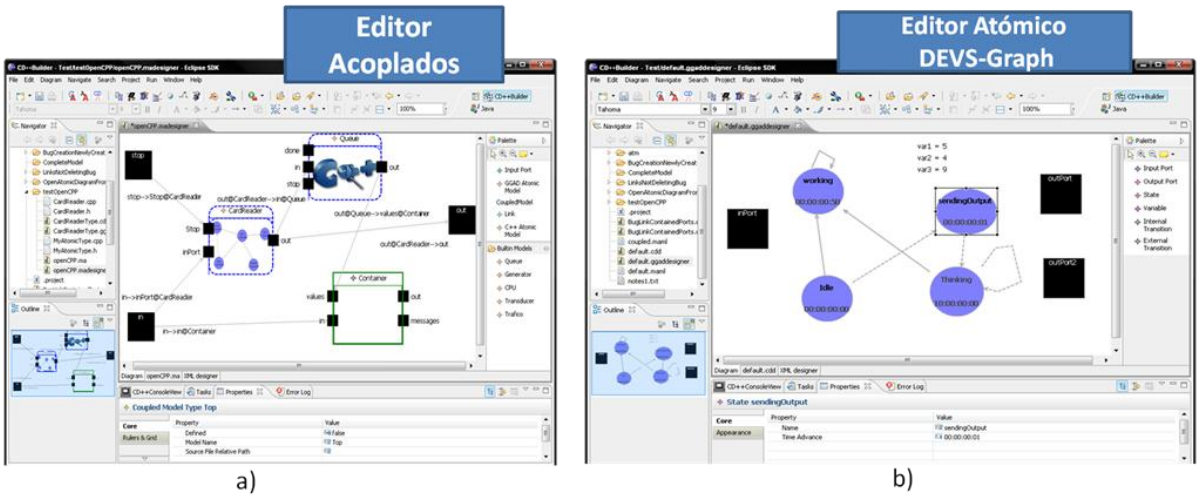


Figura 89 - Editores gráficos de modelos DEVS en CD++Builder2.0

Los nuevos editores pueden abrirse haciendo doble click sobre los archivos en la ventana de navegación, cada editor se muestra en una solapa separada (permitiendo tener varios modelos abiertos al mismo tiempo), pueden crearse nuevos modelos a través de wizards utilizando los menús de Eclipse, utilizan las ventanas estándar de Eclipse para mostrar, por ejemplo, las propiedades, el Outline de los modelos, etc.

Los editores gráficos de CD++Builder 2.0 presentan muchas mejoras en a la usabilidad frente a los editores de CD++Modeler. Distinguiremos a continuación algunas de las mejoras más importantes para los editores de modelos atómicos y acoplados en cuanto a la visualización y accesibilidad:

Mejoras de usabilidad generales:

1. **Zoom in/out**

   Los editores permiten agrandar o achicar el espacio de visualización y edición del modelo. Esto es útil para modelos de gran tamaño que contienen muchos componentes. Para acercar o alejar la vista de los modelos se puede utilizar tanto los botones de zoom del panel de herramientas () o bien presionando la tecla Ctrl y moviendo la ruedita del mouse.

2. **Vista completa del modelo**

   La vista de **Outline** permite visualizar el modelo completo independientemente del zoom que tenga aplicado y mover el espacio de visualización. Esta ventana además puede colocarse en cualquier lugar de la pantalla, mostrarse como una solapa, ocultarse, etc. La siguiente figura muestra la vista de Outline de un modelo acoplado con 3 atómicos y 2 puertos:

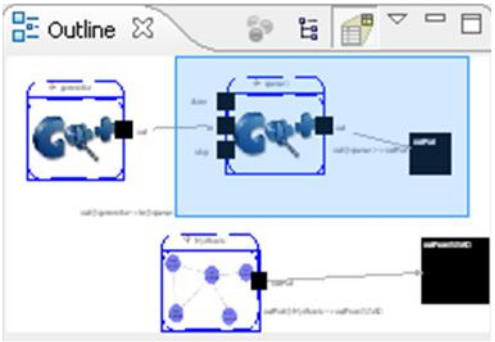


Figura 90 - Overview de modelos en CD++Builder 2.0.

3. Vista y edición de Propiedades

La vista **Properties** permite ver y editar fácilmente todas las propiedades de la entidad seleccionada. Se pueden ver tanto las propiedades referentes al modelo (utilizando la solapa **Core** como muestra la Figura 91 a) o las propiedades de visualización (utilizando la solapa **Appearence** como muestra la Figura 93). Haciendo un click sobre las propiedades que son visibles en el editor (por ejemplo el nombre de los modelos) estas pueden también editarse directamente sobre las figuras, al habilitarse el campo para editarlas como muestra la Figura 91 b). En CD++Modeler la edición de las propiedades era menos intuitivo, ya que había que primero seleccionar la entidad (con doble click), luego hacer click derecho seleccionando *Edit Properties*, y cambiar las propiedades en una nueva ventana emergente como muestra la Figura 91 c).

En CD++Modeler no todas las propiedades están visibles y es necesario abrir una nueva ventana para editarlas.

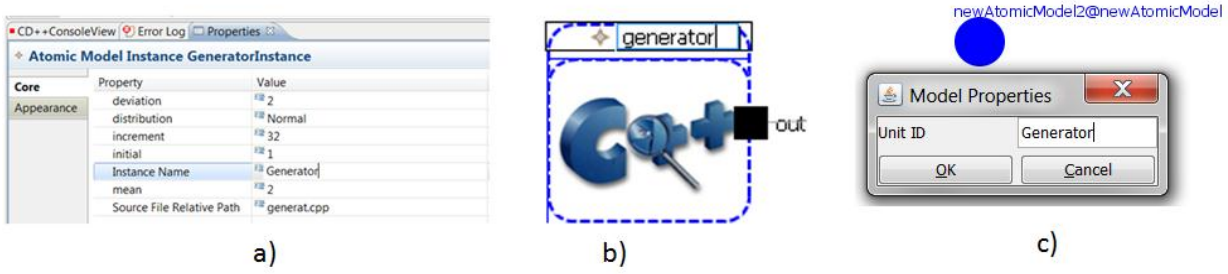


Figura 91 - Vista y edición de propiedades en CD++Builder 2.0 a) mediante la vista de Propiedades b) edición directa en el modelo c) en CD++Modeler.

4. Links mejorados

En CD++Builder 2.0 los links y transiciones internas o externas pueden configurarse para que se muestren de diferentes formas (ver Figura 92 a)). Esto ayuda mucho al modelado ya que en CD++Modeler los links eran fijos y quedaban en muchos casos superpuestos. En la Figura 92 b)

puede verse como las transiciones state1→state2 y state2→state1 quedan superpuestas en CD++Modeler. En la Figura 92 a) puede verse el mismo modelo utilizando CD++Builder 2.0.

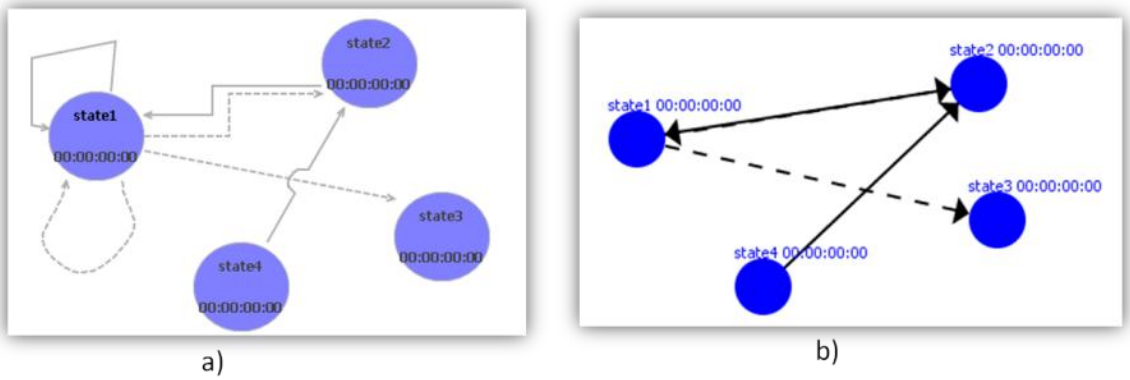


Figura 92 –Comparación de links entre a) CD++Builder2.0 y b) CD++Modeler

En la figura de la izquierda se muestra cómo se pueden configurar los links para que sean oblicuos (por ejemplo state1→state1), rectilíneos (state2→state1), para que no se superpongan (state1→state2 y state2→state1), y para que "salten" otros links (state4→state2).

5. Visualización configurable de elementos

Todos los elementos gráficos de los editores pueden ser configurados flexiblemente para que se muestren de diferente forma. Se permite cambiar el tamaño de los elementos, el tipo y color de letra en que se muestran sus propiedades, y para ciertos elementos otras propiedades específicas. Por ejemplo la Figura 93 muestra las propiedades de apariencia de las transiciones que pueden ser configuradas como se explicó anteriormente.

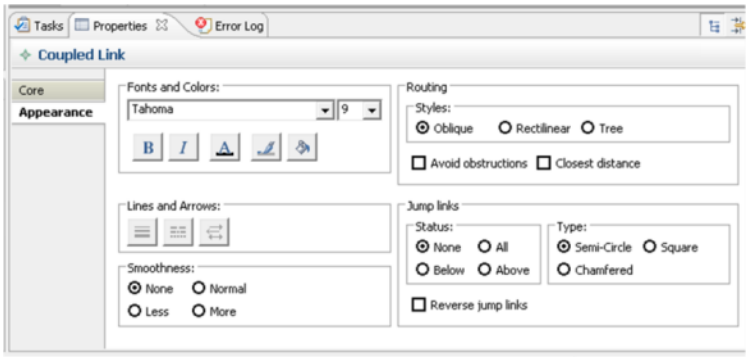


Figura 93 - Propiedades de apariencia en CD++Builder 2.0.

En CD++Modeler sólo existe la posibilidad de configurar una imagen personalizada para cada modelo. Esto no está soportado aún por CD++Builder 2.0 y se propone como mejora para trabajos futuros en el próximo capítulo.

6. Comandos habituales y teclas de acceso rápido

Los editores poseen comandos como borrar, deshacer, rehacer, copiar, pegar, etc., soportados para todos los elementos gráficos. Estos comandos pueden ejecutarse tanto del menú **Edit** de Eclipse, desde los menús contextuales que se despliegan al hacer click izquierdo sobre un elemento o bien con las teclas de acceso rápido habituales (Ctrl-c, Ctrl-v, Ctrl-z, etc.). CD++Modeler solo soporta el comando de borrado.

Más allá de proveer todas las características de los editores anteriores con algunas mejoras de usabilidad, los editores desarrollados también proporcionan nuevas funcionalidades importantes para modelar. A continuación se detallan algunas de las nuevas funcionalidades del editor para modelos acoplados y luego del editor para modelos atómicos.

Nuevas funcionalidades del editor de modelos acoplados:

1. **Puertos internos visibles individualmente**

El editor de modelos acoplados muestra los puertos de los modelos atómicos y acoplados que contiene. En CD++Modeler los componentes se muestran como figuras circulares (modelos atómicos) o rectangulares (modelos acoplados), pero no se muestran los puertos que éstos contienen (Figura 94 a). En la figura de la izquierda puede verse como en CD++Builder 2.0 se muestran todos los puertos de los componentes.

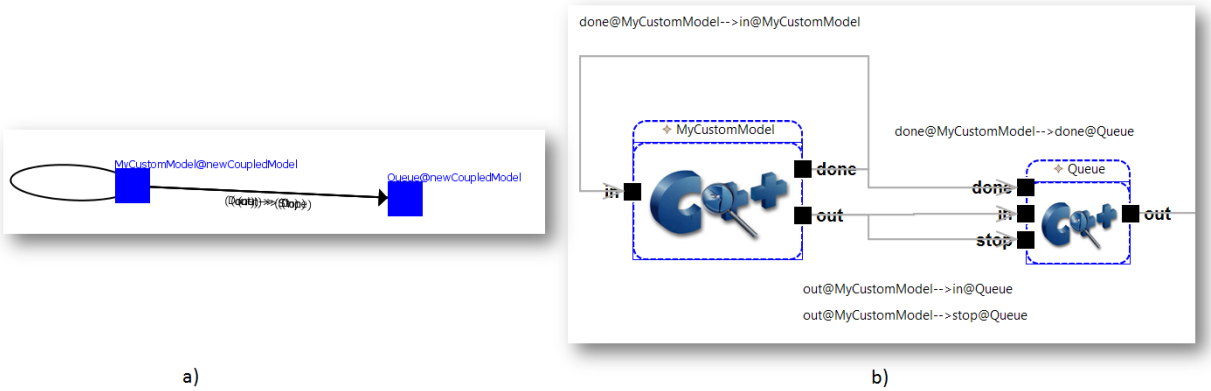


Figura 94 - Visualización de componentes con puertos en a) CD++Modeler b) CD++Builder 2.0

Esta forma de visualizar los componentes posee 2 grandes ventajas respecto de la anterior. Por un lado simplifica mucho la creación de links entre componentes, ya que se realizan directamente con el mouse, de un puerto al otro. En CD++Modeler es necesario primero crear el link entre los componentes y luego editar el link para especificar a qué puertos de cada uno de los modelos está relacionado el link. Esto a su vez facilitó en CD++Builder 2.0 la validación, ya que directamente no se permite crear links inválidos (por ejemplo de un puerto de entrada a otro de entrada). En CD++Modeler la validación se realiza recién al exportar el modelo.
Por otro lado, al mostrar los puertos individuales que posee cada componente, se facilita la comprensión del modelo ya que no sólo se ve cuáles componentes están relacionados, sino por

medio de cuales señales individuales lo hacen. En CD++Modeler no se puede distinguir visualmente si dos componentes se relacionan a través de 1 o más puertos, se solapa el texto de los links, etc.

2. Navegación de la jerarquía de modelos

CD++Builder 2.0 permite abrir, desde el editor de modelos acoplados, un nuevo editor para los submodelos. Ambos editores se mantienen sincronizados permitiendo editar un modelo (contenido dentro de otro modelo) sencillamente. CD++Modeler permite esta navegación, pero al editar un submodelo se pierden todos los links que éste tenía con otros componentes, haciendo la navegación jerárquica prácticamente prohibitiva.

3. Modelos atómicos C++

Una de las funcionalidades nuevas más importantes es la posibilidad de crear modelos acoplados gráficamente que contengan modelos atómicos especificados en C++. Tanto CD++Modeler como GGADTool permitían crear únicamente modelos atómicos DEVS-Graphs.

El editor de modelos acoplados posee un parser simple que identifica los puertos y parámetros declarados en código C ++ de los modelos atómicos y permite mostrarlos en forma gráfica. Estos puertos y parámetros se sincronizan automáticamente al realizarse un cambio en los archivos C++ permitiendo editar el código y seguir visualizando los modelos gráficos consistentemente.

Un punto importante en cuanto a los modelos C++ es que si los archivos de especificación no existen, al hacer doble click sobre el modelo se crean nuevos archivos que poseen el esqueleto de código para implementar un modelo atómico utilizando el framework CD++. La Figura 95 muestra el código generado para un modelo atómico C ++. Esto no solo simplifica y agiliza la creación de modelos atómicos evitando pasos repetitivos, sino que promueve buenas prácticas de programación y modelado. Esto último es especialmente importante para los principiantes que utilicen la herramienta, disminuyendo notablemente las barreras de entrada en la adopción de la metodología DEVS. CD++Modeler y la versión anterior de CD++Builder no brindaban ningún soporte para desarrollar modelos atómicos C++.

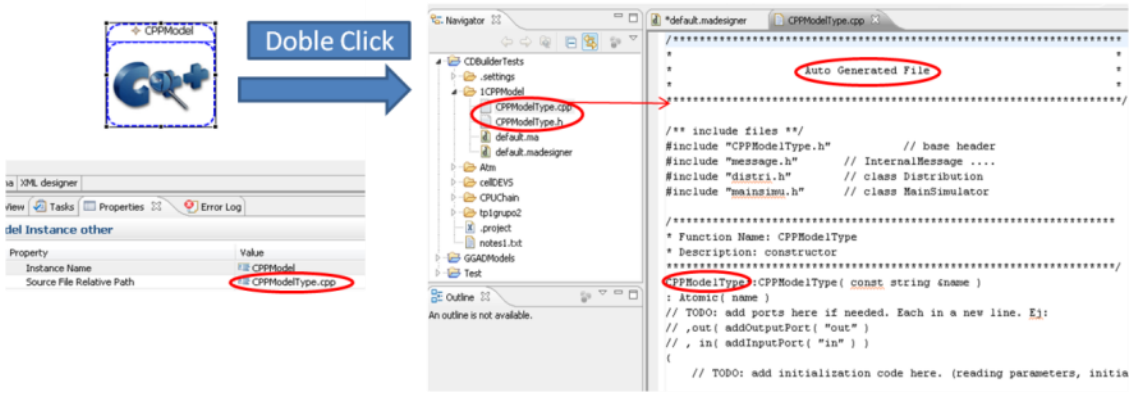


Figura 95 - Código generado automáticamente por CD++Builder 2.0 para modelos atómicos C++

4. Barra de herramientas con modelos reusables

El editor de modelos acoplados de CD++Builder 2.0 provee un panel que muestra otros modelos acoplados y atómicos que pueden reutilizase, y permite agregarlos al modelo en forma sencilla.

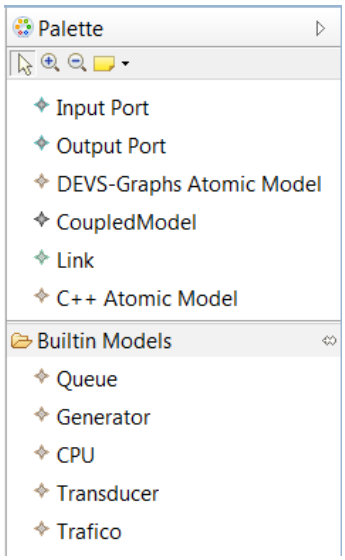
En la 

Figura 96 se muestra el panel del editor de modelos acoplados de CD++Builder 2.0 con las herramientas para crear los elementos del modelo.

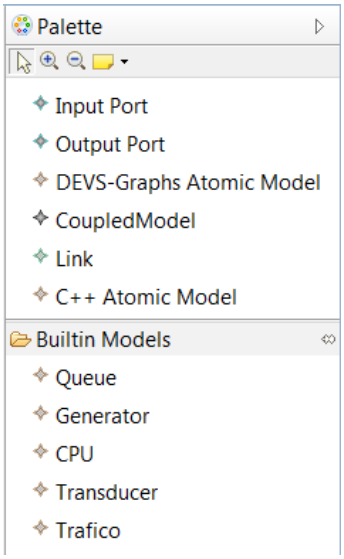


Figura 96 - Panel de herramientas para modelos acoplados CD++Builder 2.0

En la parte inferior del panel de herramientas se muestra un conjunto de modelos que pueden ser añadidos al modelo acoplado que está siendo editado arrastrándolos a la zona de edición. Esta sección es importante ya que brinda no sólo una forma accesible de agregar otros modelos, sino que para usuarios que no están totalmente familiarizados con CD++ les permite conocer qué otros

modelos ya fueron creados. Esto fomenta la reutilización de modelos agilizando el modelado, reduciendo los errores y disminuyendo el esfuerzo de testeo. En las herramientas previas, los modelos incluidos en CD++ no podían utilizarse, ya que no soportan modelos atómicos implementados en C++.

Nuevas funcionalidades del editor de modelos atómicos:

1. Variables y Puertos Visibles

Tanto CD++Modeler como GGADTool permiten crear puertos y variables para los modelos DEVS-Graphs, pero estos no se muestran gráficamente. En CD++Builder 2.0 todos los elementos que forman parte del modelo atómico pueden verse y editarse en el diagrama.

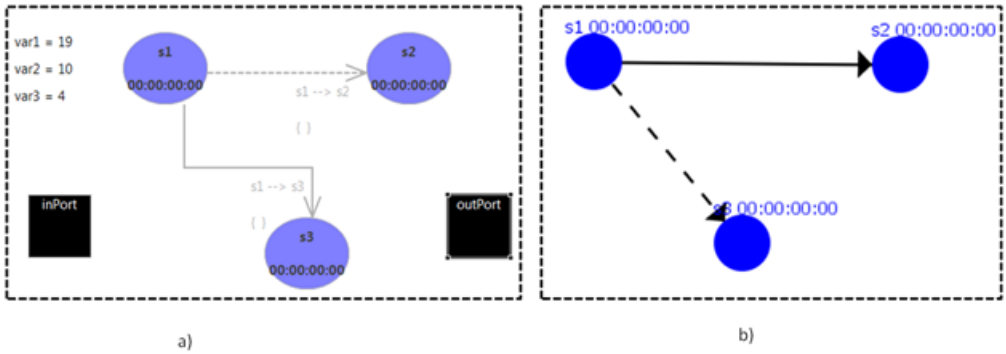


Figura 97 - Modelo DEVS-Graphs con puertos y variables a) en CD++Builder 2.0 b) en CD++Modeler

En la figura de la izquierda puede verse cómo CD++Builder expone una vista completa del modelo DEVS-Graphs, mostrando no sólo los estados y transiciones sino también los puertos de entrada/salida y las variables con sus valores iniciales.

2. Expresiones

Tanto las acciones de las transiciones como la condición de ejecución de las transiciones externas se describen utilizando expresiones, como fue detallado en el capítulo 2. Tanto en CD++Modeler como en GGATool las expresiones de definían como campos de texto, sin ningún tipo de validación (Figura 98 b).

En CD++Builder 2.0 las expresiones que se utilizan en las transiciones forman parte del modelo conceptual (ver capitulo de implementación) y se validan durante el proceso de modelado. La siguiente figura muestra a la izquierda el editor de expresiones que se utiliza en CD++Builder 2.0 para las acciones. Notar que se muestra un mensaje de error si la expresión no es válida:

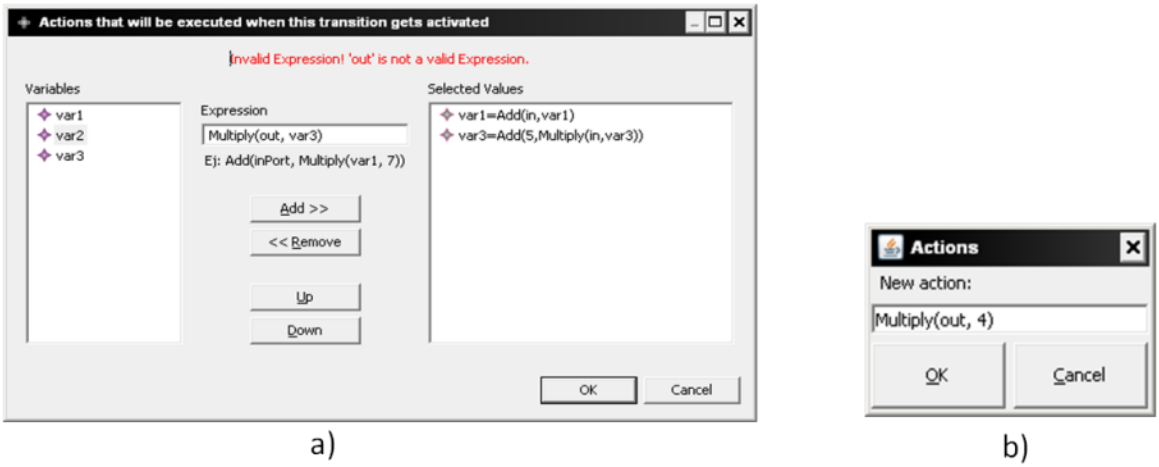


Figura 98 - Editor de expresiones a) en CD++Builder 2.0 b) en CD++Modeler

Nuevas funcionalidades comunes a ambos editores.

Por último respecto de los editores gráficos, se agregaron nuevas funcionalidades comunes a ambos editores (el editor de modelos acoplados y el editor de modelos atómicos). Estas características no estaban disponibles en CD++Modeler ni GGADTool.

1. Posibilidad de abrir modelos existentes

En CD++Builder 2.0, tanto el editor de modelos acoplados como el editor de modelos atómicos permiten mostrar en forma gráfica modelos que fueron creados utilizando sólo la gramática de CD++ y que por lo tanto no poseen originalmente una definición gráfica. CD++Modeler permite abrir o importar exclusivamente modelos que fueron creados utilizando su formato gráfico.

Esta característica es importante ya que CD++ cuenta con un extenso repositorio de modelos desarrollados, cuya gran mayoría no cuenta con una definición gráfica. CD++Builder 2.0 permite abrir la definición de estos modelos y generar automáticamente un modelo gráfico. Este punto es central a la hora de reutilizar modelos y transmitir conocimientos. Si un modelo creado para CD++ puede visualizarse en forma gráfica es mucho más simple de comprender para otros modeladores (especialmente alumnos), ya que acceden a una primer impresión global y visual del modelo, su estructura, componentes e interacciones, facilitando su reutilización en otros sistemas.

Utilizando las herramientas previas, si uno deseaba ver gráficamente un modelo ya existente de CD++, debía crearlo nuevamente desde cero. Al no tener soporte para modelos atómicos C++, muchos modelos debían además traducirse a DEVS-Graphs, y para aquellos en los que su comportamiento fuese imposible de traducir, no existía una herramienta que permita representarlos gráficamente.
En cambio, al abrir un modelo ya existente de CD++ en CD++Builder 2.0 se genera automáticamente la vista gráfica del mismo, independientemente de su comportamiento. La vista gráfica generada que posee todas las características que se mencionaron en este capítulo: navegación jerárquica,

zoom in/out, edición gráfica, etc. Dado que CD++Builder 2.0 soporta tanto modelos DEVS-Graphs como modelos C++ no es necesario realizar ninguna modificación al modelo original.

2. Visualización gráfica y textual

Ambos editores permiten ver tanto la definición gráfica de los modelos como la definición textual (que utiliza la gramática correspondiente de CD++). CD++Modeler sólo mostraba la representación gráfica de los modelos y era necesario exportarlos para que se generen archivos textuales aptos para ser simulados.

En CD++Builder 2.0, para cambiar de la vista gráfica a la textual se utilizan las solapas que aparecen debajo del modelo. La solapa **Diagram (** Diagram **,** Figura 99 a) muestra el modelo en forma gráfica, mientras que la solapa con el nombre del archivo CD++ (default.ma , Figura 99 b) muestra el modelo en forma textual.

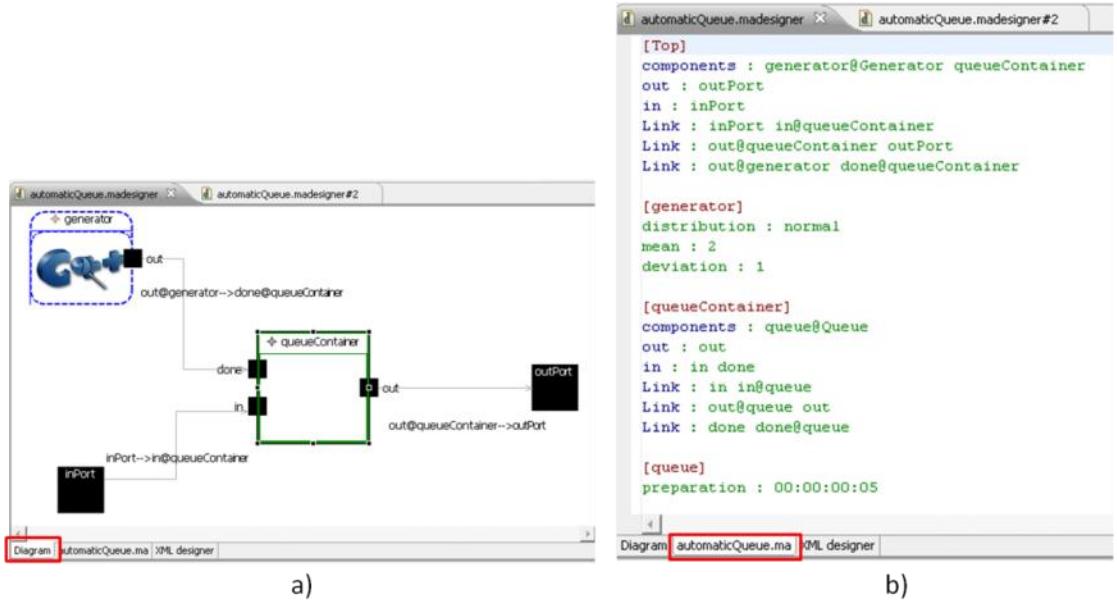


Figura 99 - Comparación de vistas de un mismo modelo en CD++Builder 2.0 a) Vista gráfica b) Vista textual.

Ambas vistas se sincronizan automáticamente al guardar los cambios en cualquier de ellas. Si se realiza una modificación en el modelo gráfico y se guarda la nueva configuración, se regenerará el archivo textual. Si se realiza una modificación en la definición textual del modelo y se guardan los cambios, se regenerará el diagrama. En algunos casos la regeneración del diagrama implica la pérdida de la disposición espacial de los componentes.

4.2.3 Instalación y actualización

Las herramientas existentes previas a CD++Builder 2.0 no contaban con ningún mecanismo de actualización. Esto traía aparejado los problemas que fueron descriptos previamente: problemas entre versiones, usuarios utilizando herramientas inadecuadas, imposibilidad de reportar bugs y sus soluciones, etc. A continuación de describirán a grandes rasgos los resultados de la implementación de

procesos de instalación y actualizaciones automáticas en CD++Builder 2.0. En el **Apéndice A: Manual de Instalación de CD++Builder 2.0**, se pueden ver todos los pasos necesarios para instalar y actualizar CD++Builder 2.0.

En CD++Builder 2.0 se extendieron los mecanismos de actualización automática que provee la plataforma de Eclipse para facilitar la actualización de todas las herramientas relacionadas con CD++. Esto resuelve los problemas encontrados durante el proceso de análisis de forma integrada con el resto del entorno, mediante un mecanismo estándar, flexible y probado.

Para poder comenzar con el entorno CD++Builder 2.0, hay dos posibilidades para su instalación. La primera es la que proveía la versión anterior de CD++Builder, que consiste en descargar una versión de Eclipse con el plugin ya instalado. La segunda posibilidad es más flexible ya que permite instalar el plugin desarrollado en este trabajo en un entorno Eclipse previamente instalado y personalizado. Para esto último se utiliza lo que se denomina en Eclipse como "**Update Manager**". Mediante este componente se pueden encontrar nuevos plugins para instalar. Especificando la dirección de publicación de CD++Builder se puede instalar este plugin en 4 pasos como muestra la siguiente figura:

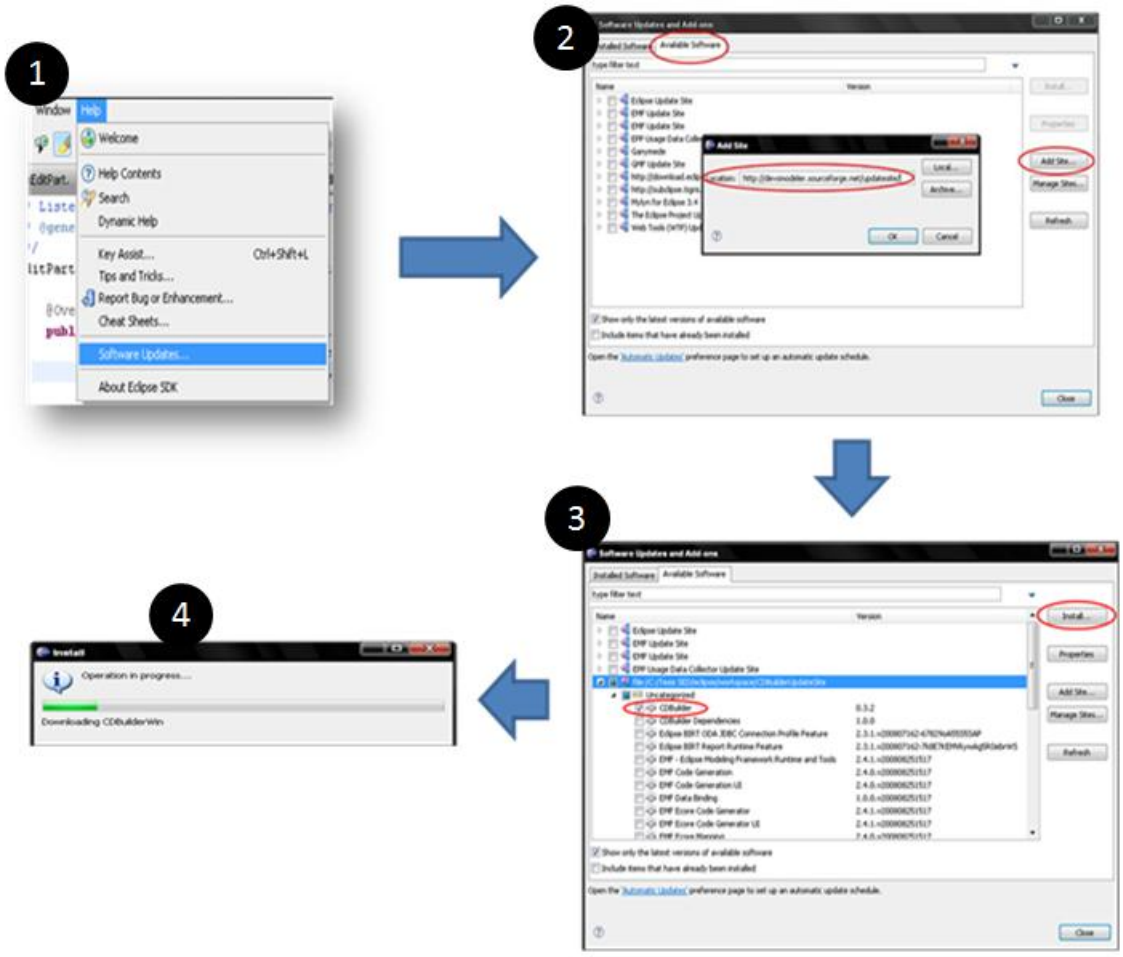


Figura 100 - Proceso de instalación de CD++Builder 2.0 mediante el Update Manager.1) Selección del menú 2) Agregar el update site de CD++Builder 3) Selección del plugin 4) Instalación.

Una vez que el plugin ya está instalado en Eclipse, éste puede ser actualizado automáticamente. De todos modos, sigue siendo posible actualizar el plugin en forma manual remplazando la instalación de Eclipse como se realizaba en la versión anterior. Para actualizar automáticamente el plugin también se utiliza el **Update Manager**, pero en este caso no es necesario especificar nuevamente la dirección de publicación de CD++Builder ya que queda registrada en la instalación. El componente del **Update Manager** verifica si se han publicado nuevas versiones de los plugin instalados y los actualiza de ser necesario. La siguiente figura muestra los 3 pasos necesarios para actualizar automáticamente el plugin:

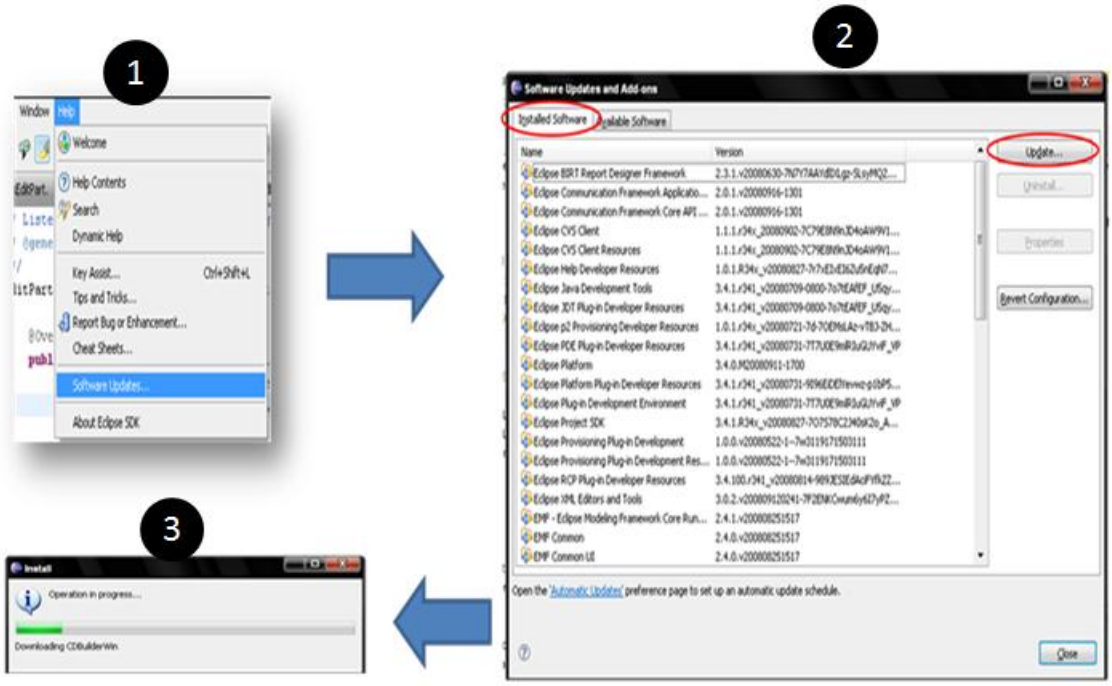


Figura 101 –Proceso de actualización automática de CD++Builder 2.0. 1) Selección del menú 2) Click en Update 3) Actualización.

Update Manager permite incluso configurar Eclipse para que realice el chequeo de actualizaciones de sus plugins en forma periódica y automática. Esto libera completamente al usuario de mantener actualizadas las versiones de CD++Builder y CD++, ya que el entorno verificará y actualizará los plugins automáticamente. La siguiente figura muestra la ventana de configuración para realizar chequeos periódicos de actualizaciones:

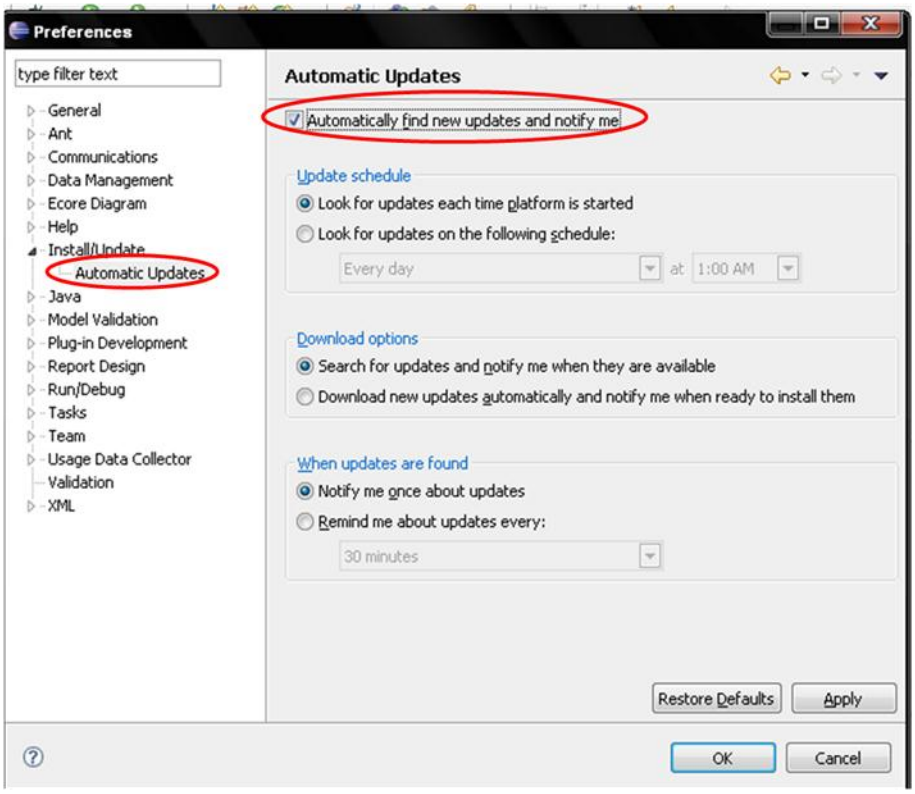


Figura 102 –Chequeo automático de actualizaciones CD++Builder 2.0.

Luego, mediante estos mecanismos es posible para alumnos e investigadores trabajando en mejoras para la herramienta, liberar una nueva versión del plugin o del simulador, resolver bugs o añadir nuevas funcionalidades, asegurándose que todos los usuarios utilizan la herramienta adecuada. Por otro lado, no es necesario para los usuarios estar al tanto de si se liberan nuevas versiones del plugin, permitiéndoles concentrarse en el modelado.

4.2.4 Tests automatizados

Las herramientas anteriores y la versión previa de CD++Builder no poseían tests automatizados. Esto dificultaba extender o modificar su comportamiento y continuar asegurando su correcto funcionamiento.

El desarrollo de CD++Builder 2.0 incluye 138 tests automatizados para las funcionalidades más importantes del plugin. Estos tests están orientados a los usuarios que deseen extender la plataforma, agregando o modificando las características existentes. Los tests provistos con CD++Builder 2.0 pueden ejecutarse para verificar que el plugin se comporta de la forma esperada para un conjunto importante de funciones. De esta manera, ante cambios o nuevas implementaciones, los tests pueden volver a ejecutarse para verificar que el comportamiento sigue siendo el correcto.

Capítulo 5 - Conclusiones

En este trabajo presentamos las características y arquitectura de CD++Builder 2.0, que extiende la versión anterior para proveer un entorno de simulación que permite realizar todas las tareas de M&S e integra las herramientas de CD++ dentro de una única interfaz. CD++Builder está basado en la plataforma de Eclipse y provee un IDE común aprovechando la arquitectura extensible en forma de plugins.

La nueva herramienta desarrollada ahora provee:

- Un entorno integrado de desarrollo (IDE) para todas las tareas de modelado y simulación: modelado, compilación, ejecución de la simulación y análisis de los resultados.
- Editores que permiten realizar el ciclo completo de modelado en forma gráfica.
- Soporte para el desarrollo de modelos atómicos C++, a través de templates de código que ayudan en su implementación y manteniendo los editores gráficos consistentes con el código C++.
- Una plataforma extensible para el desarrollo de nuevas funcionalidades dentro del entorno, que incluye tests de regresión automáticos y un sistema de actualización automática.

Se implementaron editores que permiten crear y modificar gráficamente modelos acoplados DEVS y modelos atómicos DEVS-Graphs directamente desde el entorno de Eclipse, integrando las capacidades de las herramientas anteriores y resolviendo las limitaciones de usabilidad. Al estar integrados los editores gráficos con la interfaz de Eclipse se simplifica el proceso de modelado, brindando una interfaz simple y evitando pasos innecesarios para exportar los modelos de una herramienta a la otra.

Se implementó la capacidad de componer modelos acoplados gráficamente utilizando tanto modelos atómicos DEVS-Graphs como modelos atómicos definidos en C++. Esto brinda mayor flexibilidad para modelar el comportamiento de los sistemas. DEVS-Graphs permite especificar los modelos atómicos más simples gráficamente y mediante el lenguaje C++ se pueden especificar los modelos de comportamiento más complejos. Por otro lado, para facilitar la implementación de modelos atómicos C++ y promover buenas prácticas de programación, se desarrolló la capacidad de generar código C++ automáticamente. El código generado provee la estructura básica para extender la clase **Atomic** de CD++ y comentarios de ayuda con ejemplos de código.

CD++Builder brinda dos posibles vistas para los modelos acoplados y DEVS-Graphs: la vista gráfica y la vista textual (descripta mediante la gramática correspondiente de CD++). Los editores sincronizan automáticamente ambas vistas. Esto le da al usuario la libertad de visualizar y modificar los modelos en cualquiera de las vistas, pudiendo hacer cambios utilizando la misma interfaz, sin tener que exportar diferentes formatos.

La herramienta desarrollada permite también la visualización gráfica de modelos CD++ ya existentes. CD++ cuenta con una gran librería de modelos pero éstos no fueron desarrollados mediante una herramienta visual por lo que no cuentan con una descripción gráfica. La sincronización automática

mencionada anteriormente, junto con la capacidad de incorporar modelos atómicos C++ a la vista gráfica de modelos acoplados, permite proveer automáticamente una visualización gráfica de modelos existentes en el repositorio.

Se implementaron además nuevas funcionalidades, que permiten por ejemplo la reutilización de modelos en forma más sencilla. El editor gráfico de modelos acoplados provee un panel de herramientas, donde pueden encontrarse algunos de los modelos ya desarrollados. Los modelos que se encuentran en el panel de herramientas pueden ser reutilizados arrastrándolos a la sección de edición. Esto le permite al usuario ver fácilmente las funcionalidades ya desarrolladas y utilizarlas en forma sencilla.

En la versión anterior de CD++Builder los usuarios tenían problemas para mantener actualizada la versión del plugin, y al detectarse problemas se hacía difícil la distribución de las soluciones. Para resolver estos problemas, en la versión 2.0 de CD++Builder se utilizaron las herramientas de instalación y actualización provistas por la plataforma de Eclipse. Se desarrollaron *features* y un *update site* que permiten la descarga y actualización de CD++Builder a partir de una dirección de publicación. De esta manera los usuarios pueden actualizar la versión del plugin fácilmente o incluso programar actualizaciones periódicas automáticas. Por otro lado, si se desarrollan nuevas funcionalidades o se solucionan problemas en la versión actual de la herramienta, no es necesario redistribuir los paquetes personalmente a cada usuario. Las actualizaciones se instalan en un único sitio de publicación, y luego se descargan en los clientes automáticamente.

Por último, CD++Builder 2.0 fue concebido para que pueda ser extendido y reutilizado en futuros desarrollos. Es por esto que se han utilizando en el desarrollo de CD++Builder 2.0 tecnologías y frameworks estándares, patrones de arquitectura conocidos, que facilitan la comprensión y reutilización de la implementación.

Para facilitar la tarea de extender el comportamiento de la plataforma a futuros desarrolladores, se implementaron tests automatizados que verifican el correcto funcionamiento de las características más importantes. Al contar con tests automáticos es posible modificar parte del código y de la lógica de la herramienta, ya sea para resolver problemas o para agregar nueva funcionalidad. También permite corroborar que el resto de los módulos se siga comportando como se espera. Los tests automáticos también brindan una mayor seguridad en cuanto a la calidad del desarrollo, disminuyendo la posibilidad de errores.

Se espera que este trabajo brinde la base para nuevas implementaciones. Se han implementado varias características que ofrecen la plataforma para desarrollar nuevas funcionalidades. Es posible incorporar nuevas operaciones a los editores gráficos, extender los generadores de código y reutilizar los mecanismos de actualización e instalación provistos.

Capítulo 6 - Bibliografía

- [AWG03] Ameghino, J.; Wainer, G.; Glinsky, E. “Applying Cell-DEVS in Models of Complex Systems”. Proceedings of the SCS Summer Computer Simulation Conference. Montreal, Canada, 2003.
- [Bec03] Beck, K. “Test-Driven Development”, Addison-Wesley, 2003.
- [BV02] Bolduc, J.S.; H. Vangheluwe, “A modeling and simulation package for classic hierarchical DEVS”. Technical report, Modelling, Simulation and Design Lab (MSDL), School of Computer Science, McGill University, 2002.
- [CDW04] Christen, G.; Dobniewski, A.; Wainer, G. “Modeling state-based DEVS models CD++”. Proceedings of Advanced Simulation Technologies, Arlington, VA, 2004.
- [CW06] Cidre, J. I.; Wainer, G. “Un nuevo entorno integrado para el desarrollo de modelos DEVS”, Tesis de Licenciatura. Departamento de Computacion FCEyN - UBA, 2006.
- [CW07] Chidisiuc, C.; Wainer, G. “CD++Builder: An Eclipse-Based IDE for DEVS Modeling”. Proceedings of SpringSim 2007. Norfolk, VA. USA, 2007.
- [CW09] Chreyh, R.; Wainer, G. “CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames”, Proceedings of SpringSim’09, San Diego, CA, 2009.
- [DCW03] Dobniewski Alejandro, Christen Gastón, Wainer Gabriel, “Representación gráfica de modelos DEVS y modificaciones a CD++ para su simulación”, Tesis de Licenciatura, Departamento de Computación, FCEyN - UBA, 2003.
- [DVW01] Díaz, A.; Vázquez, V.; Wainer, G. “Application of the ATLAS language in models of urban traffic”. Proceedings of the Annual Simulation Symposium. Seattle, USA, 2001.
- [ECL10] Eclipse Foundation. Eclipse versión 3.4.0 en http://www.eclipse.org/. [Accedido el 25 de Marzo de 2010].
- [EEHT05] Ehrig, K; Ermel, C; Hansgen, S; Taentzer, G. 2005. “Generation of visual editors as Eclipse plug-ins”. 20th IEEE/ACM International Conference on Automated software engineering, Long Beach, CA, USA.
- [EFCDT10] Eclipse Foundation. "CDT - Eclipse C/C++ Development Tooling - Version 6.0", en http://www.eclipse.org/cdt/. [Accedido en 25 de Marzo de 2010].
- [EGEF10] Eclipse Consortium. 2009. “Eclipse Graphical Editing Framework (GEF) - Version 3.4”, available at http://www.eclipse.org/gef. [Accedido el 25 de Marzo de 2010].
- [EGMF10] Eclipse Consortium. 2009. “Eclipse Graphical Modeling Framework (GMF)” http://www.eclipse.org/gmf. [Accedido el 25 de Marzo de 2010].
- [FDB02] Filippi, J-B.; Delhom, J.; Bernardi, F. “The JDEVS Environmental Modeling and Simulation Environment,” Proceedings of the first Biennial Meeting of iEMSs. Lugano, Switzerland, 2002.
- [GG96] Ghosh, S.; Giambiasi, N. "On the need for consistency between the VHDL language constructions and the underlying hardware design". Proceedings of the 8th. European Simulation Symposium. Genoa, Italy. Vol. I. pp. 562-567. 1996.
- [GW02] Glinsky, E.; Wainer, G. “Definition of RT simulation in the CD++ toolkit”. Proceedings of the SCS Summer Computer Simulation Conference. San Diego, USA, 2002.
- [GW08] Gutierrez-Alcaraz, M.; Wainer, G. “Experiences with the DEVStone benchmark”, Proceedings of the 2008 Spring simulation multiconference, 2008.

- [JK06] Janousek, V.; Kironsky, E. "Exploratory Modeling with SmallDEVS", Proc. of ESM 2006, Toulouse, France, 2006.
- [JWBC09] Jafer, S.; Wainer, G.; Bonaventura, M.; Castro, R. "Algoritmos Conservadores Para Simuladores Achatados DEVS y Cell-DEVS", Simposio de High Performance Computing, Mar del Plata, Argentina, 2009.
- [KSE09] Kim, S.; Sarjoughian, H.; Elamvazhuthi, V. "DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring", Spring Simulation Multi-conference, San Diego, CA, USA. 2009.
- [Nut] Nutaro, J., "ADEVS (A Discrete EVent System simulator)", Arizona Center for Integrative Modeling & Simulation (ACIMS), University, http://www.ece.arizona.edu/nutaro/index.php.
- [PLK03] Pagliero, E; Lapadula, M; Kofman, E. "PowerDEVS. An Integrated Tool for Discrete Event Simulation". (in Spanish). Proceedings of RPIC, San Nicolas, Argentina, 2003.
- [PP93] Praehofer, H.; Pree, D. "Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs". 25th Winter Simulation Conference, Los Angeles, CA, 1993.
- [QDRT07] Quesnel, G.; Duboz, R.; Ramat, E.; Traoré, M. "VLE: a multimodeling and simulation environment". Proceedings of Summer Computer Simulation Conference. San Diego, CA, 2007.
- [SE09] Sarjoughian, H.; Elamvazhuthi, V. "CoSMos: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation". Proceedings of SIMUTools 2009, Rome, Italy, 2009.
- [SS04] Sarjoughian, H.; Singh, R. "Building Simulation Modeling Environments Using Systems Theory and Soft-ware Architecture Principles". Proceedings of the Advanced Simulation Technology Conference, 235-240, Washington DC, USA. 2004.
- [SSE09] Sungung, K.; Sarjoughian, H.; Elamvazhuthi, V. "DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring". Spring Simulation Multi-conference, San Diego, CA, 2009.
- [ST06] Shatalin, A; Tikhomirov, A. "Graphical Modeling Framework Architecture Overview", Eclipse Modeling Symposium, 2006.
- [SZ98] Sarjoughian, H; Zeigler, Bernard P. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment". Proceedings of the International Conference on Web-based Modeling & Simulation, San Diego, CA, 1998.
- [TW03] Troccoli, A.; Wainer, G. "Implementing Parallel Cell-DEVS". Proceedings of 36th IEEE/SCS Annual Simulation Symposium. Orlando, USA. 2003.
- [Wai03] Wainer, A. Gabriel, "Metodologías de modelización y simulación de eventos discretos", Nueva Librería, 2003.
- [Wai02] Wainer, G. "CD++: A Toolkit to Define Discrete Event Models". Software - Practice and Experience, Vol. 32, No.13, (November): 1261-1306, 2002.
- [Wai09] Wainer, G. "Discrete-Event Modeling and Simulation: a Practitioner's approach". CRC Press, 2009.
- [WDSW01] Wainer, G.; Daicz, S.; De Simoni, L.; Wasserman, D. "Using the ALFA-1 simulated processor for educational purposes". ACM Journal on Educational Resources in Computing. vol. 1, no. 4. pp. 111-151, December 2001.
- [Wol86] Wolfram, S. "Theory and applications of cellular automata". Vol. 1. Advances Series on Complex Systems. World Scientific. Singapore. 1986.
- [XMI98] OMG/XMI: XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998.

- [YW07] Yinfeng, Henry Yu; Wainer, G. “eCD++: an engine for executing DEVS models in embedded platforms”, Proceedings of the 2007 summer computer simulation conference, 2007.
- [Zei76] Zeigler, Bernard P. "Theory of modeling and simulation". Wiley, 1976.
- [Zei84] Zeigler, Bernard P. "Multifaceted Modeling and Discrete Event Simulation", Academic Press, London U.K, 1984.
- [Zei90] Zeigler, Bernard P. "Object Oriented Simulation with Hierarchical, Modular Models". Academic Press, San Diego, California, 1990.
- [ZKP00] Zeigler, Bernard P.; Kim, T.; Praehofer, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems", Academic Press, 2000.
- [ZS03] Zeigler, Bernard P.; Sarjoughian, H. “Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models”, Technical Document, University of Arizona, 2003.

Apéndice A: Manual de Instalación de CD++Builder 2.0

El siguiente es el manual de instalación del plugin CD++Builder 2.0, con los pasos detallados para descargar y comenzar a utilizar el software. . El manual puede descargarse junto con el software de http://sourceforge.net/projects/cdppbuilder/files/.

Installing CD++Builder 2.0 Plug-in under Windows

If you find any problem installing or using CD++Builder you can report it using the Bug Tracker. Please verify the issue is not already reported before submitting a new one.

First step is to install the base requirements. Then, there are 2 ways of installing CD++Builder 2.0 plug-in in Eclipse: Downloading a full Eclipse or Installing from update manager.

Remember that it is always recommended to update and configure Eclipse to work with the last version of the tool.

Installing Prerequisites

To install CD++Builder2.0 you will first need to install all the software provided by the CD++ for Eclipse 3.3 installer.

The installer comes with a guideline (it's copied to **c:\CD++Builder Toolkit Automatic Installation Guide.pdf** when the installer is run) that explains the detailed steps.

After the installation it is important to:

1. Follow the instructions in the section **11. After installation tasks.**
2. Follow **ALWAYS** the instruction in section *"The following next step must be follow ONLY if you have install Cygwin MANNUALLY "***.** You will have to change the profile file as described there (there is a subtle error in the file where you will need to change **HOME="/cygwin/c"** for **HOME="/cygdrive/c".**
3. It might be needed to run the **c:\cygwin\setup.exe** with the **Install from local Directory** option (then **Next**, **Next**, **Next**, ...).
4. Delete the eclipse folder that is created during the installation (by default it is located in **c:\eclipse**). A newer version of Eclipse will be installed in the following steps.

Installing CD++Builder 2.0

Downloading a full Eclipse

The first and simpler way (though it takes more time to download), is to download the last Eclipse provided by the cathedra located in:

https://sourceforge.net/project/showfiles.php?group_id=235328 (this URL might change)

Once you get the zip file, just unzip in the folder you want Eclipse to be installed.

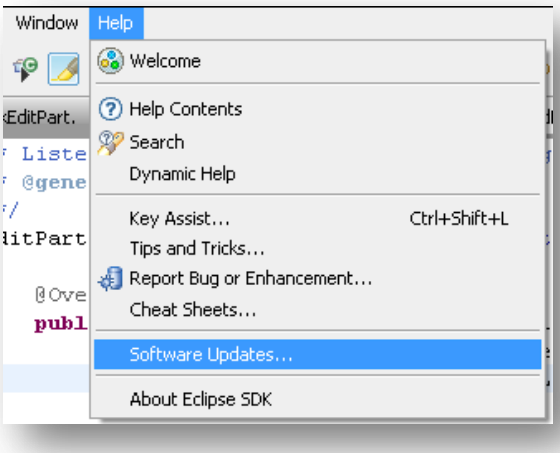
This download includes a workspace with examples of DEVS and DEVS-Graphs models ready to be used. This workspace is located in **%eclipseInstallationDir%\CD++Builder Example Workspace.** To use it, open Eclipse and select it from the **File --> Swith Workspace --> Other... .** Then import the projects contained in that folder (**File --> Import --> Existing Projects into workspace** and select all the projects in **%eclipseInstallationDir%\CD++Builder Example Workspace).**

IMPORTANT: Remember to run the update (see Updating Eclipse) to work with the last version of the tool.

Installing from update manager

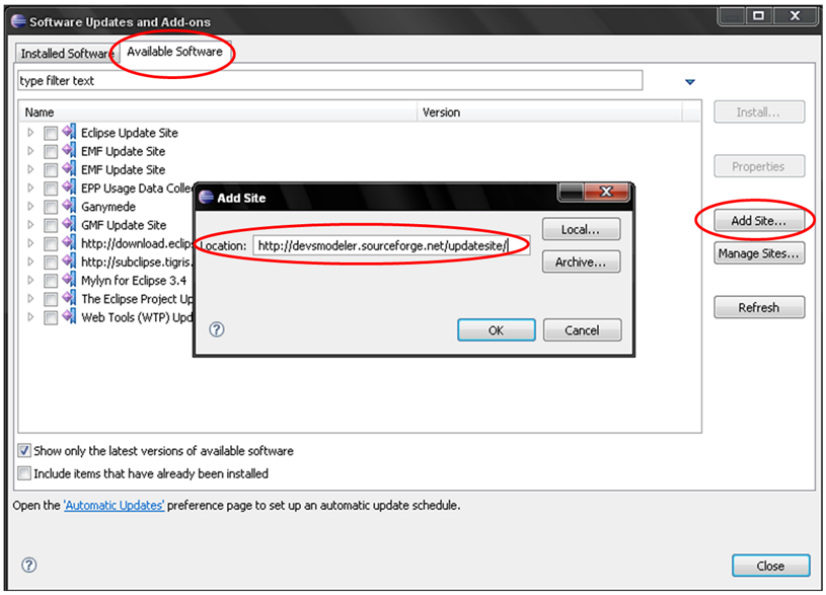
If you already have a working Eclipse, you can install CD++Builder 2.0 Plug-in using Eclipse update manager as follows:

1. In Eclipse go to **Help** -> **Software Updates**...

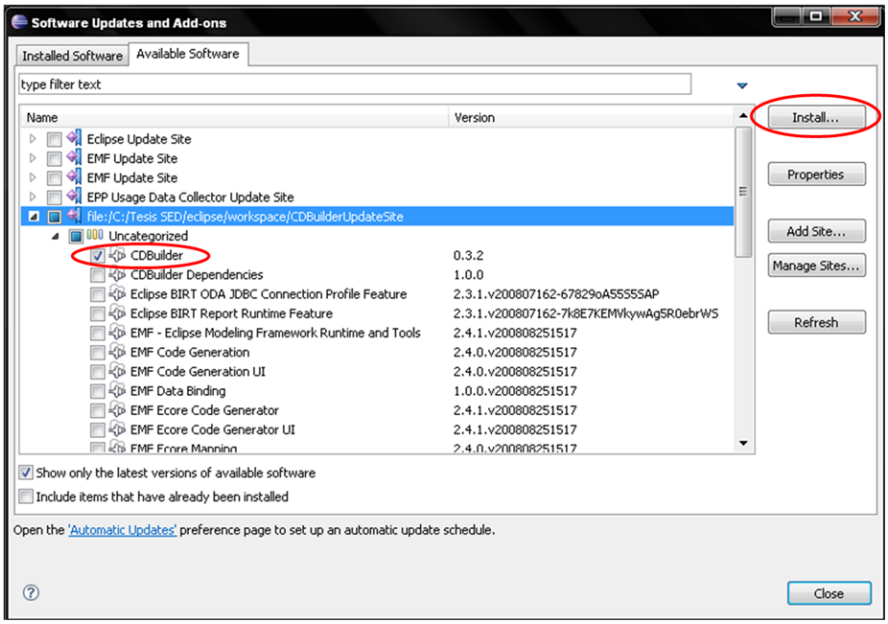


1. The **Software updates and add-on** will appear.
   Go to the **Available software** tab and click **add site.**

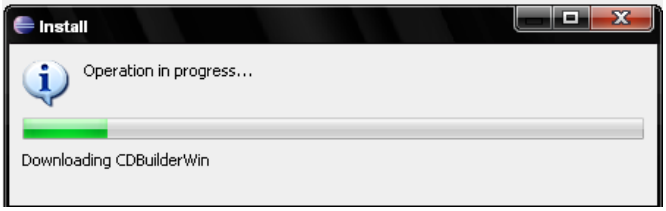
2. The **add site wizard** will appear.
   In the **Location** field write the following URL: http://cdppbuilder.sourceforge.net/updatesite/
   (this URL might change, contact Gabriel if it does not work)



3. Click OK
4. Select the **CD++Builder** category inside the newly added site and click **Install**.



5. **Accept** the license and click **finish** to start with the installation.



(**Note:** this operation can take 5-10 mins depending on the internet connection)

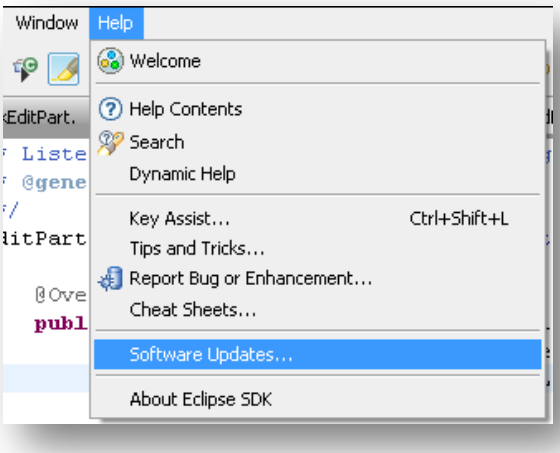
6. Restart Eclipse.

NOTE: It is recommended that you configure Eclipse to run Automatic Updates frequently (see Configure Automatic Updates) so that you work with the last version of the tool.

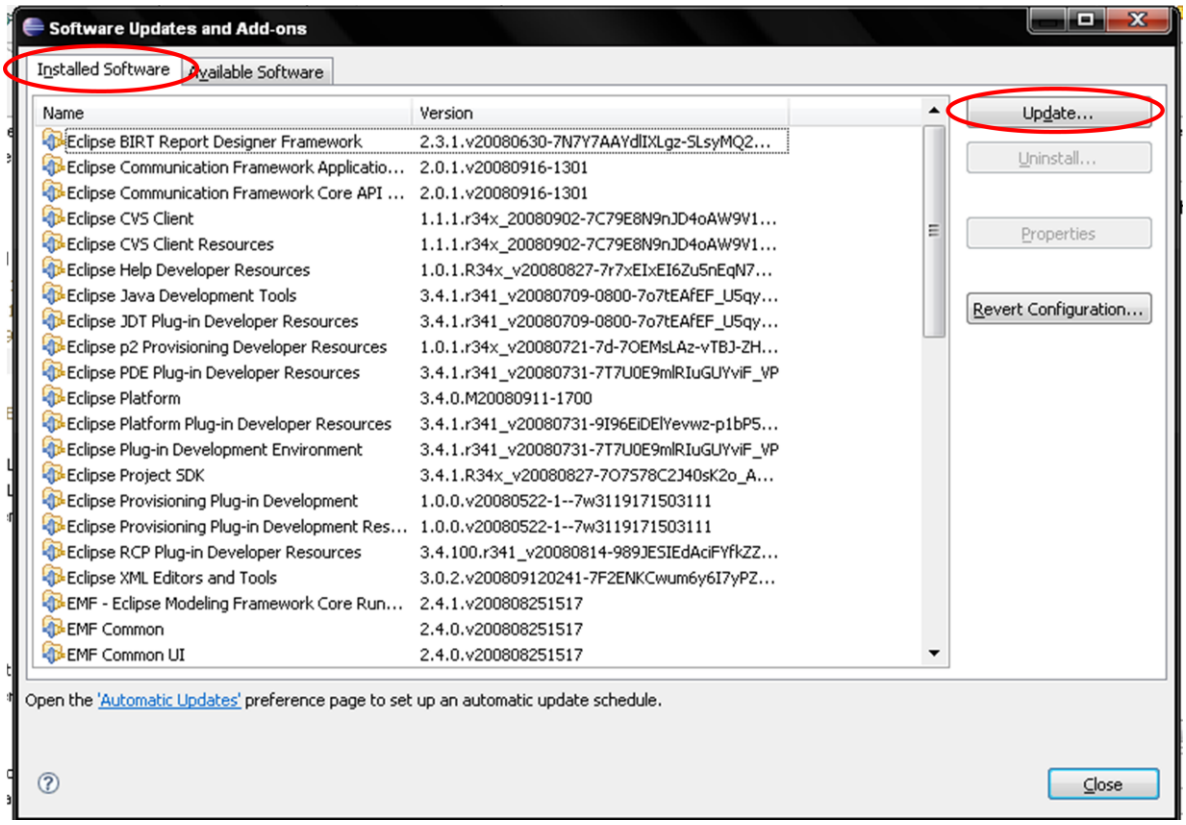
Keeping Updated

Updating Eclipse

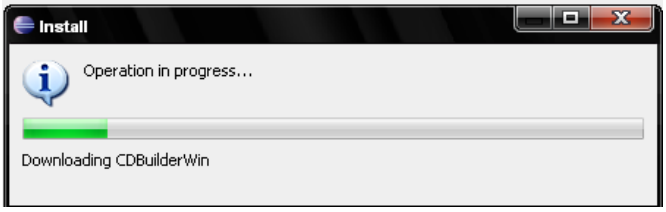
1. In Eclipse go to **Help** -> **Software Updates**…



2. The **Software updates and add-on** will appear.
   In the **Installed Software** tab click **Update**



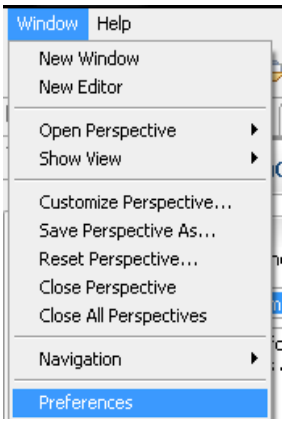
3. Click **Finish** to start updating Eclipse



(**Note:** this operation can take 5-10 mins depending on the internet connection)

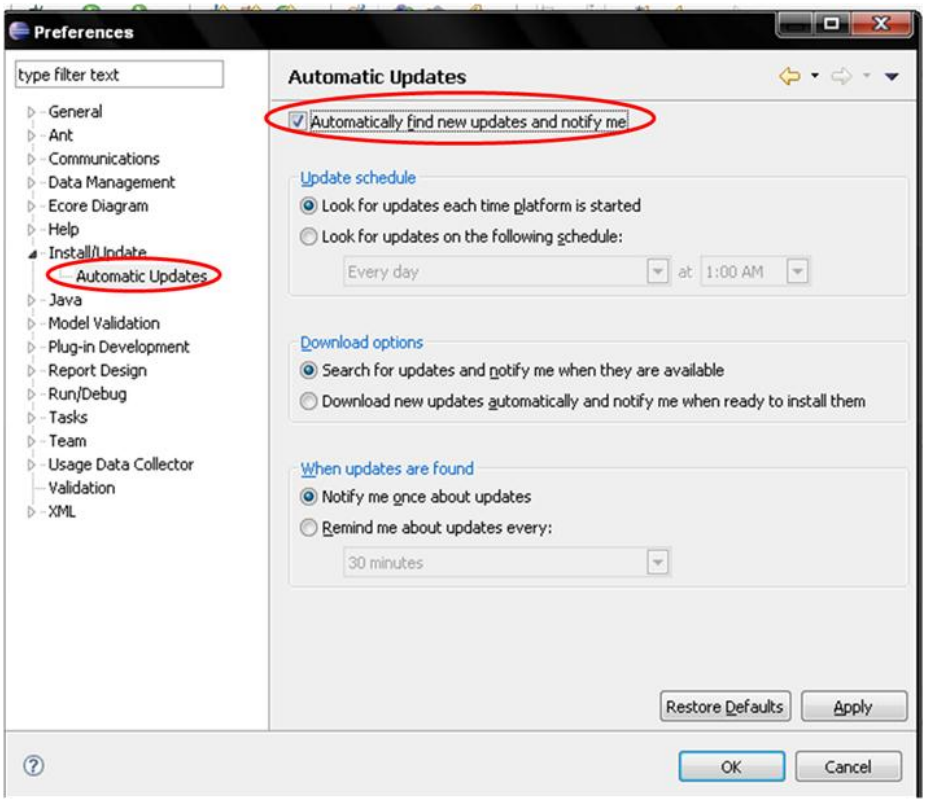
Configure Eclipse Automatic Updates

1. In Eclipse go to **Window** -> **Preferences.**



2. Select **Automatic Updates** from the right menu inside the **Install/Update** category.

Check Automatically find new updates and notify me



Apéndice B: Manual de usuario de CD++Builder

El siguiente es el manual de usuario de CD++Builder 2.0, con los pasos detallados para utilizar las principales funcionalidades del nuevo plugin. El manual puede descargarse junto con el software de http://sourceforge.net/projects/cdppbuilder/files/.

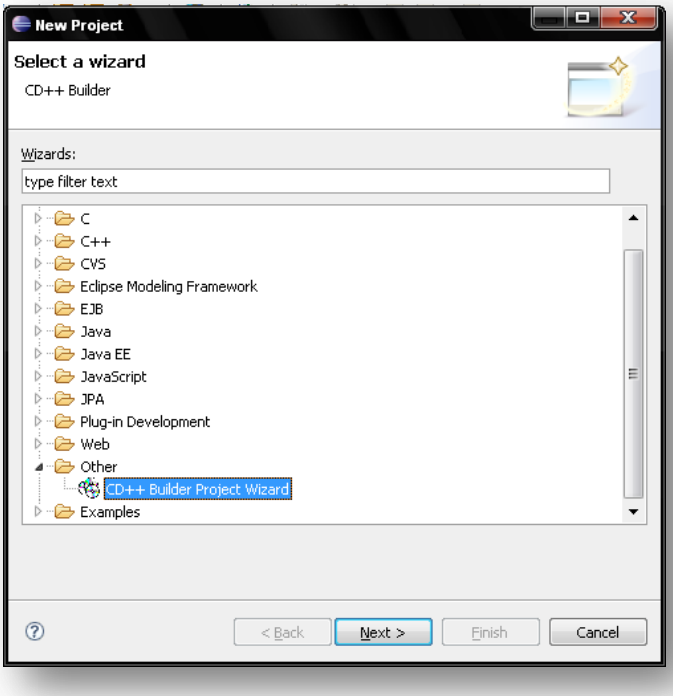
CD++Builder User Manual

This manual is only available in spanish.

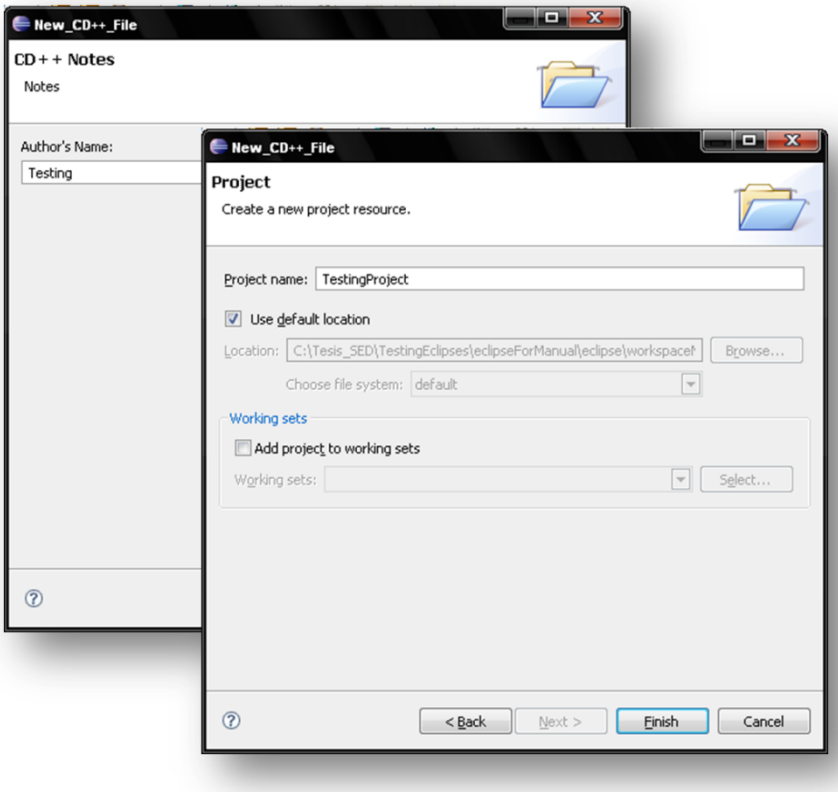
Si tiene problemas utilizando CD++Builder 2.0, verifique en el tracker de bugs si se ha resuelto el problema o para cargar nuevas fallas. Bug Tracker: http://sourceforge.net/tracker/?group_id=235328&atid=1095931

Crear un proyecto en Eclipse

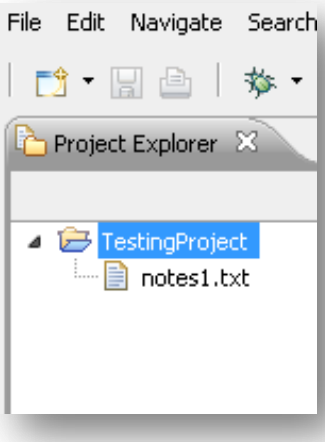
1. Arrancar el Eclipse
2. Ir a **File | new | Project …**
3. En la categoría **DEVS** elegir **CD++ Builder Project Wizard** y hacer click **Next .**



4. Poner un nombre de autor y un nombre para el proyecto (para estos ejemplos **TestProject**)



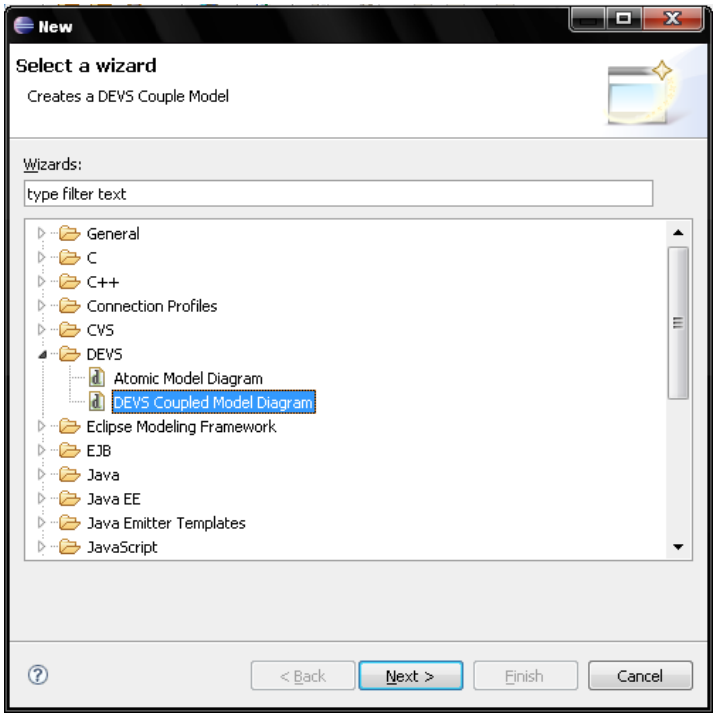
1. Hacer click en **Finish**.
   Esto genera un proyecto de Eclipse



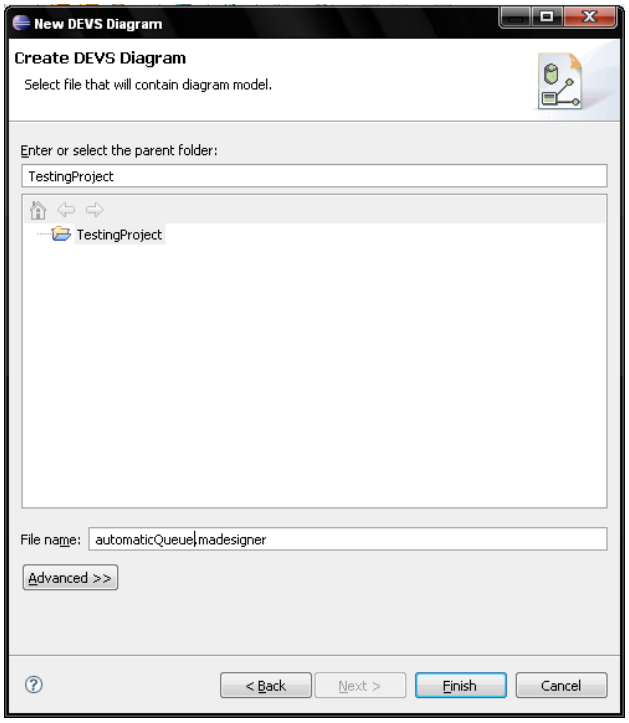
Crear un modelo acoplado

Vamos a crear un modelo que sea una cola que emita los valores que contiene encolados 1 vez por segundo.

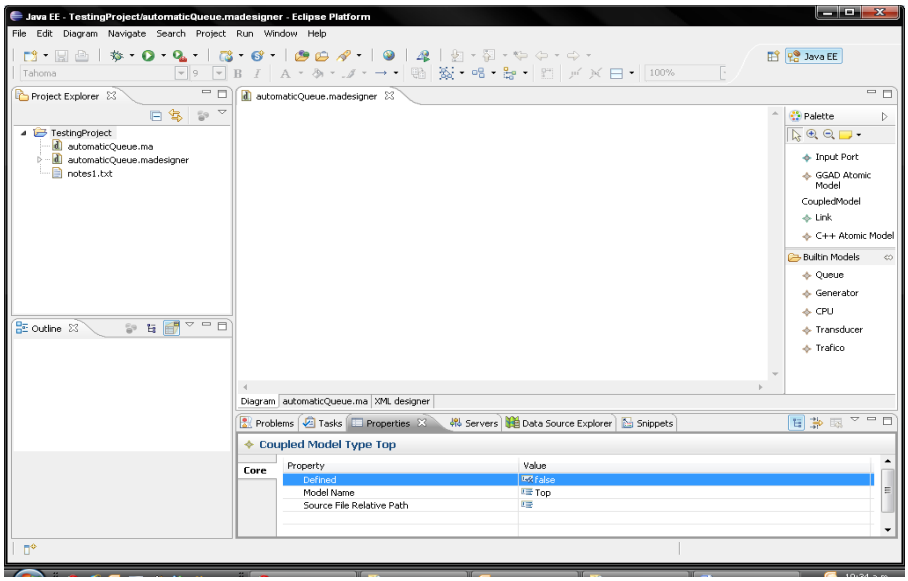
1. Click derecho sobre el proyecto **TestProject** , **new | Other** para abrir el wizard (también desde **File | new | Other)**
2. En la categoría **DEVS** elegir **DEVS Coupled Model Diagram** y hacer click **Next**



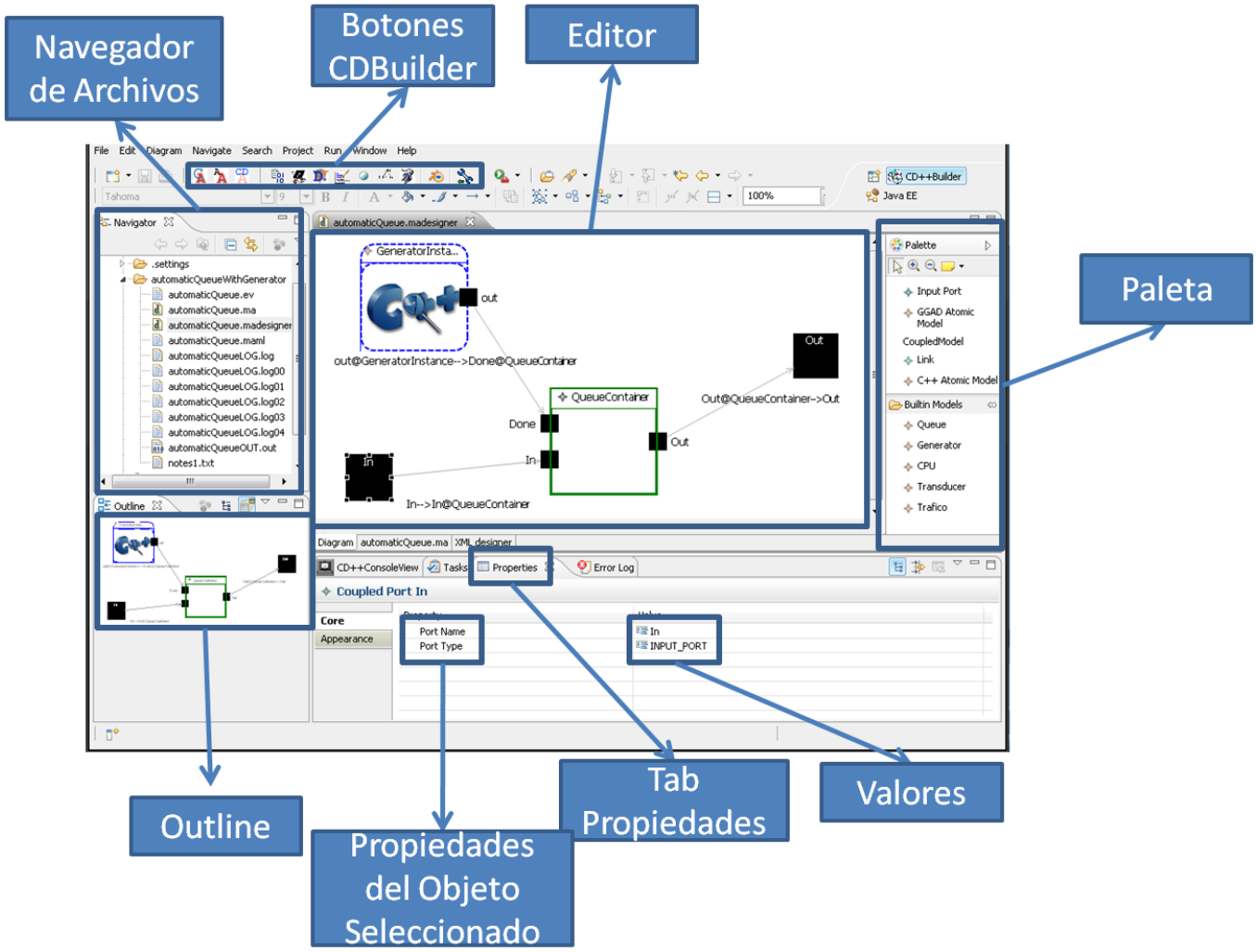
3. Darle un nombre al modelo y hacer click **Finish (automaticQueue.madesigner** en nuestro caso)



4. Esto genera un modelo acoplado vacio.

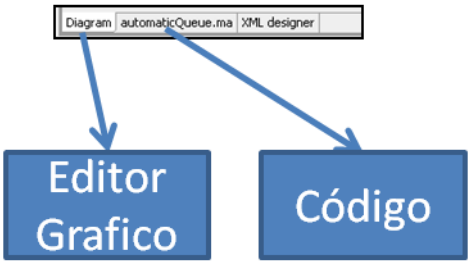


5. Abajo las áreas del editor:



Algunos Tips generales:

- Con **cntrl+ruedita del mouse** se puede hacer zoom in/out . También desde la paleta con los botones 
- En la solapa propiedades se puede hacer visible haciendo Clic derecho sobre cualquier objeto y seleccionando **Properties**
- Se puede pasar de la vista del editor al código con las solapas de abajo del editor

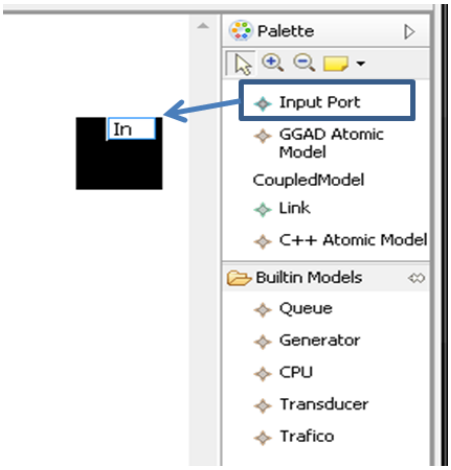


Crear puertos

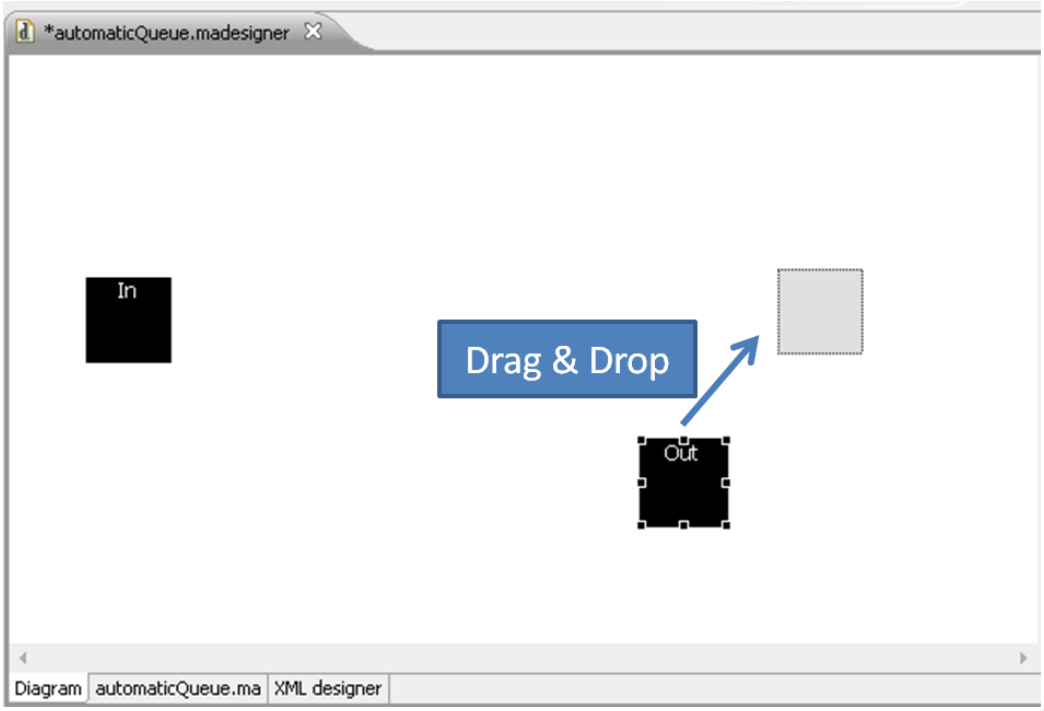
Vamos a crear 2 puertos: 1 de entrada y otro de salida

1. Hacer click en la paleta el icono **Input Port** 

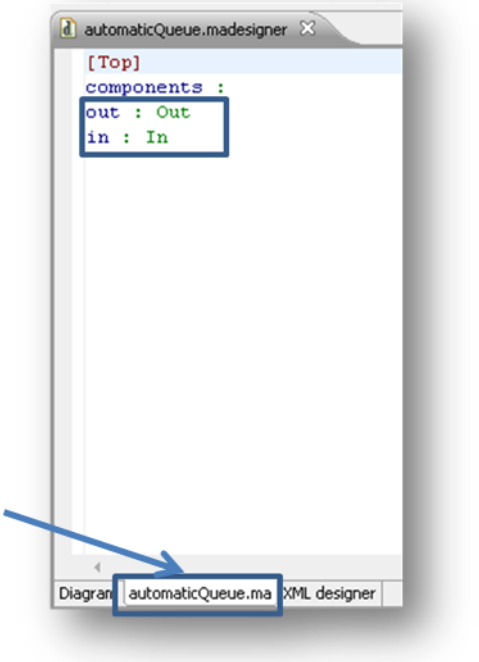
y hacer click en cualquier lado sobre el editor (le aparece un + a la flechita).
2. Aparece un cuadradito para darle un nombre al puerto. Poner **In.**
También se puede cambiar el nombre desde la solapa **Properties**



3. Repetir el paso 1 y 2, pero utilizando el icono **Output Port** y poner el nombre **Out**.
4. Todos los elementos se pueden mover a cualquier lado del editor, seleccionándolos y arrastrando. Poner el puerto **In** a la izquierda y el **Out** a la derecha.



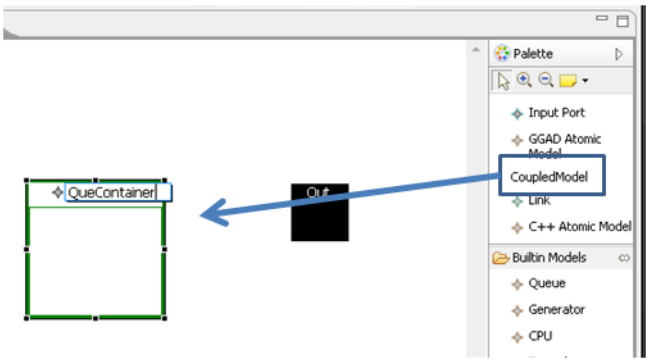
5. Grabar y verificar que se haya escrito bien el .ma desde la solapa **AutomaticQueue.ma**
   NOTA: se graba con **cntrl+S** o sino desde el botón de grabar de Eclipse o de **File | save**



Crear modelos acoplados internos

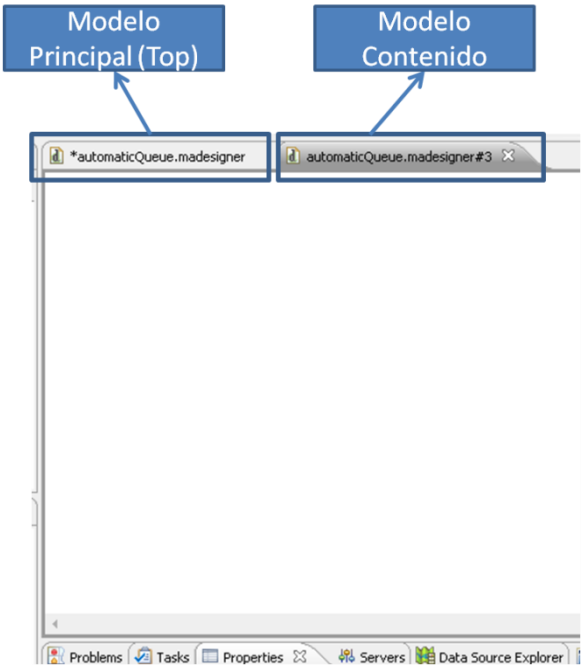
1. Elegir de la paleta el icono **Coupled Model** 

y hacer clic sobre el editor.
2. Darle nombre **QueueContainer** al nuevo modelo.

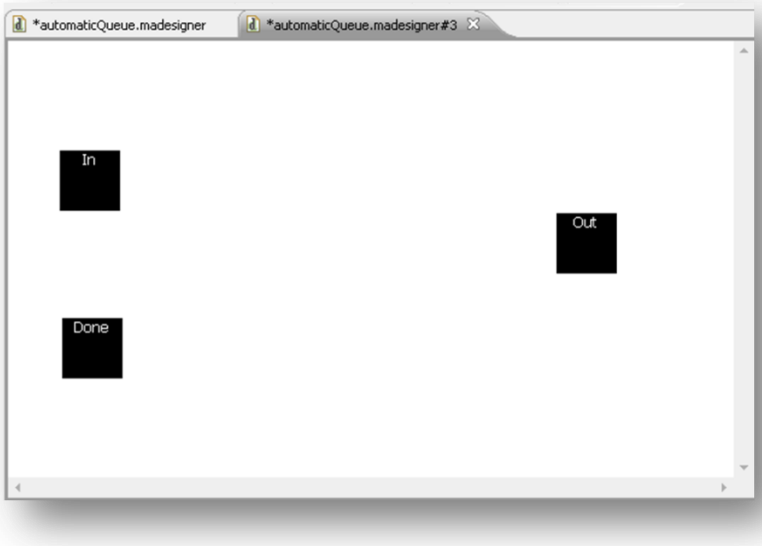


Editar modelos acoplados internos

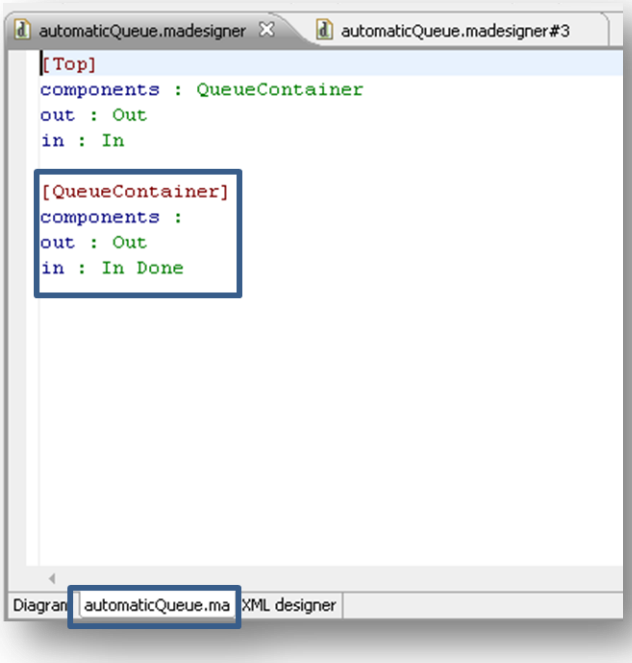
1. Para editar el modelo **QueueContainer,** hacer doble clic en la figura del modelo.
   Esto abre un nuevo editor (muy similar al anterior, pero sin las solapas de abajo)



2. A este modelo vamos a agregarle 3 puertos como se mostro antes: 2 de entrada (**In** y **Done**) y 1 de Salida (**Out**, recordando cambiar el tipo de puerto en las propiedades)



3. Grabar desde el modelo principal **automaticQueue.ma** y verificar que se haya escrito bien el .ma.
   NOTA: la solapita del .ma está en el editor principal (porque los modelos contenidos no tienen su propio .ma)



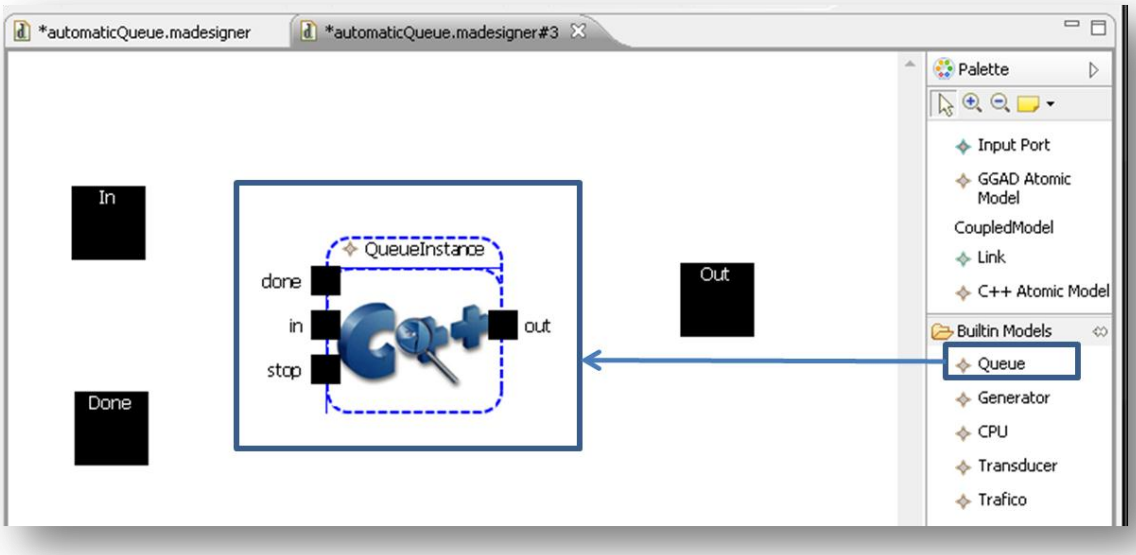
- Observar que los puertos en el modelo principal aparecen automáticamente. Si se modifican los puertos del modelo contenido, los cambios se mantienen (incluso los links).

Crear modelos atómicos (Built-in)

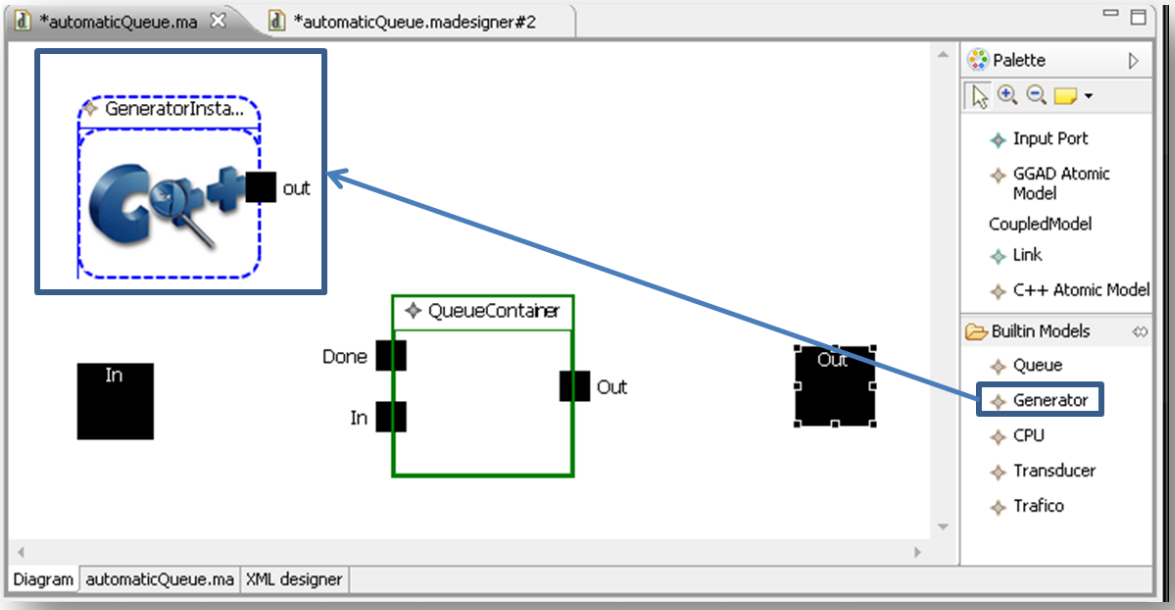
1. Ir al editor del modelo **QueueContainer,** elegir de la paleta **Queue** 

y apretar sobre el editor. Esto crea un modelo atómico **Queue.**

NOTA: Los modelos que aparecen en la solapa como **Built-in** son los que están registrados en el **register.cpp.** (en este caso el register.cpp interno). Si cambia el register.cpp cambian los modelos que aparecen ahí. Los puertos y parámetros que reciben estos modelos son leídos del archivo de código C++ que los define.



2. Hacer lo mismo, pero en el modelo **TOP**, agregando un **Generator** 



3. Al modelo de **QueueInstance y GeneratorInstance** hay que definirle los parámetros que aparecen en las solapa de **Properties**. Para el generador utilizaremos los siguientes valores

   preparation: 00:00:00:05

   Para la cola:

distribution: normal
mean: 2
deviation: 1

4. Grabar con el botón  .

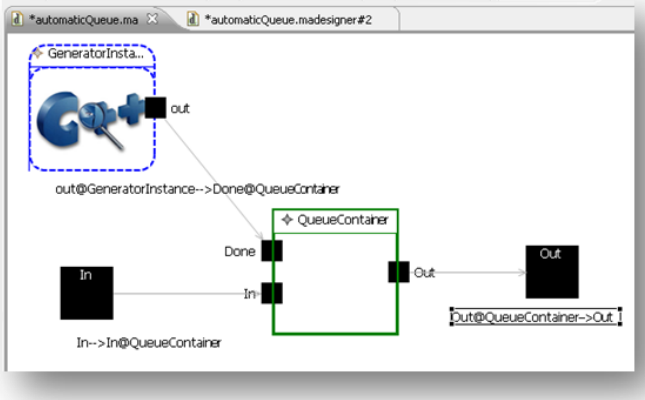
Crear Links

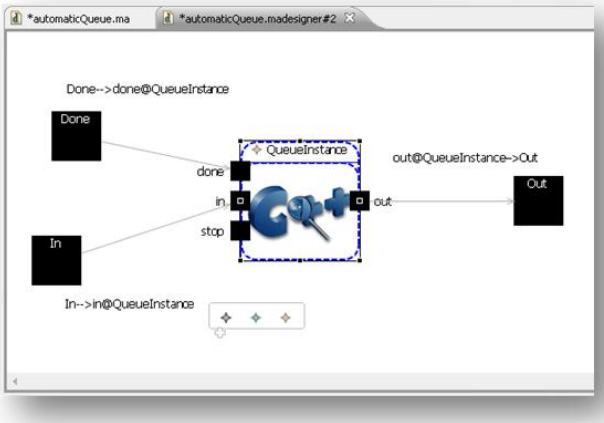
Lo único que estaría faltando ahora es unir los puertos.

Pasos para crear un link:

1. Seleccionar de la paleta el icono **Link**
2. Hacer click y mantener apretado sobre el puerto origen
   - Soltar el botón sobre el puerto destino. (Observar que el cursor cambia de figura avisando si el link es válido o no).

1. Crear los siguientes Links en el modelo Top:
   - De **In** a **In@QueueContainer**
   - De **out@QueueContainer** a **Out**
2. Crear los siguientes links en el modelo **QueueContainer**:
   - De **In** a **In@QueueInstance**
   - De **Done** a **Done@QueueInstance**
   - De **Out@QueueInstance** a **Out**





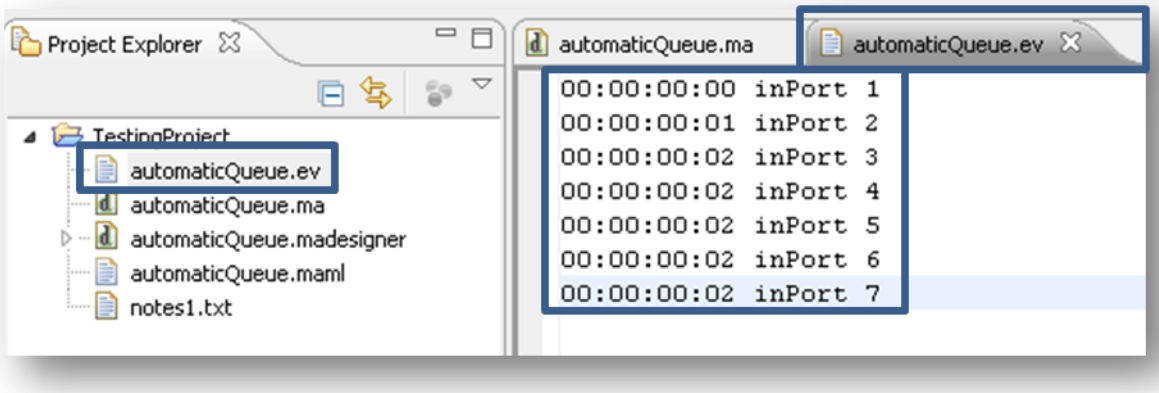
3. Guardar los cambios.

Ejecutar la simulación

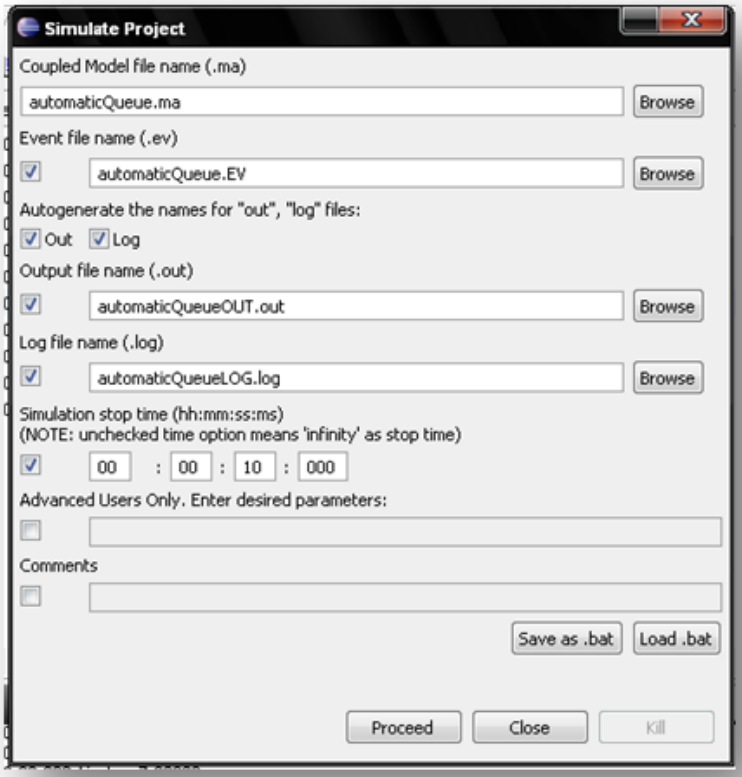
Para correr la simulación hace falta un archivo de eventos.

1. Crear un archivo **automaticQueue.ev** dentro del proyecto (**File | new | File)**
2. Copiar este código:

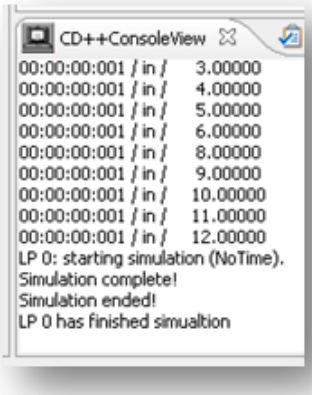
```
00:00:00:00 in 1
00:00:00:01 in 2
00:00:00:01 in 3
00:00:00:01 in 4
00:00:00:01 in 5
00:00:00:01 in 6
00:00:00:01 in 8
00:00:00:01 in 9
00:00:00:01 in 10
00:00:00:01 in 11
00:00:00:01 in 12
```

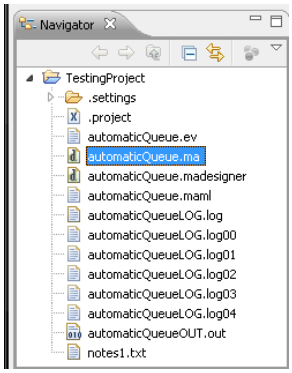


3. Grabar el archivo de eventos **automaticQueue.ev.**
4. Hacer click en el boton de simulación de CD++Builder 
5. Completar los campos como se ve en la siguiente Figura:



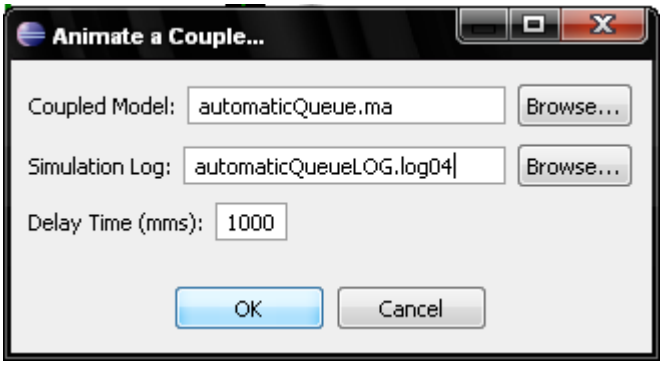
1. Hacer click en **Proceed** y va a comenzar la simulacion y al terminar genera varios archivos de log



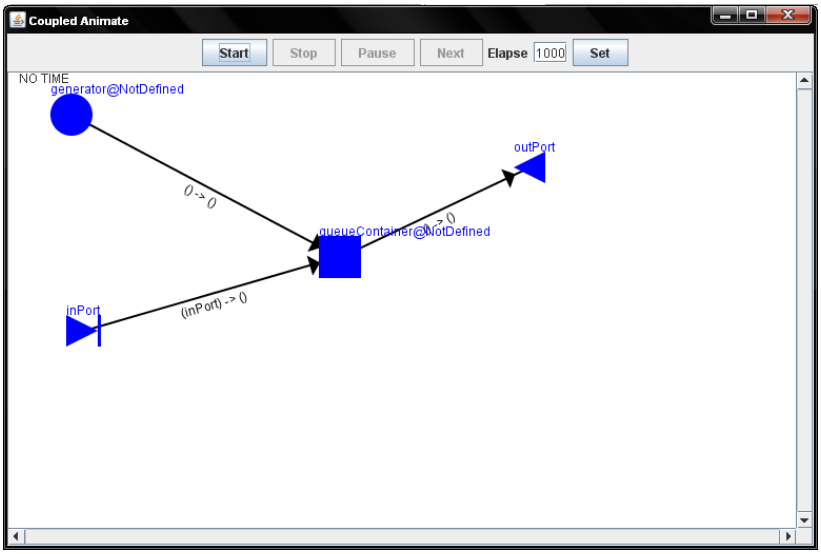


Ver las animaciones

1. Hacer click sobre el boton de animate Coupled Model 
2. Completar el fomulario como en la figura:



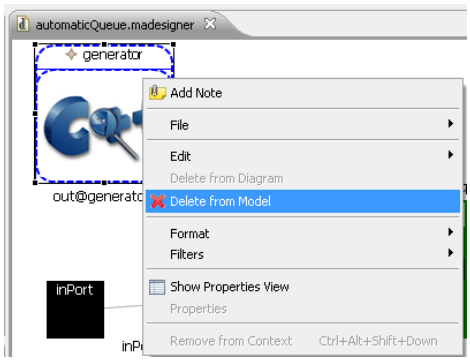
3. Se va a abrir el visualizador de animaciones de CDMolder con el modelos que habiamos definido.



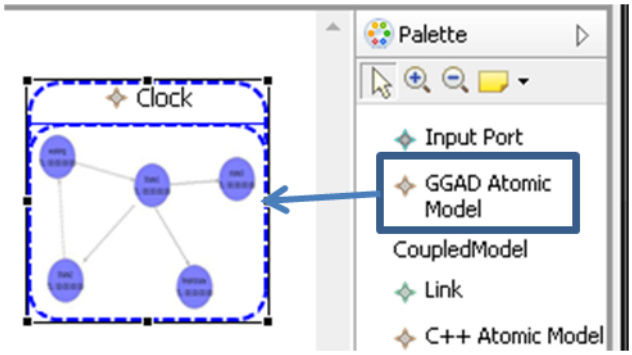
Crear un modelo atómico DEVS-Graphs

Vamos a reemplazar el **generator** que viene con el simulador, por un modelo atómico DEVS-Graphs con funcionalidad similar.

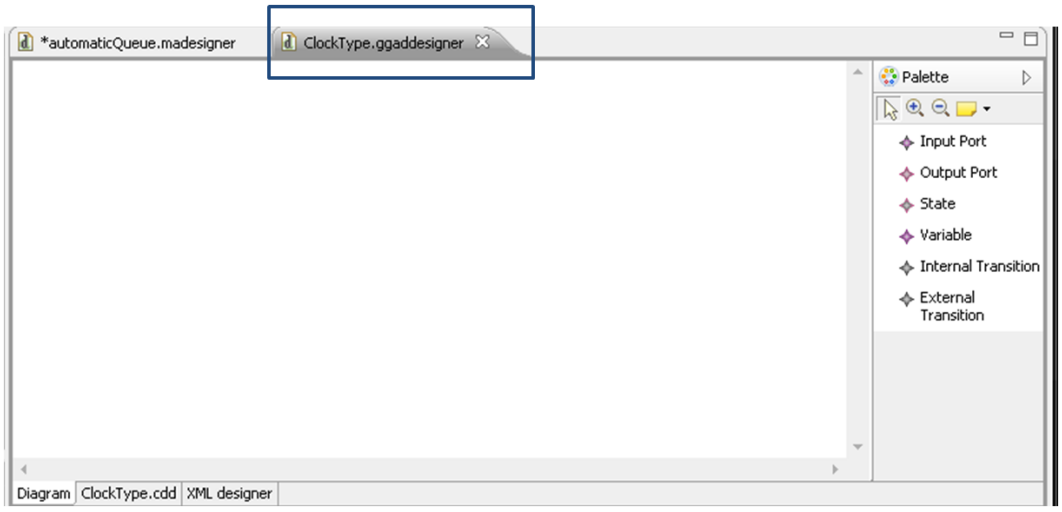
1. Eliminar el modelo **generatorInstance** eligiéndolo en el editor, haciendo click derecho y elegir **Delete From Model.**



1. Crear un modelo DEVS-Graphs eligiendo de la paleta **DEVS-Graphs Atomic Model** y hacer click sobre el modelo. Darle nombre **Clock.**



2. Hacer doble click sobre el modelo **Clock** para editarlo.

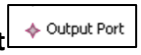


Crear puertos

Para crear puertos:

- De entrada, elegir de la paleta **Input Port** 

y hacer click sobre el modelo.

- De salida, elegir de la paleta **Output Port**  y hacer click sobre el modelo.

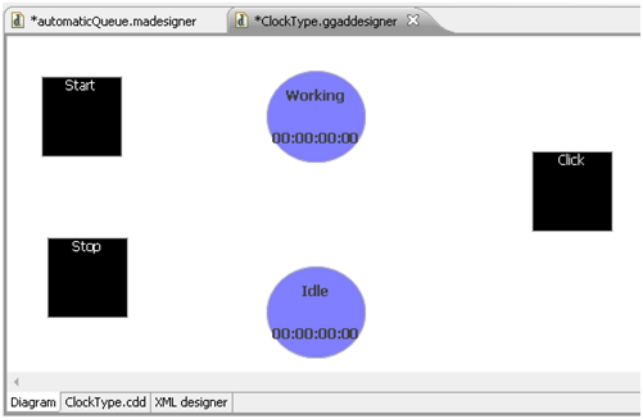
1. Crear 3 puertos: 2 de entrada (**Start** y **Stop**) y uno de salida (**Tick**).

Crear estados

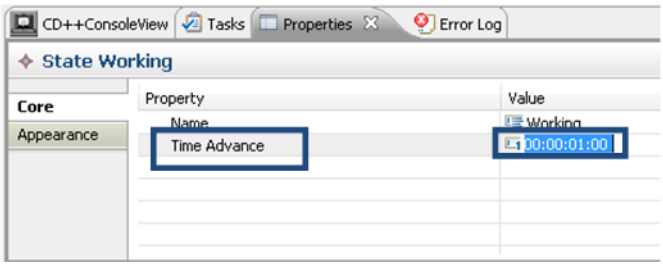
Para crear estados:

- elegir de la paleta **State**  y hacer click sobre el modelo.
- Darle un nombre al estado

1. Crear 2 estados: uno **Idle** y otro **Working**



2. Colocar en el estado **Working** el **TA** con **00:00:01:00** desde la solapa **Properties**.
3. Colocar en el estado **Idle** el **TA** con **infinity** desde la solapa **Properties**.

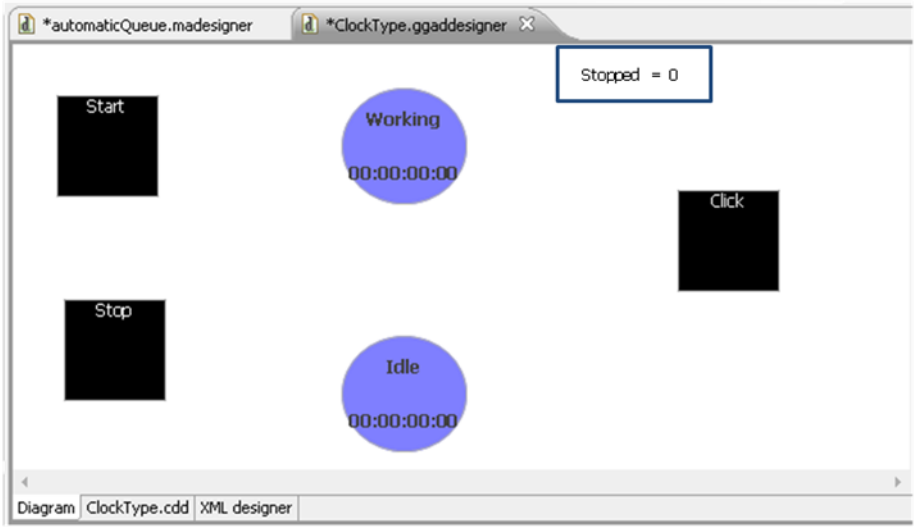


Crear Variables

Para crear variables:

- Elegir de la paleta **Variable** [Variable] y hacer click sobre el modelo.
- Darle un nombre a la variable

1. Crear 1 variable: **Stopped**

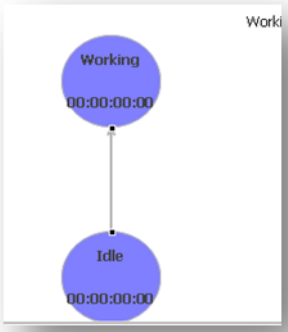


Crear Transiciones Externas

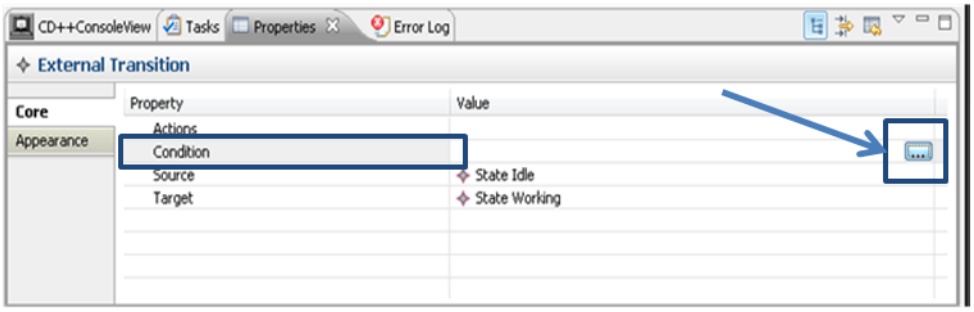
Para crear transiciones externas:

- Elegir de la paleta **External Transition** [External Transition], hacer click sobre el estado originen y arrastrar hasta el estado destino.
- Definir la **Condición** desde la solapa **Properties**
- Definir la **Acción** desde la solapa **Properties** (opcional)

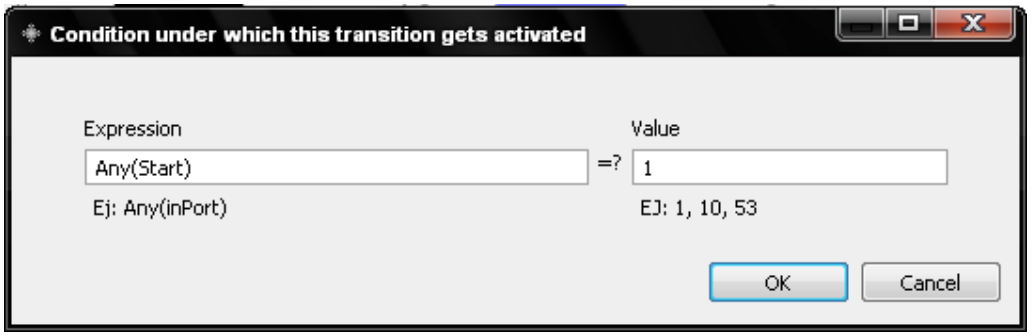
1. Crear una transición externa entre **Idle** y **Working**



2. Desde la solapa de **Properties** editar **Condition** de la transición con el botón 

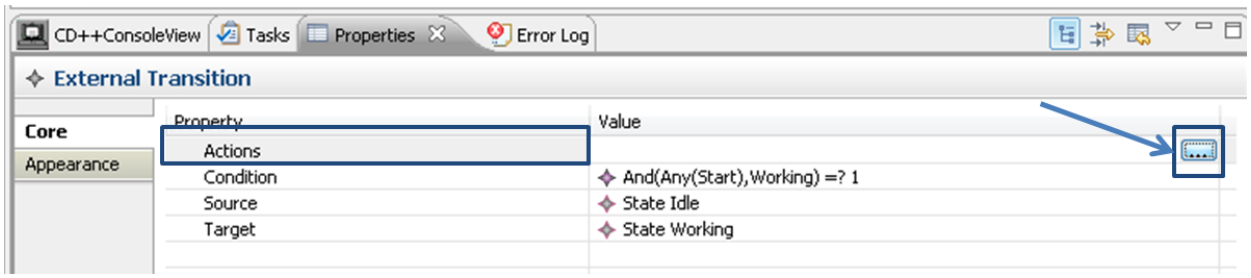


3. Completar el formulario de la siguiente manera:

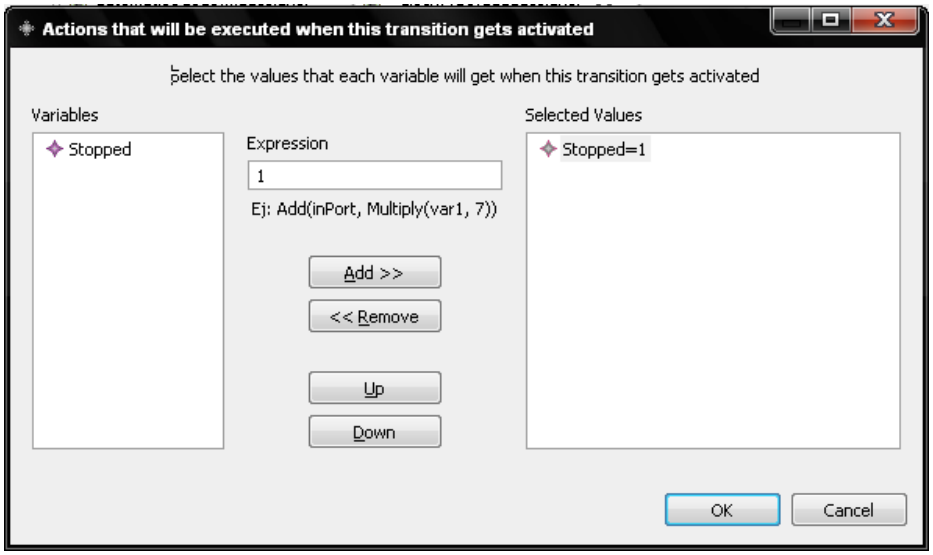


NOTA: Esta condición indica que la transición se ejecutará para cualquier valor del puerto Start (para hacer bien el modelo habría que hacer la expresión más complicada para que use el valor de la variable)

4. Desde la solapa de **Properties** editar **Actions** de la transición con el botón 



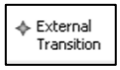
5. Completar el formulario de la siguiente manera (usar el botón **Add** para agregar los valores):



NOTA: Esto significa que cuando se ejecuta la transición la variable **Stopped** toma el valor 1

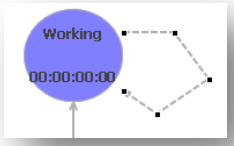
Crear Transiciones Internas

Para crear transiciones internas:

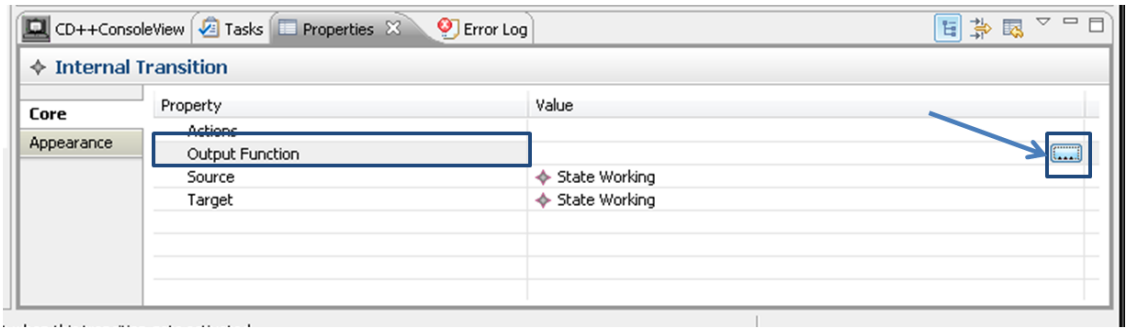
- Elegir de la paleta **Internal Transition** 

, hacer click sobre el estado originen y arrastrar hasta el estado destino.
- Definir la **OutPut function** desde la solapa **Properties**
- Definir la **Acción** desde la solapa **Properties** (opcional)

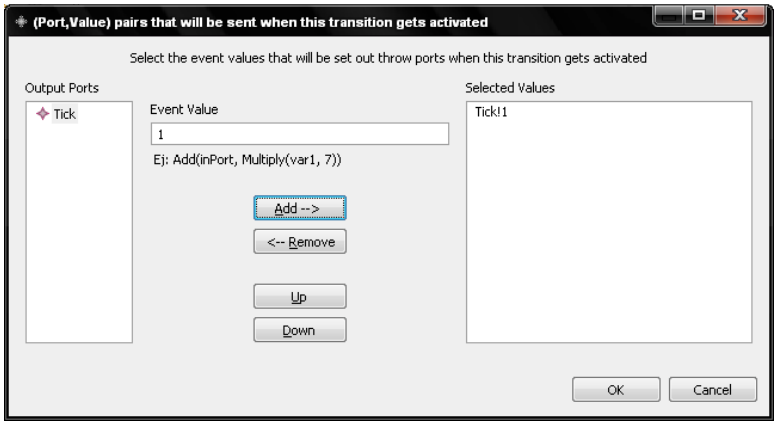
1. Crear una transición interna de **Working** a si mismo



2. Editar la output Function desde la solapa **Properties**

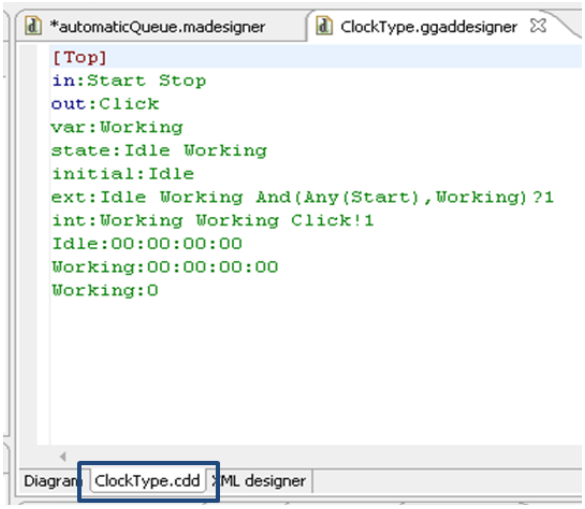


3. Completar el formulario de la siguiente manera(usar el botón **Add** para agregar los valores):

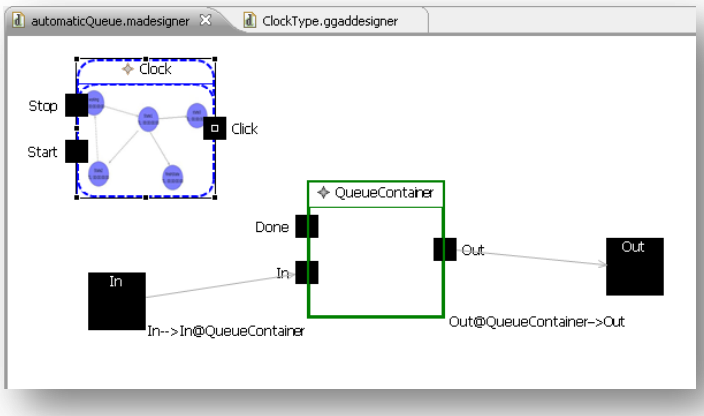


NOTA: esto hace que cuando el TA de *working* expira, por el puerto **Tick** se manda el valor 1

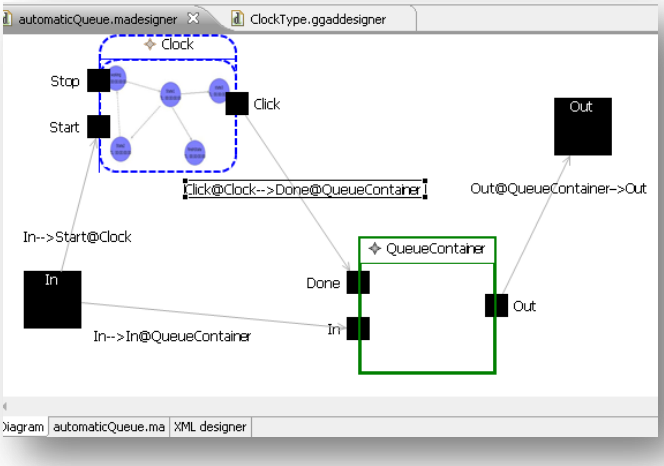
4. Grabar el modelo y observar el código en el archivo **ClockType.cdd**



5. Volver al modelo **Top** principal. Observar que los puertos se reflejan automáticamente en el modelo acoplado

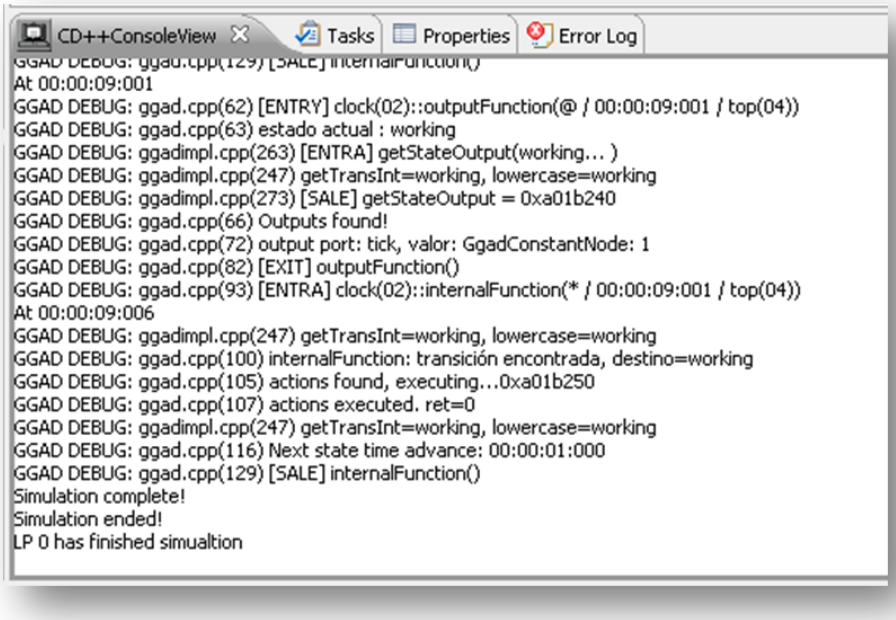


6. Falta editar el modelo acoplado para unir los puertos con el nuevo modelo DEVS-GRAPHS. Crear los siguientes link como se explicó en Crear Links
   - Crear un link entre **In** y **Start@Clock**
   - Crear un link entre **Tick@Clock** y **Done@QueueContainer**



Ejecutar la simulación

Para ejecutar la simulación y ver las animación es igual a como lo hicimos con el modelo acoplado Ejecutar la simulación.



NOTA: Cuando se simulan modelos con atómicos DEVS-Graphs, el ejecutable del simulador que se usa es diferente.

Apéndice C: Manual de desarrollo de CD++Builder

El siguiente es el manual para desarrolladores de CD++Builder, que brinda información útil para aquellos desarrolladores que deseen implementar nuevas funcionalidades o resolver Bugs en CD++Builder. El manual puede descargarse junto con el software de http://sourceforge.net/projects/cdppbuilder/files/.

CD++Builder Developer Guide

This guide provides useful information and steps to start developing CD++Builder Eclipse plugin. It should be used by developers that want to implement new features or fix bugs on CD++Builder. Users that what to create models and simulate them should read *CD++Builder User Guide.*

How to Configure Development Environment Configuration

1- Download the plugin source code (see CD++Builder SVN repository)
2- Download and install Eclipse 3.5.1 de http://www.eclipse.org/downloads/
3- Use the SVN root as the workspace for Eclipse. **File-> Switch Worspace** and choose the directory where you downloaded the source code.
4- Import all projects to the workspace. **File -> Import --> Existing Project into Workspace**, choose the SVN root directory and select all available projects (except for the ones in the **Usefull Things** folder).
5- Install Birt, CDT and GMF Frameworks. **Help --> Install new Software** and select the **Galileo - http://download.eclipse.org/releases/galileo** update site in the work with textbox.
    a. From the **Modeling** category check **Graphical Modeling Framework SDK** (GMF)
    b. From the **Business Intelligence, Reporting....** category check **BIRT Framework.**
    c. From the **Programming Languages** category select **Eclipse C/C++ Development Tools** (CDT).
    d. Click **Next** and accept all forms.
    e. Restart Eclipse.

All projects should now compile correctly. It might be necessary to clean project output (Project --> Clean... -> clean all projects).

CD++Builder SVN Repository

The latest source code version for the CD++Builder plugin is maintained in the following SVN repository: https://svn2.assembla.com/svn/devsmodeler/. To download the code, you will need to be granted access (contact gwainer@sce.carleton.ca) and download some SVN client software

(Tortoise SVN is recommended for Windows and can be downloaded from http://tortoisesvn.net/downloads).

How to upload a new version to CD++Builder update site.

The update site for CD++Builder is hosted in the Sourceforge web site for the cdppBuilder project: http://cdppbuilder.sourceforge.net/updatesite/. To upload new versions you will need administrator privileges on the cdppBuilder sourceForge project (contact gwainer@sce.carleton.ca) and the WinSCP client.

To build the new version, open the **site.xml** file located in the **cdBuilder.updateSite** project. The **plugins** and **features** folders can be deleted to avoid uploading unnecessary files. Click the **Build All** button to generate the compiled .jar files and XML metadata files.

Using WinSCP upload the following files (located in the cdBuilder.updateSite project) to the **/home/groups/c/cd/cdppbuilder/htdocs/updatesite** folder of the website (see below on see how to configure WinSCP for uploading files to the website):

- **site.xml** file
- **plugins** folder
- **featues** folder
- **artifacts.jar** file
- **content.jar** file

To configure WinSCP for uploading files to the website:

Host name: web.sourceforge.net
Username: <sourceForge usename>,cdppbuilder (Ej: **camisa666,cdppbuilder**)**. It is important not to place a space after the comma).**
Protocol: SFTP
Port number: 22

How to upload a new version of Eclipse to Sourceforge File System.

CD++Builder eclipse versions are hosted in the Sourceforge File Sytem for the cdppBuilder project: http://sourceforge.net/projects/cdppbuilder/files/. To upload new files you will need administrator privileges on the cdppBuilder sourceForge project (contact gwainer@sce.carleton.ca) and the WinSCP client (can be downloaded from http://winscp.net/eng/download.php).

To configure WinSCP for uploading files to the sourceforge File System:

Host name: frs.sourceforge.net
Username: <sourceForge usename>,cdppbuilder (e.g.: **camisa666,cdppbuilder**)**. It is important not to place a space after the comma).**
Protocol: SFTP
Port number: 22

Files must be uploaded to the **/home/pfs/project/c/cd/cdppbuilder** folder of the remote file system.

Checklist of things to review before uploading a new Eclipse version:

- When starting Eclipse, it should ask which workspace to use.
- The default workspace should be **C:\eclipse\CD++Builder Example Workspace**
- Verify that the CD++Builder update site is in the list of updatesites. No local updatesite should be there.
- If the default workspace is chosen, the examples projects should be there.

Apéndice D: Trabajo aceptado en el Simposio de Teoría de M&S 2010

El siguiente trabajo, que presenta los principales resultados obtenidos en la tesis, fue presentado y aceptado en el Simposio en Teoría de Modelado y Simulación: 2010 Spring Simulation Multiconference (SpringSim'10, April 11-15 2010, Orlando, FL, USA), organizada anualmente por SCS - The Society for Modeling & Simulation International.

El trabajo obtuvo el "Runner-Up Overall Paper Award" (2do. mejor paper) seleccionado de entre 350 trabajos enviados a los 9 simposios de toda la conferencia.

Advanced IDE for Modeling and Simulation of Discrete Event Systems

Matías Bonaventura[1], Gabriel A. Wainer[2], Rodrigo Castro[1]

[1] Computer Science Department
Universidad de Buenos Aires.
Ciudad Universitaria. Pabellón I
(1428) Buenos Aires, Argentina.
Email: abonaven@dc.uba.ar, rodrigocastro@ieee.org

[2] Dept. of Systems and Computer Engineering
Carleton University Centre of Visualization and Simulation (V-Sim)
1125 Colonel By Dr. Ottawa, ON, Canada.
Email: gwainer@sce.carleton.ca

Abstract: Creating models and analyzing simulation results can be a difficult and time-consuming task, especially for non-experienced users. Although several DEVS simulators have been developed, the software that aids in the modeling and simulation cycle still requires advanced development skills, and they are implemented using non-standard interfaces, which makes them difficult to extend. The architecture and design of CD++Builder we present here can simplify the construction and simulation of DEVS models, facilitate model reuse and promote good modeling practices by allowing enhanced graphical editing and integration of tools into a single environment. The Eclipse-based environment includes new graphical editors for DEVS coupled models, DEVS-Graphs and C++ atomic models (including code templates that are synchronized with the graphical versions). Integration with Eclipse allows extensibility while simplifying software development, installation and updates.

1. INTRODUCTION

In recent years, the DEVS formalism [1] has become very popular, and several simulators have been implemented using diverse technology. In most DEVS simulators, models are defined using some programming language, which makes a difficult task modeling real world systems for non-expert developers. In order to deal with this problem, several tools now simplify the process of creating DEVS models, executing simulations and analyzing the results. Some provide graphical modeling capabilities, tools for tracking and viewing simulation results. Most of the tools have some modeling limitations and require the users to have programming experience.

While most of these tools allow the graphical definition of DEVS coupled models, they have been developed from the ground up without using standard user interfaces. That makes extending their functionality very difficult, as it is required to know the implementation details. Likewise, atomic models must be defined in some programming language. This makes it difficult for non-expert users to create new models. In some cases where code structure aids are provided, no graphical support is given for atomic model behavior specification. For instance, CD++ [2] provides different languages for specifying DEVS coupled and atomic models. CD++ models can be defined in C++, but graphical representation of the model is also available.

Here, we present new facilities of the CD++Builder tool [3], which have focused on these problems. The integration of several CD++ tools available into a single environment reduces the learning curve for new users and students, and simplifies the M&S processes by avoiding different model formats. Enabling the creation and edition of coupled and atomic models graphically allows users to specify complete DEVS models without programming. In the cases where complex behavior needs to be developed in C++, code templates avoid repetitive and error-prone tasks, and provide basic sample structures that promote good

programming and modeling practices. CD++ simulator is continually being updated and new tools are provided. To keep the M&S environment integrated in the future and avoid new tools to be developed in different platforms, CD++Builder features need to be easily reusable and simple to extend.

To fulfill these requirements, CD++Builder refactors the graphical modeling capabilities of CD++Modeler and GGADTool [4] (which were developed in Java and Visual Basic respectively) into the Eclipse environment [5].

Eclipse's plug-in architecture makes it simple to add new features, while guaranteeing the easy reuse of components (by means of the standard frameworks used by CD++ Builder). Thus, all the main activities can be done using the same user interface: composing and defining DEVS models, animation of simulation results, programming of C++ code, compilation of new atomic models into the CD++ framework and launching the execution of simulations.

CD++Builder provides graphical modeling editors, based on atomic DEVS-Graphs models [6]. Coupled models are described in a model definition file using a platform-independent language that can potentially be used by other tools as a mean for porting models between simulators. CD++Builder provides code templates with the basic structure to implement the CD++ abstract *Atomic* class, and these models are graphically represented and kept synchronized with graphical coupled editors. Automated regression tests are included, and they help to include new functionalities in CD++Builder, while provide a way to verify the correct behavior of the software after new code is introduced.

2. BACKGROUND

DEVS [1] is a general formalism for modeling and simulation of any discrete systems using hierarchical composition of behavioral models (atomic) and structural models (coupled) with well-defined interfaces. DEVS is independent from any simulation mechanism, which allowed several simulation tools to be developed, tackling different needs and providing advantages on specific scenarios. A non-comprehensive list includes:

- DEVSJAVA [7] provides four Java packages that separate modeling and simulation from user interface, allowing hierarchical model definition and visualization. DEVS coupled and atomic models are built by extending one of the base Java classes provided by the framework. Models need coded and recompiled for changes to take effect (which is difficult for non-Java experts). SimView [8], a graphical component of DEVSJAVA allows the user to specify the model layout in the model's source code. CoSMos [9] allows generating the Java code used to extend DEVSJAVA base classes, which simplifies the definition of atomic models. On the other hand, the user still needs to program, and once the code is generated, model structure cannot be modified (adding/removing ports, changing model name).

- JDEVS [10] is a visual tool that provides a module for executing simulations, a module for the user interface and two modules for 2D and 3D visualization. Though general-purpose models can be defined, the visualization modules are specifically built for natural systems and its graphical editors do not allow creating hierarchical models.

- SimStudio [11] is a web-based framework implemented using Java web technologies, using a layered architecture that supplies modeling, visualization and analysis players. The modeling plug-in, implemented in Flash, allows to get from a graphical specification of a model, a XML file that is used by other tools.

- VLE [12], implemented in C++, is a modeling and simulation environment oriented to integrate heterogeneous formalisms wrapping submodels as DEVS models to enable interoperability. It provides separated modules for the GUI, for visualizing results and a core that implements the simulation algorithms.

- PowerDEVS [13] allows graphical specification of coupled DEVS models, and atomic models are defined in C++. A special editor aids the modeler with code structure, and a model library enables model reuse in a drag and drop fashion. Tracking model state during simulation is done by special atomic models that interact with outside devices. Although this approach is useful, model definition and simulation tracking are mixed into the same editor.

Our work is based on CD++ [2, 14], a modeling and simulation tool that implements the DEVS and Cell-DEVS theory. CD++ has been widely used is several areas of interest such as urban traffic, physical systems, computer architecture and embedded systems. CD++ is implemented in C++ as a class hierarchy where models are related to simulation entities. Atomic models behavior is programmed in C++, and coupled models are defined in a model definition file using a built-in high-level language. Though defining atomic models in C++ gives the modeler flexibility to specify behavior, it requires advanced programming skills. Thus, CD++ also supports defining DEVS-Graphs atomic models without a programming language.

CD++Builder [3] is an Eclipse plug-in that integrates varied applications and utilities that aids in creating CD++ DEVS models, simulating and analyzing results. Among these applications, CD++Modeler provides a graphical editor for coupled and DEVS-Graphs atomic editors, and visualization of simulation results. CD++Modeler lacks integration with the rest of CD++ applications and utilities (requiring to export models to different formats) and has some limitations (i.e., not being able to define atomic

models in C++). New extensions to CD++Builder were built from the ground up, tackling most of the problems of other CD++ tools and introducing new features available in other simulators into CD++ to facilitate modeling and simulation tasks.

3. CD++BUILDER

Figure 1 shows the CD++Builder environment, which integrates all the capabilities available in CD++ tools with features that facilitate modeling and compilation. Incorporation of all features into the Eclipse environment allowed the development of powerful tools built upon existing features and Eclipse infrastructure.

Action buttons in the top toolbar allow executing external tools. A special section is reserved for CD++Modeler animations, including CELL-DEVS, Atomic and Coupled models. Buttons for launching other legacy tools (CD++ Modeler, GGADTool and Drawlog) are provided. The *Build* button generates a make file and compiles the source code of all atomic models in the project, generating a simulator. The *Execute* button pops up a wizard that allows specifying necessary parameters to run a simulation. New model files (coupled or atomic) can be added using Eclipse *New File* wizard, which will be shown in the project files navigation panel. Editing models is also done within Eclipse interface by means of graphical editors. Coupled model editor allows defining C++ atomic models and generates code structure to extend the Atomic CD++ class. Eclipse allows users to rearrange windows to personalize the interface, and the CD++Builder *Perspectives* layout buttons and panels have been used to improve the modeling of particular scenarios.

As seen on Figure 1, CD++Builder has been now integrated with Eclipse C/C++ Development Tools. It also has been provided with new animation features to allow visualization of simulation results graphically. This graphical framework improves usability including usual editing actions (as copy, paste, undo and redo).

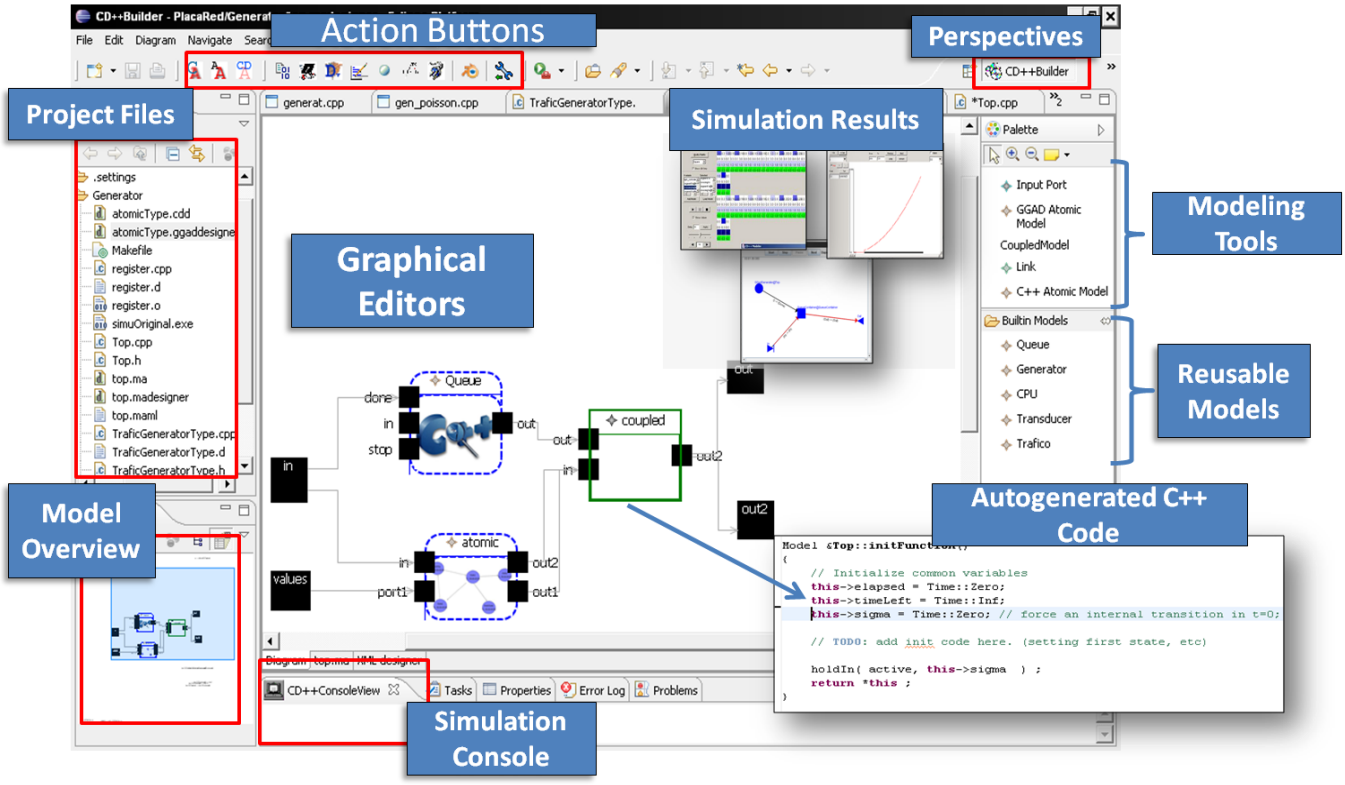


Figure 1. *CD++Builder environment*

Other features that facilitate graphical modeling were incorporated, such as zoom in/out capabilities, flexible look and feel for texts and shapes and different styles for model links to avoid obstructions and overlapping. Both coupled and atomic model editors provide a special pane with tools for easily creating available entities (atomic/ coupled models, links, ports, transitions, states, variables, etc.).

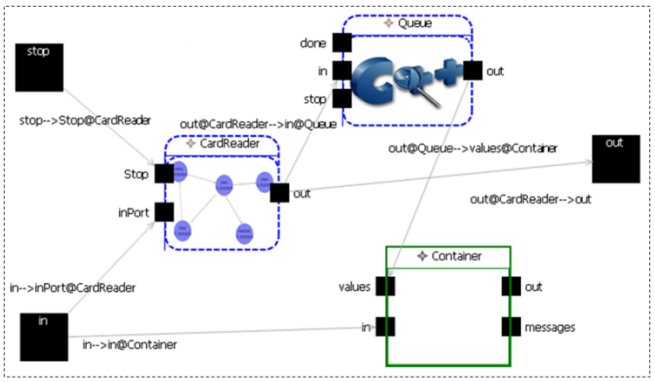


Figure 2. CD++Builder coupled model editor.

The Eclipse Properties view is used to show and edit entities details, the Outline view to show an overview of the model and the New File Wizards for creating new atomic and coupled DEVS model diagrams. CD++Builder uses a common description file and editing domain for models and submodels. Submodels are edited in their own tabs, and all the opened models are kept consistently linked.

In the coupled model editor (shown on Figure 1 and 2), models are graphically represented by colored rectangles with the model's name. Coupled and atomic models are differentiated by color and shape, while inner images are used to distinguish atomic models types (DEVS-Graphs and C++). Ports are rendered as black boxes with their names and directed links to represent the associations with the models (which makes easy the creation and understanding of links).

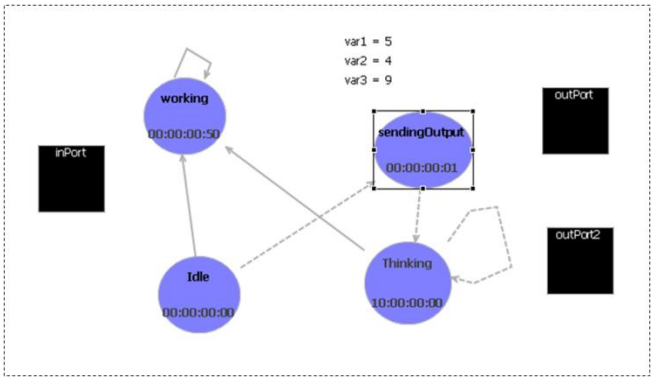


Figure 3. CD++Builder DEVS-Graphs model editor.

The DEVS-Graphs atomic editor (Figure 3) uses DEVS-Graphs notation: atomic model's *states* are represented by circles with *id* and *time advance* values. Internal transitions are represented with dotted arrows and external transitions with full arrows linking origin and destination states.

Implementing C++ atomic models in CD++ requires creating C++ files that define a new class derived from the abstract *Atomic* class, and modifying the *register.cpp* file to register the model within the framework. These tasks are tedious and error prone. To simplify and speed up C++ atomic model definition, code generation capabilities were added to the coupled DEVS model editor. When a new C++ atomic model is selected, C++ files are generated based on a template, and *register.cpp* file is automatically updated. The template (shown in Figure 4) supplies the code structure to extend the *Atomic* class, providing helpful comments and code samples. The model name is used to create the .cpp and .h files and to name the new class. All methods that must be implemented (initFunction, externalFunction, internalFunction and outputFunction) are set in place with a brief comments useful for people learning DEVS or CD++. Comments are also used to give an example of how to add input/output ports and parameters to the model. A similar template is also provided to implement the header (.h) file.

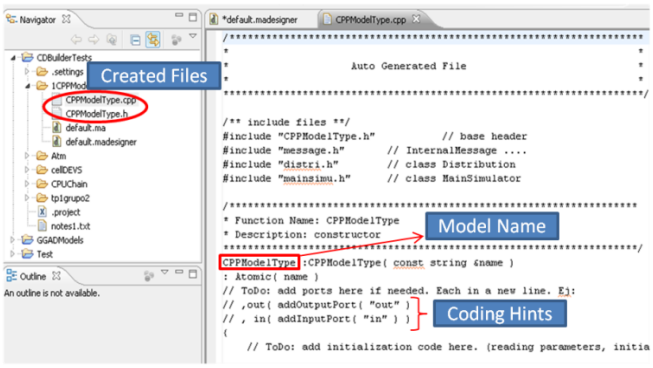


Figure 4 - Template-generated code for the Atomic class.

When developing atomic models directly in C++, the model's graphical representation is kept consistent with its C++ underlying code by means of a newly developed parser. When C++ files are modified and saved, the parser recognizes special code structures to identify model name, input/output ports and parameters (constant values to configure atomic models). In this way, the model graphical representation and the code are always synchronized (with no restriction imposed on the code) enabling to modify the graphical metaphor at any time.

New editors have been adapted to reuse the animation features previously available for CD++Modeler (as they have been successfully used for visualizing simulation evolution and results). A control is provided to manage time advance, and links are dynamically highlighted to represent events from one model to another. For coupled models, a block representation is used; for atomic models, the input/output trajectories are shown on different ports over time

CD++ coupled models and DEVS-Graphs high-level languages have the power for fully describing DEVS models and enabling graphical representation while not restricting it with visualization-specific information. Thus, model behavior definition and graphical representation are clearly separated. Figures, sizes, layout, colors, and all graphic-specific information are stored in a separate file from model definition. While this separation is conceptually correct, it presents some challenges when implementing graphical editors. CD++Modeler and GGADTool used custom structured file formats for representing model graphics from which model definition could be extracted (through export operations). Nevertheless, the opposite operation (generating a graphical representation from model definition) was unavailable in both tools. CD++ has a vast model repository, so this limitation is a stopper for using these tools: models that were already implemented and tested could not be opened in previous graphical tools. Moreover, once models were exported to CD++ formats they could not be easily updated without losing consistency with the graphical representation. In CD++Builder, the graphical representation information is stored in XMI [15] (XML Metadata Interchange) format and persistence from and to this format is handled by EMF services. New Parsers and Writers were developed to implement translators, from CD++ grammar to graphical representation and vice versa. This way, new coupled and DEVS-Graphs editors can show both views (graphical model representation and textual CD++ model definition) which can be selected by means of small tabs at the bottom (Figure 5). When any of the views is saved, both files are synchronized using the translators to keep them consistent.

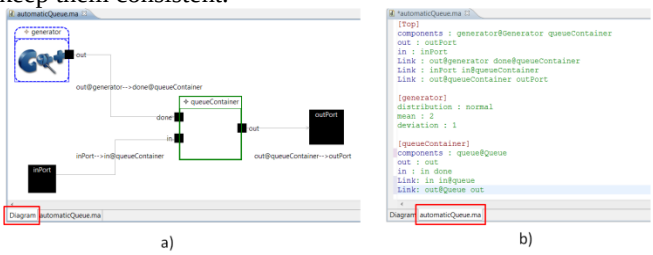


Figure 5 - Model views available: a) graphical b) textual.

Although graphic files contain all the information to rewrite the CD++ model definition completely, the opposite is not true. Thus, when the textual file is saved, the graphic diagram file can be consistently updated, but all graphical information is lost. To overcome this issue, when the textual model definition is saved, the idea is to synchronize the old diagram using its graphical information (layout, figure sizes, colors, etc) to supersede any missing information. To tackle the limitations of previous tools, translators can also generate a new graphical representation from a textual model definition, enabling to use models not built using graphical editors. In this case, a default values are used for the missing graphical information.

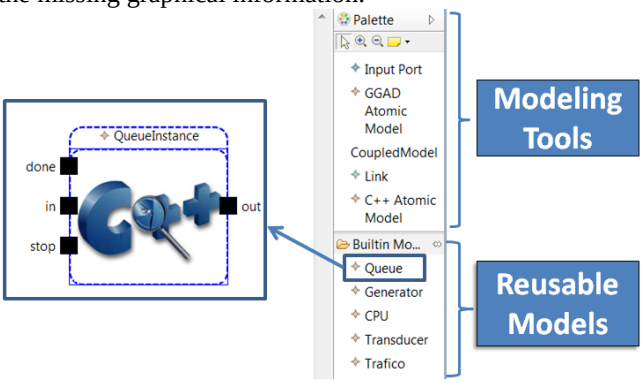


Figure 6 - Coupled model tool pane with reusable models.

Having the ability to view all models graphically helps in better understanding other users' models, facilitating model reuse among the community. In this sense, we added a pane (Figure 6) including a section for reusable models. This let users that are not familiar with the CD++ model library to know which models previously created are available. By dragging and dropping, these models can be composed within the model being edited.

Installing CD++Builder previously was done by downloading Eclipse and other tools. Installation and updates of CD++Builder Eclipse plug-in is now moved into a centralized schema, integrated with the Eclipse Update Manager package. This allows hosting plug-in compiled code and its metadata in a single publication site, which all users will access for installation and periodical check for updates. The following figure shows this new architecture:

Figure 7. Centralized installation and update architecture.

This new scheme allows easier wizard-guided installation into already running instances of Eclipse. More importantly, it resolves versioning problems; software bug

fixes and latest features do not have to be distributed to users individually, but uploaded into a centralized point. Integration with the Eclipse Update Manager allows clients to trigger manual update checks or to configure for scheduling automatic periodic updates.

For the development of key components in CD++Builder, a Test Driven Development [16] approach was used. Automated unit tests provided help improving software quality, and they facilitate extensibility. Tests provide a higher level of certainty that a given functionality is correctly implemented, and as they can be automatically run at any time, they can be used to verify whether a code refactoring or a new feature did not break other features. For this aim, JUnit [17] framework was used, as it provides out-of-the-box support of Eclipse plug-ins testing.

4. Architecture and Technology

The layered scheme in Figure 8 depicts the role of each CD++ component and their main relationships. At the lowest layer (on top of the operating system) the CD++ Simulator implements the simulation algorithms based on DEVS and Cell-DEVS formalisms. CD++ provides different versions of the abstract simulators (e.g. parallel, flat, real-time).

At the next level (Libraries) CD++ provides a basic out-of-the-box atomic model library, including a Generator, a Transducer, and a Queue, which can be directly used to define coupled models. The Core Simulator and the Libraries layers are usually distributed together as the CD++ Simulator. In addition, interpreters for high-level modeling languages are part of the Libraries. These interpreters accept input files, coming from the Modeling level, which can define coupled models compositions, Cell-DEVS models or DEVS-Graphs atomic models, without requiring the use of a high-level programming language for their definition.

Other area-specific interpreters are also available in some versions, such as ATLAS for describing urban traffic or M/CD++ to describe continuous models using Bond Graphs and Modelica [2]. All these interpreters are implemented using the core classes, so they can be used independently of the simulator version.

When a custom atomic model behavior needs to be defined, it can be done with User Models, which extend the Atomic C++ base class of the framework (in this case recompiling CD++ is required).

To execute a simulation, the coupled model definition file and input events must be specified, among other options. A simulation can generates two output files that can be used to track the simulation run; the *.out* file contains the port-value pairs for the output events of the model, while the *.log* file contains all the message passing and synchronization information between different models.

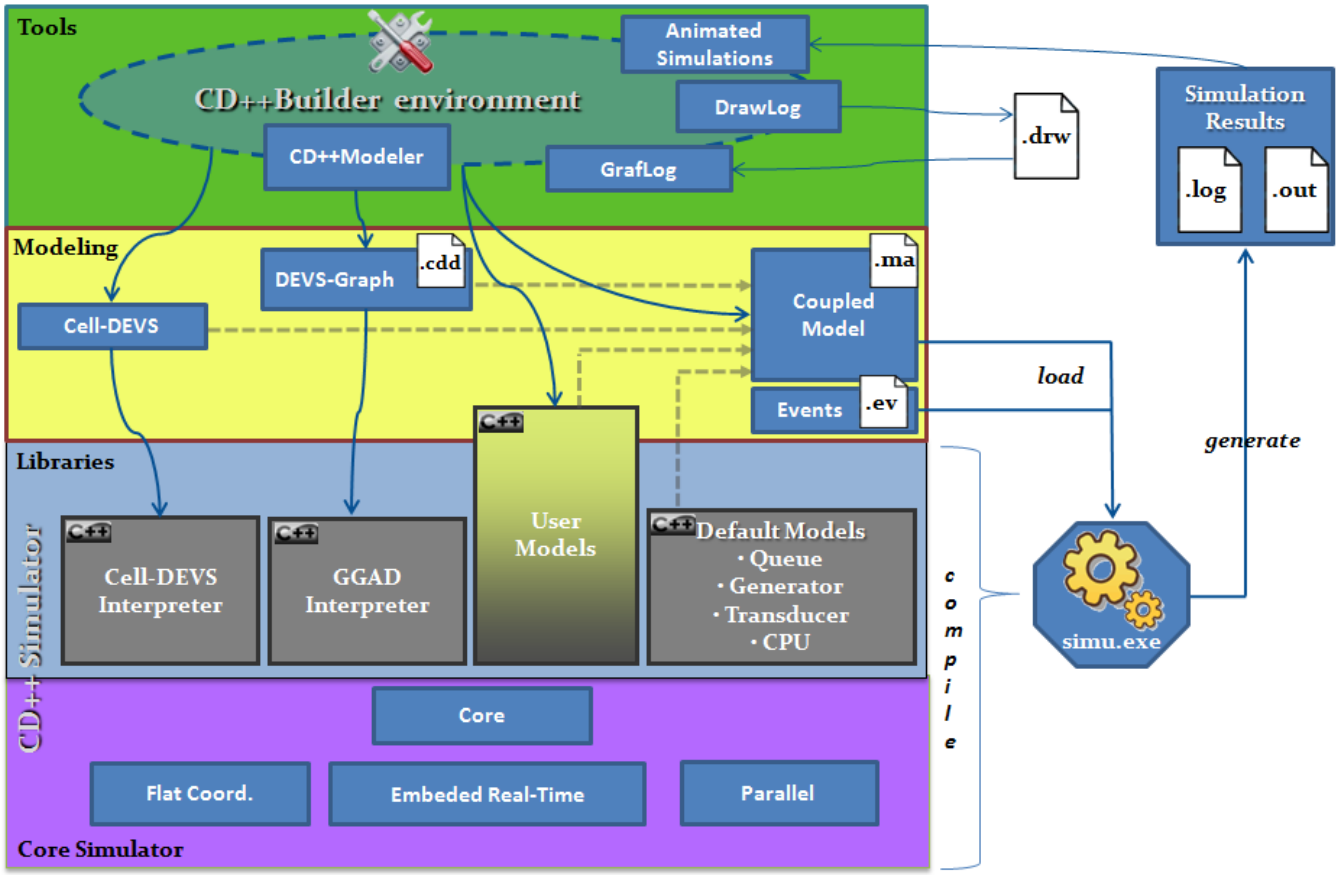


Figure 8. *CD++, high-level modeling languages, execution process and supporting tools.*

At the top Tools layer, different applications have been developed to facilitate output file visualization, such as Drawlog for Cell-DEVS model simulation visualization and CD++Modeler to animate coupled models message passing and atomic model output values. CD++ provides graphical editors to specify model behavior, and generate high-level specifications that must be interpreted by the lower layers.

This layered architecture and the clear separation between simulation execution, model definition, supporting tools and underlying libraries, allows modifying the simulation runtime without affecting already developed models, tools or visualization engines. The same tools and interfaces can be used to facilitate model definition, whether those models will run in a single processor, in a parallel distributed environment or an embedded system.

CD++Builder is implemented on top of several well-known Eclipse frameworks, which provide the overall user interface and core plug-in services. A core requirement for CD++Builder was to allow easy extensibility, as new features are continuously being added and developed in geographically distant places. The Eclipse plug-in architecture enables developing new decoupled features into CD++Builder and integrate them seamlessly. One example of CD++Builder extensibility is the CD++Repository [18], an internet searchable database of CD++ models, which was developed in parallel with the present work in a totally independent way, and has been easily integrated as a part of the same software package.

To implement the graphical editors discussed earlier, several frameworks have been considered. Some of them include the basic graphic libraries Standard Widget Toolkit (SWT), the Abstract Window Toolkit (AWT) and Swing; others more specific include Draw2D and the Graphical Editing Framework (GEF). The first three libraries are based on Java and they provide general GUI controls useful for building form windows. Nevertheless, they are not practical for manipulating figures and shapes, and they do not provide any special infrastructure for Eclipse-based editors. Figures are the building blocks for Draw2D that builds on top of the SWT library. GEF allows generating a graphical editor based on an existing application model [19]. Due to these reasons, we choose Eclipse's Graphical Modeling Framework (GMF), as this library [20] acts as a bridge between GEF and Eclipse Modeling Framework (EMF) and it specifically tackles the creation of graphical Eclipse-based editors. GMF also relies on the Model-View-Controller

(MVC) architectural pattern to separate the model from its graphical representation, which has been successfully used in other DEVS editors. A similar framework stack has also been used in [21] for graphical editors of visual languages.

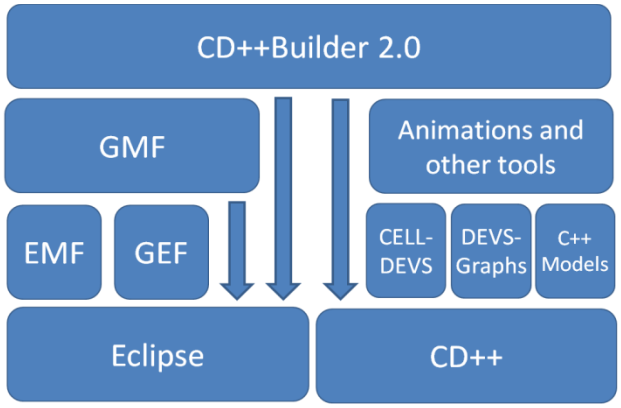


Figure 9 - CD++Builder technology architecture.

Figure 9 shows a description of the conceptual architecture of CD++Builder, considering the tools and software packages used. As we can see, EMF is used to define model entities (the *model* part of MVC), as it provides several services for specifying and maintaining entities. The model can be specified in XMI format, or in a graphical editor, which afterwards EMF uses to generate Java classes and interfaces. The Java classes generated implement the observable pattern, providing methods that notify whenever one of their properties has changed. This greatly helps in keeping the model completely decoupled from the rest of the implementation. Custom code and methods to provide extra behavior to the *model* portion of the architecture can be added to these classes. EMF recognizes special code comments in customized methods not to overwrite them when the *model* is regenerated. EMF also provides persistence and validation services for generated models. A detailed description of the model used to represent DEVS entities in CD++Builder can be found in [22].

GEF and GMF supply base classes which we extended to implement the *view* and *controller* parts of the MVC pattern. GEF extends Draw2D to make it easier to create a graphic representation of the model and provides several base Eclipse editor implementations. GEF controllers need to be provided with a *model* that exposes its properties and notifies whenever a change occurs.

The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF [23]. GMF runtime can be seen as a white-box framework as it combines and extends EMF generated models with GEF's controllers and views, and provides additional services such as transactional support. A *generative* part can be seen as a black-box framework as it allows defining meta-model information in XML files (graphical editors are provided for this purpose), which are used afterwards to generate Eclipse editors code.

In order to be able to generate new graphical editors code based on an EMF model, GMF needs to be provided with three meta-model files: graphical definition model (*.gmfgraph*), tooling definition model (*.gmftool*), and a mapping model (*.gmfmap*). *gmfgraph XML* file describe the shapes and figures that are going to be used in the editor, together with their properties and how they will be composed and layout. The *gmftool* file is used to describe diagram palette tools set, like Selection Tool, Zoom Tool, Creation Tool, etc, and how they will be grouped and shown. The *gmfmap* file references both previous files, and maps EMF model entities to a graphical representation (defined in *gmfgraph*) and associated tools (defined in *gmftool*).

For the purposes of CD++Builder editors, GMF code generation facilities were used in the beginning to define the general editors look and feel, layout and behavior. Nevertheless many necessary features have been developed, customizing and extending the generated code. GMF generates a decoupled infrastructure, where controllers, views and Eclipse editors implementation are separated from the model, which is kept in a separated project. This suites CD++Builder's requirements as the model can be reused by other CD++ or DEVS plug-ins without the need to depend on the editors implementation. The model is completely agnostic from graphical and edition details.

5. CONCLUSIONS

We presented a new architecture and the new features available in CD++Builder. This Eclipse plug-in is intended to facilitate the process of modeling and simulation with the CD++ simulator. The tool now:

- Provides an Integrated Development Environment (IDE) for all modeling and simulation tasks (modeling, compiling, simulation execution and analysis).
- Supplies editors that support the complete modeling cycle to be performed in a graphical manner.
- Includes C++ code templates to aid in the implementation of atomic models, while keeping the graphical representation of these models consistent with their C++ code.
- Supports extensibility and development of new features into the environment, including automated regression testing capabilities.

We showed how CD++Builder, in contrast with previous CD++ tools, provides a unified user interface under Eclipse for all the tasks involved in creating and

updating DEVS models, compiling new CD++ atomic models, executing simulations and analyzing results. We also showed how new DEVS graphical editors have been integrated into the Eclipse IDE that facilitates model creation, maintainability and comprehension. These editors are based on CD++'s high-level languages to represent model definition. Graphical information is stored independently and is kept synchronized with model definition automatically.

Additionally, DEVS-Graphs and C++ atomic models can now be used to define atomic model behavior. Modeling and definition of new C++ atomic models is simplified by auto-generated code templates, which are kept synchronized with their graphical representation. In addition, the CDT Eclipse plug-in is used for highlighting of C++ code.

Issues about usability and modeling limitations have been overcome with new editors, a tool for easier model reuse, a coupled model editor with discovery, and new install and update mechanisms. In the future, we will synchronize the new right tool pane with the online CD++Repository, to extend the set of models to be reused and facilitate searching and uploading models.

Having all features integrated into the Eclipse environment allows for easy extensibility by adding new plug-ins. An example of this is the Virtual Laboratory of Model-Based Development for Network Processors, (NP) currently under construction by our group. The Lab is fully based on CD++Builder, and is targeted to design advanced embedded control algorithms for the Intel IXP family of NPs [24]. CD++Builder provides a transparent interface for dealing with the intricacies of the target hardware such as compiling, downloading and monitoring models for their real time execution on an IXP chip. It also provides an integrated environment for mixing DEVS models with low-level hardware-specific drivers, making the simulator interact with real network signals in a Hardware In the Loop fashion.

References

[1] Zeigler, B; Praehofer, H; Kim, T. 2000, "Theory of Modeling and Simulation", 2nd Edition. Academic Press,

[2] Wainer, G. 2002. "CD++: A Toolkit to Define Discrete Event Models". Software - Practice and Experience, Vol. 32, No.13, (November): 1261-1306.

[3] Chidisiuc, C.; Wainer G. 2007, "CD++Builder: An Eclipse-Based IDE for DEVS Modeling". Proceedings of SpringSim 2007. Norfolk, VA. USA.

[4] Christen, G.; Dobniewski, A.; Wainer, G. 2004, "Modeling state-based DEVS models CD++". Proceedings of Advanced Simulation Technologies, Arlington, VA.

[5] Budinsky, F; Steinberg, D.; Merks, E.; Ellersick, R.; Grose, T.. "Eclipse Modeling Framework". Addison-Wesley Professional, 2003.

[6] Praehofer, H.; Pree, D. 1993, "Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs". 25th Winter Simulation Conference, Los Angeles, CA.

[7] Sarjoughian, H; Zeigler, B. 1998, "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment". Proceedings of the International Conference on Web-based Modeling & Simulation, San Diego, CA.

[8] Sungung, K.; Sarjoughian, H.; Elamvazhuthi, V. 2009. "DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring". Spring Simulation Multi-conference, San Diego, CA.

[9] Sarjoughian, H.; Elamvazhuthi, V. 2009. "CoSMos: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation". Proceedings of SIMUTools 2009, Rome, Italy.

[10] Filippi, J-B.; Delhom, J.; Bernardi, F. 2002. "The JDEVS Environmental Modeling and Simulation Environment," Proceedings of the first Biennial Meeting of iEMSs. Lugano, Switzerland.

[11] Traoré, M. 2008, "SimStudio: a next generation modeling and simulation framework". Proceedings of SIMUTools 2008. Marseille, France.

[12] Quesnel, G.; Duboz, R.; Ramat, E.; Traoré, M. 2007, "VLE: a multimodeling and simulation environment". Proceedings of Summer Computer Simulation Conference. San Diego, CA.

[13] Pagliero, E; Lapadula, M; Kofman, E. 2003, "PowerDEVS. An Integrated Tool for Discrete Event Simulation". (in Spanish). Proceedings of RPIC, San Nicolas, Argentina.

[14] Wainer, G. 2009, "Discrete-Event Modeling and Simulation: a Practitioner's approach". CRC Press.

[15] OMG/XMI: XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998.

[16] Beck, K."Test-Driven Development". Addison-Wesley, 2003.

[17] Massol, V; Husted, T.. "JUnit in Action". Manning Publications, 2003.

[18] Chreyh, R.; Wainer, G. 2009, "CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames". Proceedings of SpringSim'09, San Diego, CA.

[19] Eclipse Consortium. 2009. "Eclipse Graphical Editing Framework (GEF) - Version 3.4", available at http://www.eclipse.org/gef. [Accessed on Nov. 18, 2009].

[20] Eclipse Consortium. 2009. "Eclipse Graphical Modeling Framework (GMF)" http://www.eclipse.org/gmf. [Accessed on November 18, 2009].

[21] Ehrig, K; Ermel, C; Hansgen, S; Taentzer, G. 2005. "Generation of visual editors as eclipse plug-ins". 20th

IEEE/ACM International Conference on Automated software engineering, Long Beach, CA, USA.
[22] Bonaventura, M.; Wainer, G., Castro, R. 2009 "Advanced Environment for Discrete Event Simulation". Internal Report, Carleton University, Ottawa (submitted).
[23] Shatalin, A; Tikhomirov, A. 2006, "Graphical Modeling Framework Architecture Overview", Eclipse Modeling Symposium, 2006.
[24] Castro, R.; Kofman, E. and Wainer, G. 2009, "A DEVS-based End-to-end Methodology for Hybrid Control of Embedded Networking Systems", 3rd. IFAC Conference on Analysis and Design of Hybrid Systems, Zaragoza, Spain.