UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Carrera de Ingeniero Informático

ESTUDIO COMPARATIVO DE HERRAMIENTAS PARA LA SIMULACIÓN DE MODELOS DEVS

ALBERTO IBÁÑEZ BRILLAS

Dirigido por: ALFONSO URQUÍA MORALEDA

Curso: 2009 / 2010 (Marzo 2010)

ESTUDIO COMPARATIVO DE HERRAMIENTAS PARA LA

SIMULACIÓN DE MODELOS DEVS

Proyecto de Fin de Carrera de modalidad *oferta específica*

Realizado por: ALBERTO IBÁÑEZ BRILLAS (firma)

Dirigido por: ALFONSO URQUÍA MORALEDA (firma)

Tribunal calificador:

Presidente: D./Da..

(firma)

Secretario: D./Da..

(firma)

Vocal: D./Da...

(firma)

Fecha de lectura y defensa: ....

Calificación: ....

Resumen

En este Proyecto se lleva a cabo un análisis comparativo de tres herramientas software para la simulación de modelos de eventos discretos escritos mediante el formalismo DEVS. El análisis se realiza en términos de facilidad de uso y de las capacidades de cada herramienta para la descripción de los modelos. Estas herramientas son las siguientes:

- DEVSJAVA: Una librería de clases Java desarrollada en la Universidad de Arizona (Tucson, EEUU).
- CD++: Una librería de clases C++ desarrollada en la Universidad de Carleton (Ottawa, Canadá).
- El lenguaje de modelado propuesto en el Capítulo 17 del texto "Theory of Modelling and Simulation" de B.P. Zeigler et al.

DEVSJAVA y CD++ también son comparadas en términos de simulación y de las facilidades que ofrecen para la visualización de los resultados.

En la realización del análisis comparativo se han utilizado dos modelos de diferente complejidad. El primer modelo es un autómata celular bidimensional que describe el proceso de migración de metales pesados radioactivos en un terreno. El segundo modelo reproduce el comportamiento

de una gasolinera compuesta por varios surtidores y un conjunto de cajas para pagar.

Incluye instrucciones para la instalación de las herramientas DEVSJAVA y CD++. Además contiene un tutorial de introducción para la descripción de modelos, la simulación y la visualización de resultados con DEVSJAVA y CD++. Se suponen conocimientos acerca de la metodología DEVS y de los fundamentos de programación en Java y C++.

Lista de Palabras Clave

Simulación, modelado de eventos discretos, DEVS, DEVSJAVA, CD++, autómata celular.

Abstract

Three simulation tools that support the Discrete Event system Specification (DEVS) formalism are compared in terms of simplicity of use and model description capabilities. These tools are the following:

- DEVSJAVA: a Java library developed at University of Arizona (Tucson, USA).
- CD++: a C++ library developed at Carleton University (Ottawa, Canada).
- The modeling language proposed in Chapter 17 of "Theory of Modeling and Simulation", by B. P. Zeigler et al.

DEVSJAVA and CD++ are also compared in terms of simulation performance and facilities provided for result visualization.

Two models of different complexity have been used in this comparative analysis. The first model is a 2D cellular automata that describes the migration of radioactive heavy metals in soil. The second model reproduces the behavior of a gas station composed of several gas pumps and cash desks.

Installation instructions for DEVSJAVA and CD++ are included in this thesis. In addition, a tutorial introduction to model description, simulation and result visualization using DEVSJAVA and CD++ is provided. Previous knowledge of DEVS methodology, and Java and C++ programming is assumed.

Keywords

Simulation, discrete-event modeling, DEVS, DEVSJAVA, CD++, cellular automata.

Índice

Página

Lista de Figuras. 9

Capítulo1. INTRODUCCIÓN, OBJETIVOS Y ESTRUCTURA. 11
1.1 Introducción. 11
1.2 Objetivos del Proyecto. 18
1.3 Estructura del Proyecto. 19
1.4 Estructura del CD del Proyecto. 20
1.5 Conclusiones. 21

Capítulo 2. HERRAMIENTAS DE SIMULACIÓN. 23
2.1 Introducción. 23
2.2 DEVSJAVA. 24
2.2.1 Instalación de DEVSJAVA y creación de un Proyecto. 26
2.3 CD++. 31
2.3.1 Instalación de CD++ y creación de un Proyecto. 34
2.4 Lenguaje de Definición DEVS. 39
2.5 Conclusiones. 43

Capítulo 3. MODELADO MEDIANTE AUTÓMATA CELULAR. .45
3.1 Introducción. 45
3.2 Definición del Modelo. 48
3.3 Descripción con el Lenguaje de Definición DEVS. 52
3.4 Descripción y Simulación con DEVSJAVA. 54
3.5 Descripción y Simulación con CD++. 61
3.6 Conclusiones. 71

Capítulo 4. MODELADO DE UN SISTEMA LOGÍSTICO. 73
4.1 Introducción. 73
4.2 Definición del Modelo. 74
4.2.1 Definición formal DEVS. 87
4.3 Descripción con el Lenguaje de Definición DEVS. 97
4.4 Descripción y Simulación con DEVSJAVA. 106
4.5 Descripción y Simulación con CD++. 124
4.6 Conclusiones. 144

Capítulo 5. CONCLUSIONES Y TRABAJOS FUTUROS. 145
5.1 Introducción. 145
5.2 Conclusiones. 145
5.3 Trabajos futuros. 148

Anexo A. CÓDIGO DEVSJAVA DEL AUTÓMATA CELULAR 149
A.1 CelularCell.java 149
A.2 CelularSpace.java 152

Anexo B. CÓDIGO DEVSJAVA DEL SISTEMA LOGÍSTICO 155
B.1 Generador.java 155
B.2 Transd.java 156
B.3 Ef.java 160
B.4 Entrada.java 161
B.5 Surtidor.java 164
B.6 TransdSurt.java 168
B.7 ColaCajas.java 175
B.8 CajaSimp.java 179
B.9 TransdCaja.java 182
B.10 SetCajas.java 186
B.11 NetGasolinera.java 187

Anexo C. CÓDIGO CD++ DEL SISTEMA LOGÍSTICO 191
C.1 register.cpp 191
C.2 generador.h 192
C.3 generador.cpp 192
C.4 transd.h 195
C.5 transd.cpp 196
C.6 ef.ma 198
C.7 entrada.h 199
C.8 entrada.h 200
C.9 surtidor.h 202
C.10 surtidor.cpp 204
C.11 transdSurt.h 207
C.12 transdSurt.cpp 208
C.13 colaCajas.h 214
C.14 colaCajas.cpp 215
C.15 cajaSimple.h 219
C.16 cjaSimple.cpp 219
C.17 transdCaja.h 221
C.18 transdCaja.cpp 223
C.19 setCajas.ma 226
C.20 netGasolinera.ma 226

Lista de Referencias y Bibliografía 229

Siglas, Abreviaturas y Acrónimos 231

Lista de Figuras

Página

Figura 1.1: Comportamiento E/S de un modelo DEVS con un puerto de entrada y uno de salida. 13

Figura 1.2: Ejemplo de funcionamiento de un modelo DEVS. 16

Figura 1.3: Contenido del CD del Proyecto. 20

Figura 2.1: Instalación de DEVSJAVA y creación de un Proyecto. Paso 5. 27

Figura 2.2: Instalación de DEVSJAVA y creación de un Proyecto. Paso 6. 28

Figura 2.3: Instalación de DEVSJAVA y creación de un Proyecto. Paso 8. 29

Figura 2.4: Instalación de DEVSJAVA y creación de un Proyecto. Paso 9. 29

Figura 2.5: Instalación de DEVSJAVA y creación de un Proyecto. Paso 11. 30

Figura 2.6: Creación de un proyecto en CD++. Paso 1 36

Figura 2.7: Creación de un proyecto en CD++. Paso 2 36

Figura 2.8: Creación de un proyecto en CD++. Paso 4 37

Figura 2.9: Creación de un proyecto en CD++. Paso 6 38

Figura 3.1: Comportamiento de la difusión (a) y de la advección (b). 47

Figura 3.2: Puertos de entrada y salida de un célula. 49

Figura 3.3: Conexiones entre los puertos de una célula con los de cada una de sus vecinas. 50

Figura 3.4: Aspecto de Eclipse después de crear el proyecto y las clases necesarias para el modelado. 55

Figura 3.5: Clasificación de los colores de la visualización según los valores del parámetro nivel. 58

Figura 3.6: Primeros pasos de la simulación. 58

Figura 3.7: Desplazamiento de los contaminantes. 59

Figura 3.8: Los contaminantes abandonan el espacio celular. 60

Figura 3.9: Desaparición de los contaminantes. 60

Figura 3.10: Aspecto de Eclipse al finalizar la simulación. 61

Figura 3.11: Aspecto de Eclipse durante la simulación. 66

Figura 3.12: Paleta de colores para la visualización. 67

Figura 3.13: La aplicación *Cell-DEVS animation* antes del comienzo de la visualización. 67

Figura 3.14: Primeros pasos de la simulación. 68

Figura 3.15: Desplazamiento de los contaminantes. 69

Figura 3.16: Los contaminantes abandonan el espacio celular. 69

Figura 3.17: Desaparición de los contaminantes. 70

Figura 3.18: Aspecto de *Cell-DEVS animation* al final de la visualización. 70

Figura 4.1: Estructura del modelo gasolinera. 74

Figura 4.2: Diagrama de transición de estados del modelo atómico *surtidor*. 81

Figura 4.3: Diagrama de transición de estados del modelo atómico *colaCajas*. 84

Figura 4.4: Diagrama de transición de estados del modelo atómico *cajaSimp*. 86

Figura 4.5: Aspecto del entorno Eclipse con todos los archivos del modelo *Gasolinera* 119

Figura 4.6: Ventana principal de *SimView* antes de comenzar la simulación. 120

Figura 4.7: Ventana principal de *SimView* en pausa durante la simulación. 121

Figura 4.8: Resultado de situar el cursor sobre *ColaCajas* durante una pausa de la simulación. 122

Figura 4.9: Aspecto de la ventana *Personas en Caja* durante una pausa de la simulación. 123

Figura 4.10: Aspecto de las ventanas *Coches en Surtidor 1, 2 y 3* durante una pausa de la simulación. 123

Figura 4.11: Aspecto de la ventana *Simulate Project* con los datos necesarios para la simulación. 139

Figura 4.12: Aspecto de Eclipse después de crear todos los archivos del modelo *gasolineraCD++*. 140

Figura 4.13: Aspecto de la ventana *Atomic Animate* que contiene las gráficas del modelo. 141

Figura 4.14: Gráfica correspondiente al puerto *outQ* del modelo *colaCajas*. 142

1

Introducción, Objetivos y Estructura

1.1 Introducción

Según la definición dada por Robert E. Shannon: "Simulación es el proceso de diseñar y desarrollar un modelo de un sistema o proceso real y conducir experimentos con el propósito de entender el comportamiento del sistema o evaluar varias estrategias (dentro de límites impuestos por un criterio o conjunto de criterios) para la operación del sistema” [Shannon76].

La simulación, por tanto, es una herramienta muy importante en todas las áreas de investigación hoy en día debido a que nos permite analizar y experimentar con modelos de sistemas reales sin necesidad de disponer de ellos físicamente.

Para la simulación por ordenador se utilizan modelos matemáticos, los cuales se estudian mediante la aplicación de métodos numéricos. Existen diferentes técnicas para representar las distintas clases de sistemas reales.

Al principio de la década de los 70 del pasado siglo se empezaron a desarrollar la teoría y las metodologías de modelado y simulación basadas en Eventos Discretos, que dieron lugar al paradigma de simulación para los Sistemas Dinámicos de Eventos Discretos (DEDS – Discrete Events Dynamic Systems). Este paradigma asume que el sistema simulado sólo cambia de estado en puntos discretos de tiempo simulado, es decir, que el modelo cambia de estado ante la ocurrencia de un evento.

Se han desarrollado diversas aproximaciones para modelar sistemas de eventos discretos. Bernard P. Zeigler [Zeigler76] propuso un mecanismo de simulación jerárquica conocido como DEVS (Discrete EVent system Specification). Permite una descripción modular de los fenómenos a modelar, aproximación modular, y ataca la complejidad usando una aproximación jerárquica.

DEVS es un formalismo universal para modelar y simular DEDS. El sistema se describe como un conjunto compuesto de una base de tiempo, entradas, salidas y funciones para calcular los siguientes estados y salidas. El formalismo define cómo generar nuevos valores para las variables, y los momentos en los que estos valores deben cambiar. Los intervalos de tiempo entre ocurrencias son variables.

Un modelo DEVS se construye en base a un conjunto de modelos básicos, llamados atómicos, que se combinan para formar modelos acoplados. Los modelos atómicos son objetos independientes modulares, con puertos de

entrada y puertos de salida, variables de estado y parámetros, funciones de transición externa, de transición interna, de salida y avance de tiempo.

El formalismo DEVS permite representar cualquier sistema que satisfaga la condición de que, en cualquier intervalo acotado de tiempo, el sistema experimenta un número finito de cambios (eventos). La trayectoria de entrada al sistema es una secuencia ordenada de eventos, y la función de transición del modelo limita la trayectoria de salida para que sea también una secuencia de eventos.

Figura 1.1: Comportamiento E/S de un modelo DEVS con un puerto de entrada y uno de salida.

En la Figura 1.1 se muestra el comportamiento entrada/salida de un modelo DEVS atómico, con un solo puerto de entrada y un solo puerto de salida. Su especificación es la tupla siguiente:

$$DEVS = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, t_a >$$

donde:

X	Conjunto de todos los posibles valores de un evento de entrada.
S	Conjunto de posibles estados secuenciales.
Y	Conjunto de posibles valores de los eventos de salida.

$\delta_{int}: S \rightarrow S$	Función de transición interna. Describe el estado que sucederá al estado actual.
$\delta_{ext}: Q \times X \rightarrow S$	Función de transición externa. Describe la respuesta del modelo a un evento de entrada, mostrando cuál será el nuevo estado, teniendo en cuenta la entrada producida, el estado actual y el tiempo que ha transcurrido desde la última transición de estado. $Q = \{(s,e) \mid s \in S, e \in [0, t_a(s)]\}$
$\lambda: S \rightarrow Y$	Función de salida. Define cómo un estado genera un evento de salida, cuando se termina el tiempo que tenía asignado.
$ta: S \rightarrow R^{+}_{0,\infty}$	Función de avance de tiempo. Muestra el tiempo que permanecerá el modelo en el estado s, en ausencia de eventos de entrada.

En el caso de que el modelo tenga más de un puerto de entrada o de salida, y que se puedan recibir varios mensajes simultáneamente, es más conveniente utilizar la especificación del formalismo DEVS paralelo, en el que un modelo atómico se define mediante la siguiente tupla:

$$DEVS = \langle X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{cont}, \lambda, ta \rangle$$

donde:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$	*Inports* es el conjunto de puertos de entrada, y X_p el conjunto de los posibles valores de los eventos de entrada en el puerto *p*.
S	Conjunto de posibles estados secuenciales.
$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$	*Outports* es el conjunto de puertos de salida, e Y_p el conjunto de los posibles valores de los eventos de salida en el puerto *p*.
$\delta_{int}: S \rightarrow S$	Función de transición interna.

$\delta_{ext}: Q \times X^b_M \rightarrow S$	Función de transición externa. X^b_M representa una bolsa de parejas (puerto, evento). Las bolsas agrupan los eventos de entrada, para el caso de que lleguen simultáneamente a un mismo puerto, o a varios. $Q = \{(s,e) \mid s \in S, e \in [0, t_a(s)]\}$
$\delta_{cont}: Q \times X^b_M \rightarrow S$	Función de transición confluyente. Permite especificar cuál es el nuevo estado, cuando se produce la confluencia de una transición interna y una externa.
$\lambda: S \rightarrow Y^b$	Función de salida. Genera una bolsa de eventos de salida, Y^b. El sistema puede generar simultáneamente varios eventos, en uno o en varios de sus puertos de salida.
ta: $S \rightarrow R^+_{0,\infty}$	Función de avance de tiempo.

Hay dos variables que se emplean en todos los modelos DEVS para representar intervalos de tiempo: la variable e, que almacena el tiempo transcurrido desde la anterior transición de estado, ya sea interna o externa, y la variable σ, que almacena el tiempo que resta hasta el instante en que está planificada la siguiente transición interna.

Para comprender el mecanismo de funcionamiento de un modelo DEVS, es importante resaltar que durante la transición externa no se produce ningún evento de salida. Cuando tiene que realizarse una transición interna, se produce un evento de salida justo antes de efectuarse la transición interna.

La Figura 1.2 muestra un ejemplo del funcionamiento de un modelo DEVS con la secuencia de eventos y sus trayectorias.

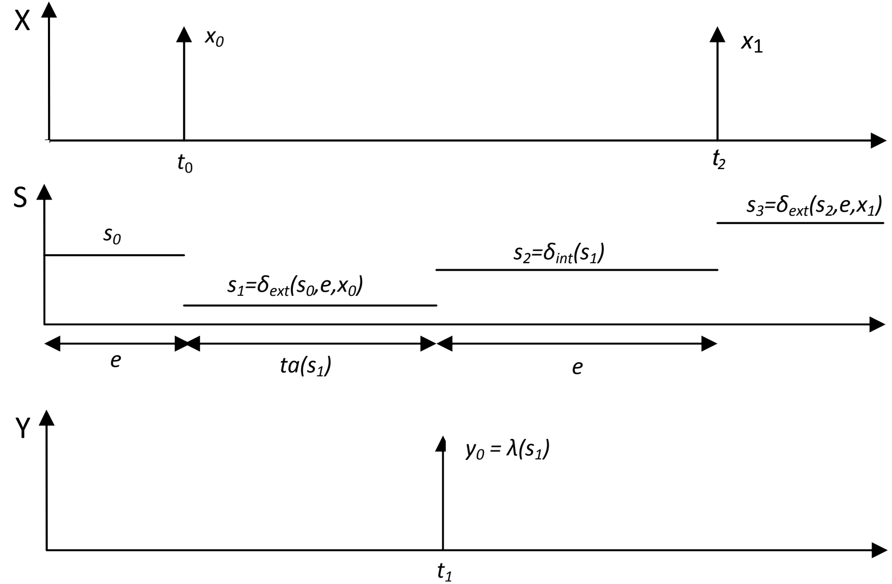


Figura 1.2: Ejemplo de funcionamiento de un modelo DEVS.

Los eventos de entrada ocurren en los instantes t_0 y t_2. En t_1 sucede un evento interno. En cada uno de los tres eventos se produce un cambio en el estado del sistema. La salida y_0 se produce justo antes de que se produzca el evento interno.

El sistema se encuentra inicialmente en el estado s_0, el cual tiene planificada una transición interna para cuando hayan transcurrido $t_a(s_0)$ unidades de tiempo. Antes de que se alcance ese momento, se produce un evento de entrada en el instante t_0. Como $t_0 < t_a(s_0)$, la transición interna prevista para el instante $t_a(s_0)$ no llega a producirse.

Como resultado del evento de entrada en t_0, se produce un cambio de estado. El nuevo estado se calcula mediante la función de transición externa:

$s_1 = \delta_{ext}(s_0, e, x_0)$, donde s_0 es el estado anterior, e es el tiempo que el sistema ha permanecido en dicho estado y x_0 es el valor del evento de entrada.

Ahora se planifica una transición interna para dentro de $t_a(s_1)$ unidades de tiempo, es decir, para el instante $t_0 + t_a(s_1)$. Como no se produce ningún evento externo antes de ese instante, la transición interna ocurre cuando estaba planificada. En ese instante se genera la salida, que se calcula con la función de salida, a partir del valor del estado: $y_0 = \lambda(s_1)$. A continuación se produce la transición de estado. El nuevo estado se calcula mediante la función de transición interna, pasándole como parámetro el estado anterior: $s_2 = \delta_{int}(s_1)$.

En este nuevo estado, se planifica la siguiente transición interna para el instante $t_1 + t_a(s_2)$, pero no llega a producirse porque lo impide la aparición de un nuevo evento externo en el instante $t_2 < t_1 + t_a(s_2)$.

El nuevo estado se calcula con la función de transición externa, se planifica la próxima transición interna, y así sucesivamente.

Un modelo acoplado describe los modelos que lo componen y cómo conectarlos entre sí, ya sean atómicos o acoplados, para formar un nuevo modelo. Este modelo puede ser empleado como componente, permitiendo una construcción jerárquica. La conexión de los modelos se realiza mediante los puertos de entrada y de salida.

Para mayor información sobre el formalismo DEVS pueden consultarse los textos [Urquía08] y [Wainer96], que son los que se han utilizado para redactar las líneas anteriores.

Existen diversas herramientas para el modelado de sistemas mediante DEVS, cada una con su propia implementación. Dos de ellas son:

- DEVSJAVA desarrollada en el lenguaje Java, por la Universidad de Arizona en Tucson, Estados Unidos, bajo la dirección de Bernard P. Zeigler.
- CD++ desarrollada en el lenguaje C++, por Gabriel Wainer en la Universidad de Carleton en Ottawa, Canadá.

Se han desarrollado otras, como JDEVS de la Universidad de Córcega en Italia, pero este Proyecto se va a centrar solamente en DEVSJAVA y CD++.

También se va a utilizar un lenguaje de modelado que ha sido específicamente diseñado para la descripción de modelos DEVS. Este lenguaje ha sido propuesto en el Capítulo 17 del texto “Theory of Modelling and Simulation” de B.P. Zeigler et al, [Zeigler00].

1.2 Objetivos del Proyecto

Este Proyecto tiene como objetivo realizar un análisis crítico, comparando la facilidad de uso y las capacidades para la descripción de los modelos DEVS, que poseen DEVSJAVA, CD++ y el lenguaje de modelado citado.

Para la realización de este análisis se utilizarán dos modelos de distinta complejidad. Ambos modelos se describirán con cada una de las tres herramientas, y se explicarán los pasos necesarios para su modelado y simulación con DEVSJAVA y CD++.

Asimismo, se va a desarrollar un tutorial sobre la instalación y el comienzo de un proyecto, con las herramientas DEVSJAVA y CD++.

1.3 Estructura del Proyecto

El Proyecto está estructurado en cinco capítulos, el primero de los cuales es esta Introducción, donde se ha explicado de una forma muy breve y concisa el funcionamiento del formalismo DEVS.

El Capítulo 2 contiene una explicación del contenido y funcionamiento de las herramientas DEVSJAVA y CD++, un tutorial de instalación y creación de un nuevo proyecto con ambas herramientas, y una descripción del lenguaje de modelado.

En el Capítulo 3 se efectúa la descripción de un autómata celular, y se desarrolla su modelado y simulación con las tres herramientas.

En el Capítulo 4 se describe un sistema logístico, desarrollando también su modelado y simulación con las tres herramientas.

El Capítulo 5 contiene las conclusiones del estudio y los trabajos futuros.

El código de los modelos figura en unos Anexos, excepto el código CD++ del autómata celular, que está completo en el Apartado 3.5.

En el Anexo A se incluye el código DEVSJAVA del autómata celular. En el Anexo B, el código DEVSJAVA del sistema logístico, y en el Anexo C el código CD++ del sistema logístico.

1.4 Estructura del CD del Proyecto

Junto a esta Memoria del Proyecto se incluye un CD con el código fuente de los modelos utilizados para el estudio de las herramientas DEVS. El contenido del CD se muestra en la Figura 1.3.

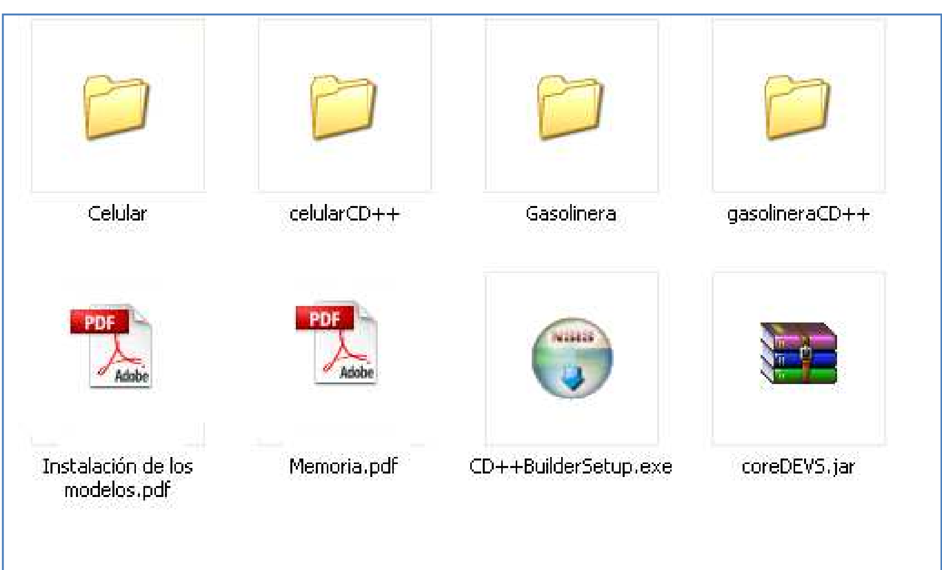


Figura 1.3: Contenido del CD del Proyecto.

Las cuatro carpetas contienen los archivos que se han creado para el modelado y simulación del autómata celular descrito en el Capítulo 3, y del

sistema logístico del Capítulo 4. La correspondencia entre las carpetas y los modelos es la siguiente:

- Celular: Autómata celular con DEVSJAVA.
- celularCD++: Autómata celular con CD++.
- Gasolinera: Sistema logístico con DEVSJAVA.
- gasolineraCD++: Sistema logístico con CD++.

El archivo *Instalación de los modelos.pdf* contiene una lista rápida, con los pasos necesarios para la instalación y visualización de cada uno de los modelos, en el entorno Eclipse.

Para los modelos DEVSJAVA es necesario utilizar la biblioteca de clases *CoreDEVS.jar* que también está incluida en el CD.

Para el caso de los modelos CD++, hay que tener instalado en Eclipse el plugin CD++Builder1.1.0.0. En el CD se incluye *CD++BuilderSetup.exe*, que es el archivo necesario para su instalación.

El archivo *Memoria.pdf* contiene esta Memoria del Proyecto.

1.5 Conclusiones

El formalismo DEVS es un mecanismo de simulación muy útil para modelar sistemas de eventos discretos.

Mediante la descripción del sistema a modelar como un conjunto, compuesto de una base de tiempo, entradas, salidas y funciones para calcular los siguientes estados y salidas, el formalismo define cómo generar nuevos valores para las variables, y los momentos en los que estos valores deben cambiar. Un modelo DEVS se construye en base a un conjunto de modelos básicos, llamados atómicos, que se combinan para formar modelos acoplados.

DEVSJAVA y CD++ son dos herramientas para el modelado mediante DEVS. Ambas van a ser analizadas a lo largo de este Proyecto junto con el lenguaje de modelado, específicamente diseñado para la descripción de modelos DEVS, propuesto en el texto "Theory of Modelling and Simulation" de B.P. Zeigler et al, [Zeigler00].

A lo largo del Proyecto se describirán la instalación y el funcionamiento de estas herramientas. Mediante el modelado de dos ejemplos: un autómata celular y un sistema logístico, se explicará detalladamente su manejo y la complejidad del modelado con cada una de ellas. El código fuente de estos modelos está en los Anexos de esta Memoria y en el CD del Proyecto.

2
Herramientas de Simulación

2.1 Introducción

Las herramientas para la simulación de modelos DEVS que van a ser objeto del estudio comparativo de este Proyecto, como ya se ha explicado anteriormente, son DEVSJAVA, CD++ y el Lenguaje de Modelado propuesto por Bernard P. Zeigler.

En los siguientes apartados se van a comentar las características de cada una de ellas, explicando su instalación y los pasos necesarios para la creación de un proyecto en el entorno Eclipse. También se detallarán las páginas web de las que se pueden descargar, ya que se trata de herramientas de uso gratuito para fines educativos.

Para trabajar con estas herramientas, durante la realización del Proyecto se ha utilizado un ordenador personal PC con el sistema operativo Windows XP, y el entorno de desarrollo Eclipse en su versión 3.4.0.

2.2 DEVSJAVA

DEVSJAVA es una de las primeras implementaciones que se desarrollaron del formalismo DEVS. Es un entorno de simulación desarrollado por Bernard P. Zeigler y Hessam S. Sarjoughian en la Universidad de Arizona. Está compuesto de varios paquetes completamente escritos en el lenguaje Java.

La versión utilizada en este Proyecto es DEVSJAVA 3.0, y se ha utilizado con JDK 1.6.

DEVSJAVA 3.0 se distribuye en un archivo llamado *Core DEVSJAVA.jar*, que se puede bajar de la página de descargas de software que tiene la Universidad de Arizona:

http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA

En él se encuentran los paquetes con las clases necesarias para el modelado y la simulación, pero no se proporciona el código fuente. También se puede bajar desde esa misma página el archivo *IllustrationCode.zip*, que contiene una serie de paquetes dedicados a diferentes categorías de modelos DEVS, los cuales tienen ejemplos ilustrativos de los conceptos DEVS. Las clases contenidas en estos paquetes se pueden aplicar directamente, o adaptarlas para realizar el modelado de nuestros sistemas, ya que aquí sí se incluye el código fuente. Hay un tercer archivo, *Examples.zip*, que contiene tres proyectos completos, que sirven como una buena ayuda en los primeros pasos con esta herramienta.

También es posible obtener algunos proyectos creados por los estudiantes, y algunos exámenes.

El archivo *Core DEVSJAVA.jar* está compuesto por los siguientes paquetes, que contienen las clases necesarias:

- *GenCol* Las clases para representar las estructuras de datos necesarias para el modelado.
- *genDevs* En sus tres subpaquetes están las clases para la descripción del modelo, la simulación y la representación de los resultados de la simulación.
- *Random* Clases en las que se utilizan números aleatorios.
- *simView* Clases de una herramienta para la visualización de la estructura de los modelos y para el control de la ejecución de la simulación
- *statistics* Clases útiles para realizar análisis estadísticos.
- *util* Clases de uso general.

Los paquetes del archivo *IllustrationCode.zip* contienen clases muy útiles para el modelado. Están divididos en categorías según sea el sistema que se desea modelar. En *SimpArc* se encuentran los modelos básicos. Hay también ejemplos para modelos celulares, modelos continuos, con estructura variable, de pulsos o modelos de sistemas cuantizados. Antes de empezar a modelar conviene revisar estos paquetes para evitar hacer un trabajo que ya se encuentre hecho aquí.

El comportamiento de un modelo atómico se controla con un conjunto de métodos:

- *public void Deltxt(double e, double input)* Para describir la función de transición externa.
- *public void deltint()* Para la función de transición interna.
- *public message out()* Para la función de salida, o *public double sisoOut()* si se trata de un modelo con un solo puerto de salida.

Además existen métodos para todas las demás funciones de un modelo DEVS, como añadir puertos, comprobar el estado actual, etc.

Si se desea obtener mayor información sobre DEVSJAVA, puede consultarse el texto [Urquía08] que ha sido utilizado como base para esta introducción.

2.2.1 Instalación de DEVSJAVA y creación de un Proyecto

En la página web perteneciente a la Universidad de Arizona, *http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA*, están los archivos necesarios para trabajar con DEVSJAVA, y también una guía de instalación para el entorno JBuilder.

Para este Proyecto se ha utilizado el entorno Eclipse 3.4.0. A continuación se detallan los pasos a seguir para la instalación y creación de un proyecto en Eclipse:

1. Crear una carpeta con el nombre *devsjava* en un directorio del disco duro.
2. Descargar en la carpeta el archivo *Core DEVSJAVA.jar* y los que se deseen.
3. Abrir Eclipse con la Perspectiva Java. Para ello hacer click sucesivamente en *Window – Open Perspective – Java*.
4. Crear un nuevo proyecto haciendo click en *File – New – Project – JavaProject – Next*.

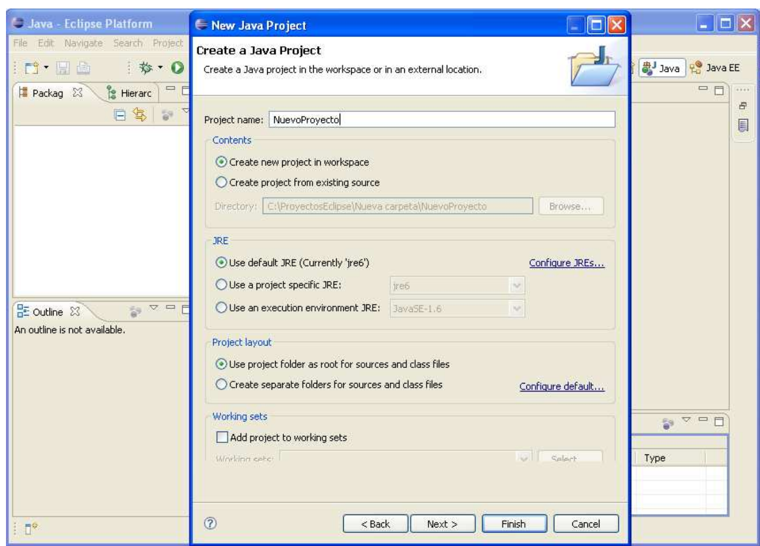


Figura 2.1: Instalación de DEVSJAVA y creación de un Proyecto. Paso 5.

5. En la ventana que aparece, poner el nombre al proyecto. Para proyectos pequeños puede activarse, en *Project layout*, la casilla correspondiente a *Use Project folder as root for sources and class files*. Hacer click en *Next*. Ver la Figura 2.1.

6. Seleccionar la pestaña *Libraries*, hacer click en *Add External JARs*, buscar el archivo c*oreDEVS.jar* en la carpeta en la que se ha guardado, seleccionarlo y hacer click en *Abrir*. Pulsar *Finish*. Ver la Figura 2.2.

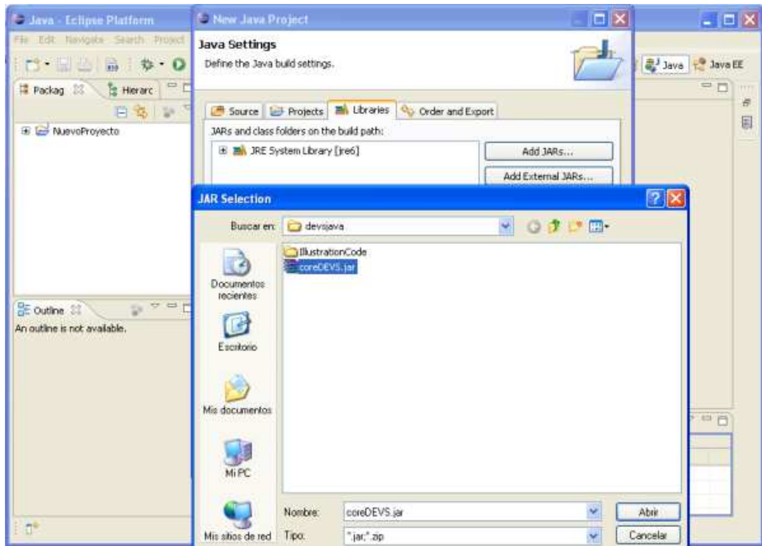


Figura 2.2: Instalación de DEVSJAVA y creación de un Proyecto. Paso 6.

7. Crear un paquete pulsando el botón derecho del ratón sobre el proyecto, que está en la ventana *Workspace*, a la izquierda del editor, y haciendo click en *New – Package*. Poner nombre al paquete. Aquí será donde iremos guardando las clases que vamos a ir creando para nuestro proyecto. Pulsar *Finish*.
8. Cada vez que queramos crear una nueva clase, pulsar el botón derecho del ratón sobre el paquete creado, hacer click en *New – Class*. Poner nombre a la clase. Pulsar *Finish*. Ver Figura 2.3.

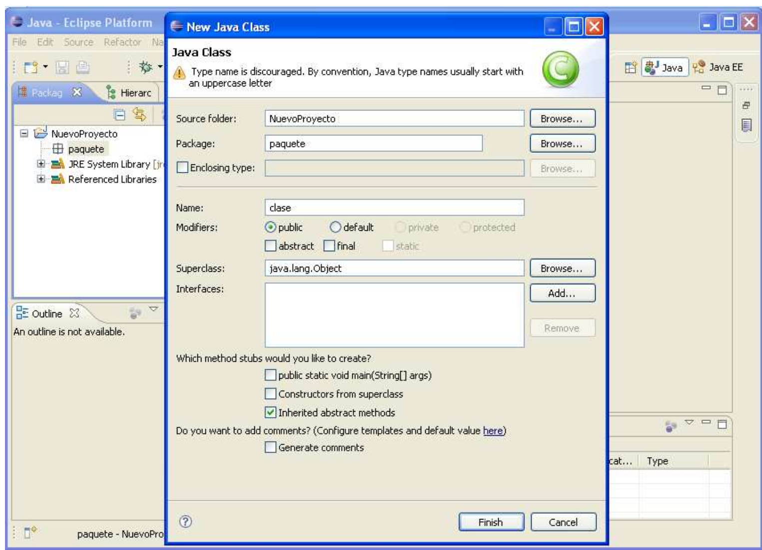


Figura 2.3: Instalación de DEVSJAVA y creación de un Proyecto. Paso 8.

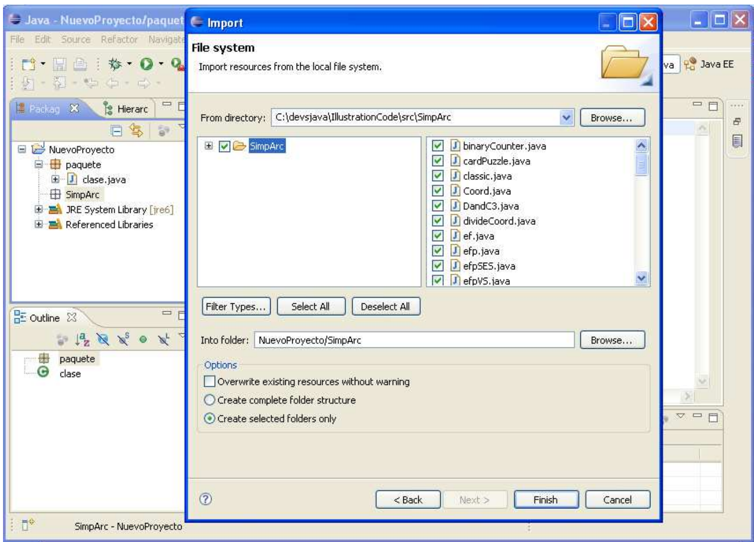


Figura 2.4: Instalación de DEVSJAVA y creación de un Proyecto. Paso 9.

9. Si se desea importar paquetes, por ejemplo alguno de los del archivo *IllustrationCode.zip*, se crea primero en el proyecto un paquete vacío y se le asigna el mismo nombre el paquete que se desea importar. Pulsar el botón derecho del ratón sobre el paquete creado, hacer click en *Import - File System* (o *Archive File* si lo vamos a importar de un archivo comprimido) – *Browse*, de

la ventana *From directory*, buscar el paquete, que será una carpeta, donde lo tengamos guardado, seleccionarlo y pulsar *Aceptar*. Si se quiere importar todas las clases del paquete, se debe señalar la carpeta que ha aparecido en la ventana izquierda y *Finish*. Si solamente se desea importar algunas de las clases pero no todas, se señalan en la ventana derecha y *Finish*. Ver Figura 2.4.

10. Para conseguir que se ejecute *SimView* al pulsar el botón *Run*, hay que pulsar el botón derecho del ratón sobre el proyecto, hacer click en *Run As – Java Application*, en la lista que aparece seleccionar *SimView-simView* y pulsar *OK*.

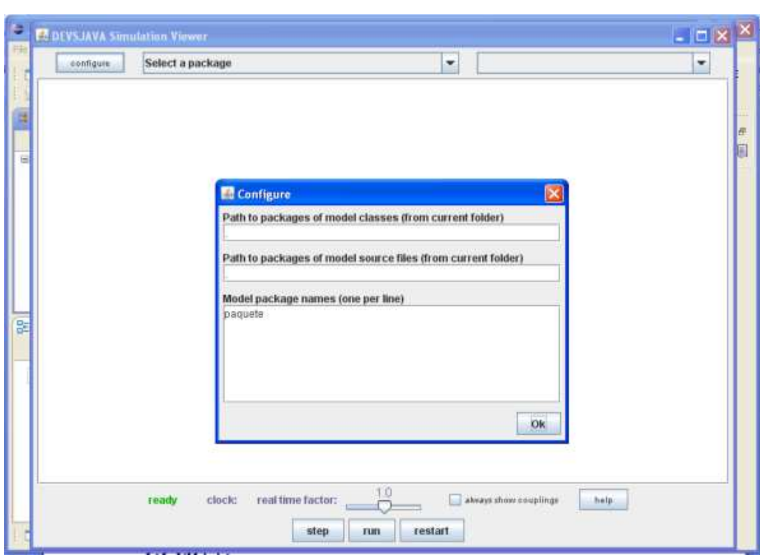


Figura 2.5 Instalación de DEVSJAVA y creación de un Proyecto. Paso 11.

11. Ejecutar *SimView* pulsando el botón *Run.* En la ventana pulsar *configure*. En *Model packages names* poner, en una línea nueva, el nombre del paquete que queremos visualizar y pulsar *Ok*. En *Select a package* seleccionar el paquete, y en *Select a model* el modelo que queremos visualizar. Ver Figura 2.5.

2.3 CD++

CD++ es una herramienta desarrollada por Gabriel Wainer, profesor de la Universidad de Carleton en Ottawa, Canadá. El lenguaje de programación que utiliza es C++.

Para la realización de este Proyecto se ha utilizado CD++Builder 1.1.0.0, que es un plugin para Eclipse.

En la página *http://cell-devs.sce.carleton.ca* existe mucha información sobre esta herramienta. Tiene un enlace desde donde se puede descargar el archivo *CD++BuilderSetup.exe* que incluye además de CD++Builder 1.1.0.0 otras aplicaciones, entre ellas el propio Eclipse. La página también contiene una colección de modelos realizados en CD++ que se pueden ver y descargar y que son una buena ayuda para empezar a modelar. Tiene un enlace para descargar un *Manual de Usuario* [Wainer05] muy útil.

Una vez instalado en nuestro ordenador podemos acceder a la carpeta *plugins* de Eclipse, y dentro de ella buscar la carpeta *CD++Builder1.1.0*. En ésta se encuentra la carpeta *internal* que contiene algunos modelos básicos. Unos solamente muestran el archivo con la definición del modelo, extensión "*.h*", y otros además añaden el archivo "*.cpp*", que contiene todo el código fuente de los métodos necesarios para el funcionamiento del modelo. Cualquier modelo atómico de un proyecto requiere la implementación de los dos archivos, *.h* y *.cpp*.

Todos los modelos atómicos derivan de la clase *Atomic*, que está en la carpeta *internal*, y es necesario sobrecargar los siguientes métodos que controlan su comportamiento:

- Función de transición externa:

  Model &Nombre::externalFunction (const ExternalMessage &msg)

- Función de transición interna:

  Model &Nombre::internalFunction (const InternalMessage &)

- Función de salida:

  Model &Nombre::outputFunction (const InternalMessage &msg)

donde *Nombre* es el nombre del modelo atómico y *Model* es la clase de la que deriva *Atomic.*

Para decirle a CD++ que se ha creado un nuevo modelo atómico, hay que incluir en el proyecto un archivo llamado *register.cpp*, que contenga el método *MainSimulator.registerNewAtomics()*, y registrar el modelo en él.

Además de los métodos mencionados, existen primitivas para interactuar con el simulador y realizar las operaciones de los métodos del modelo atómico.

Los modelos acoplados se definen utilizando un lenguaje de especificación especialmente definido con este propósito. Se debe crear en el proyecto un archivo con la extensión *.ma*, y definir en él los componentes que forman el modelo acoplado, los puertos que tiene y los enlaces entre ellos.

También se pueden definir parámetros de los modelos que forman el modelo acoplado. En el Manual de Usuario [Wainer05] viene bien explicado el lenguaje de especificación.

En CD++ los modelos basados en células, o modelos Cell-DEVS, son un caso especial de modelos acoplados, por lo que se pueden definir mediante un lenguaje de especificación, que es una ampliación del lenguaje de los modelos acoplados, y que incluye los parámetros necesarios para generar un espacio celular completo. El comportamiento de este espacio celular se controla mediante un conjunto de reglas, que se escriben con el lenguaje de especificación. Para generar el modelo se crea un archivo con extensión *.ma*, en el que se describen su composición y su comportamiento.

Para poder realizar una simulación del modelo, primero hay que compilarlo. En el siguiente apartado se describirá cómo efectuarlo con Eclipse. Una vez ejecutada la simulación del modelo, se obtienen dos archivos que son muy útiles para la visualización y para el seguimiento de todo el proceso que realiza el modelo. Estos archivos son:

- El archivo *Log* contiene todos los mensajes enviados entre los distintos modelos DEVS que componen el modelo acoplado que está siendo objeto de la simulación. Indica el tipo de mensaje, el momento en que se ha producido, y la unidad que lo envía y la que lo recibe.
- El archivo *Out* contiene todos los eventos que han sido generados por el simulador y que han salido del modelo. Incluye el

momento de la salida, el puerto y el valor. Para ello utiliza el siguiente formato: HH:MM:SS:MS: PORT VALUE

Existen varias utilidades para la visualización de modelos CD++. Con *CD++Builder* viene una de ellas, *CD++Modeler*, con la que es posible crear modelos de una forma gráfica, así como visualizarlos. Esta herramienta se puede ejecutar directamente desde Eclipse.

2.3.1 Instalación de CD++ y creación de un Proyecto.

En la página *http://cell-devs.sce.carleton.ca* se puede descargar el archivo *CD++BuilderSetup.exe*. Una vez guardado en el ordenador, se ejecuta y comienza la instalación, que va siendo guiada por una sucesión de ventanas, las cuales indican las aplicaciones que se pueden instalar y las acciones a realizar.

Nada más ejecutar el archivo, se descarga en el directorio raíz el archivo *CD++Builder Toolkit Installer. pdf,* que es una guía de instalación, y aparece la ventana de bienvenida en la pantalla.

Las aplicaciones que se pueden instalar son las siguientes:

- *JDK 1.6* Es necesario tener instalada una Java Virtual Machine en el ordenador para poder trabajar con Eclipse. Si el programa de instalación detecta una ya instalada, lo comunicará y proseguirá la instalación. En caso de que sea necesaria la instalación, es conveniente recordar que hay que

añadirlo al *Path* de Windows, en las *Variables de Entorno* de nuestro ordenador.

- *Cygwin* Es un entorno que incluye un compilador para C++ y otras herramientas que pueden ser necesarias, como el *make.* Es imprescindible descargarlo si no disponemos ya de alguna aplicación similar, porque Eclipse no lo incluye. También habrá que incluirlo en el *Path* de Windows.
- *Eclipse* Es el entorno de desarrollo integrado (IDE) elegido para trabajar con *CD++Builder*. Tampoco es necesario instalarlo si ya lo tenemos.
- *CD++Builder* Es el plugin para Eclipse que nos permitirá modelar con CD++. Al instalarlo se situará en la carpeta *plugins* de Eclipse directamente.
- *Blender* y *Python* Son aplicaciones para visualizar y animar simulaciones en 3D. No han sido utilizadas para la realización de este Proyecto.

Es conveniente consultar durante la instalación el archivo *CD++Builder Toolkit Installer. pdf*, que es una buena ayuda.

Los pasos que hay que efectuar para la creación de un proyecto son los siguientes:

1. Abrir Eclipse con la Perspectiva CD++Builder. Para ello hacer click sucesivamente en *Window – Open Perspective – Other*. En la ventana que aparece seleccionar *CD++Builder* y pulsar *OK*. Ver Figura 2.6.

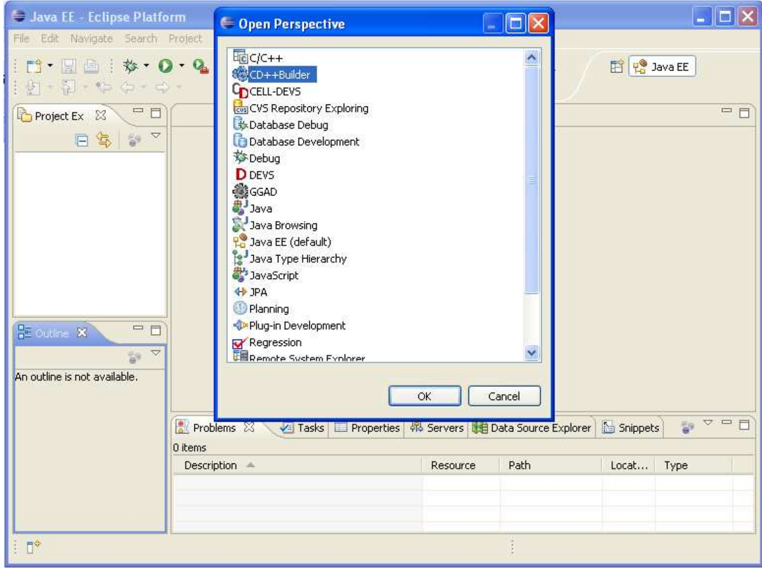


Figura 2.6 Creación de un Proyecto en CD++. Paso 1.

2. Hacer click en *File – New – Project – Other – CD++Builder – Next*. En la ventana que aparece poner el nombre del autor del proyecto, pulsar *Next*, en la nueva ventana poner el nombre del proyecto y pulsar *Finish*. Ver Figura 2.7.

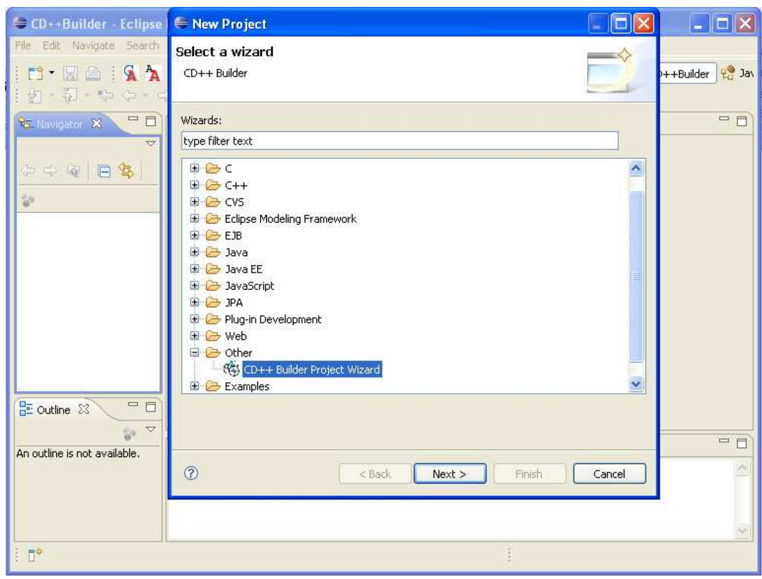


Figura 2.7 Creación de un Proyecto en CD++. Paso 2.

3. Para añadir un archivo al proyecto hay que seleccionarlo y hacer click en *File – New – File*, en la ventana que aparece, poner el nombre al archivo y pulsar

Finish. Junto al nombre del archivo hay que poner también la extensión. Según sea la extensión que le pongamos se creará un tipo de archivo u otro.

4. Para importar archivos al proyecto hay que seleccionarlo, hacer click en el botón derecho del ratón y pulsar *Import*. En la ventana abrir la carpeta *General* y seleccionar *File System* (o *Archive File* en el caso de que se trate de un archivo comprimido), pulsar *Next*. Pulsar *Browse* de la ventana *From directory*, buscar el paquete, que será una carpeta, donde lo tengamos guardado, seleccionarlo y *Aceptar*. Si se quiere importar todas las clases del paquete, se debe señalar la carpeta que ha aparecido en la ventana izquierda y *Finish*. Si solamente se desea importar algunas de las clases pero no todas, se señalan en la ventana derecha y *Finish*. Ver Figura 2.8.

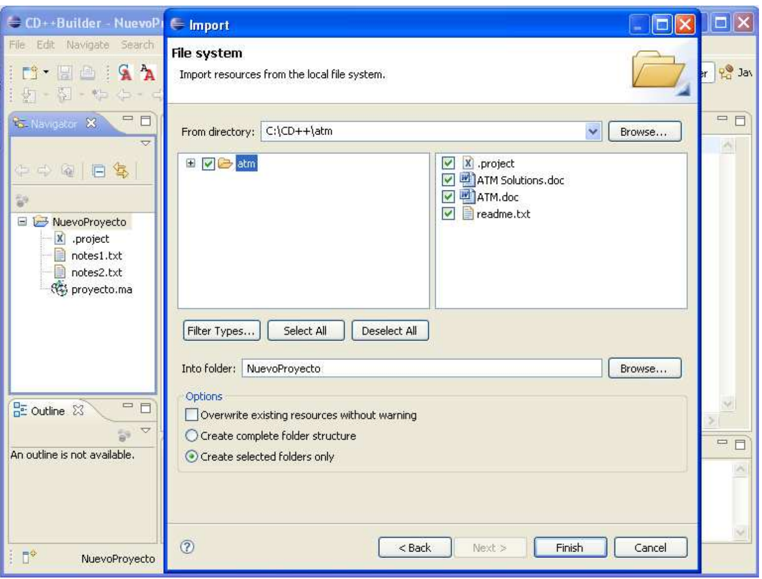


Figura 2.8 Creación de un Proyecto en CD++. Paso 4.

5. Una vez creado, hay que compilar el proyecto. Para ello primero se selecciona cualquier archivo del proyecto, y se hace click en el botón *Build* de la barra que contiene los botones de herramientas de CD++. El proyecto sólo debe

contener modelos DEVS. Esta acción compilará el proyecto y creará el archivo *simu.exe*, que es necesario para efectuar la simulación. Antes de que comience la compilación, en una ventana se preguntará si se quiere ejecutar en el modo *Verbose*, pulsar *Yes* y comenzará.

6. Para simular el proyecto hay que pulsar el botón *Simu* de la barra que contiene los botones de herramientas de CD++. Esto hará que aparezca un panel donde hay que especificar los parámetros de la simulación: el archivo *.ma* del modelo acoplado que se quiere simular, los archivos *.log* y *.out* donde se quieren obtener los resultados, el tiempo que tiene que durar la simulación y otros. Se puede crear un archivo *.bat* que sirva para no tener que introducir los parámetros cada vez que sea necesario realizar una simulación del modelo. Ver Figura 2.9.

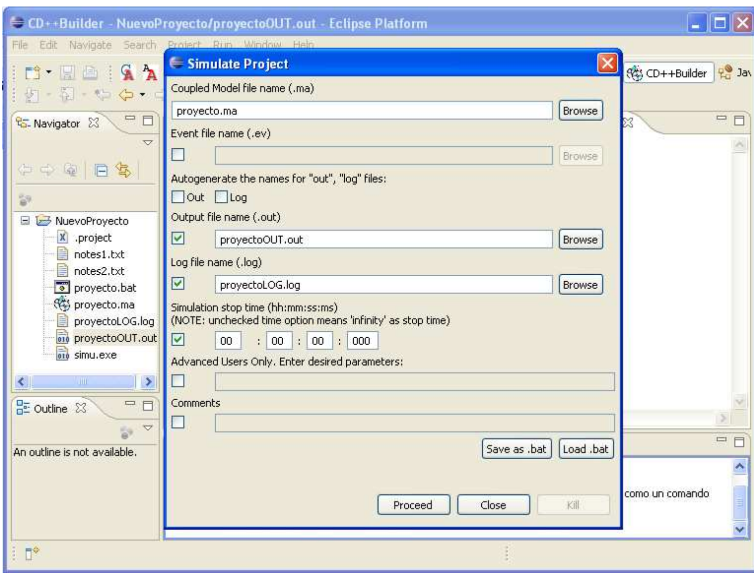


Figura 2.9 Creación de un Proyecto en CD++. Paso 6.

7. Para visualizar el resultado de la simulación, y según el tipo de modelo, se pueden utilizar los botones *Animate Coupled Model, Animate Atomic Model* o *Animate Cell-Dev Simulation* de la barra que contiene los botones de herramientas de CD++. También se puede utilizar el botón *Load CD++ Modeler* , de la misma barra, que abre la aplicación *CD++Modeler*, y desde ahí, pulsando en *Animate*, tenemos las tres opciones anteriores.

2.4 Lenguaje de Definición DEVS

El lenguaje de modelado que se va a utilizar es el Lenguaje de Definición DEVS que viene descrito en el Capítulo 17 del texto "Theory of Modelling and Simulation" de Bernard P. Zeigler et al, [Zeigler00].

Este lenguaje conserva la semántica de modelado del formalismo DEVS. Hay una correspondencia de uno a uno entre los elementos del formalismo y las sentencias del lenguaje de definición. Tiene un conjunto de palabras reservadas y unas reglas gramaticales, que permiten la descripción directa de los modelos DEVS.

El lenguaje de definición DEVS comienza con la declaración global del modelo, a continuación van la sección de definición de modelos atómicos y la sección de modelos acoplados.

```
begin description
overall model declaration -> overall model: model_name
```

```
atomic model definition section -> atomic_model_definition
                         || atomic_model_definition_section
coupled model definition section -> coupled_model_definition
                         || coupled_model_definition_section
end description
```

Tiene tres tipos de constantes: *string, integer y float* que se utilizan en las secciones de declaración de los modelos, para especificar la lista de eventos de entrada o salida, el rango de una variable y la lista de componentes.

Una variable se define con su nombre y su rango de valores.

Hay dos tipos de sentencias: las de asignación y las de comparación.

Las reglas gramaticales BNF (*Backus Normal Form*) son las siguientes:

```
constant_list -> constant; || constant;  constant_list
constant -> string_constant || integer_constant ||float_constant
variable_list -> variable_declaration
                  || variable_declaration  variable_list
variable_declaration -> variable  IN  (range_set)
range_set -> constant || constant,  range_set
statements -> assignment_statement || comparison_statement
assignment_statement -> string_constant  :=  expr
comparison_statement -> string_constant  relational_op  expr
expr -> constant  arithmetic_op  constant || constant
operators -> arithmetic_op || relational_op || logical_op
```

El lenguaje distingue entre mayúsculas y minúsculas.

Una constante de tipo *string* empieza siempre por una letra que puede ir seguida de letras, números o guiones bajos.

Los operadores aritméticos son: +, –, *, /.

Los operadores relacionales son: =, >, >=, <, <=, <>.

Y los operadores lógicos son: ||, &&, ~.

Los comentarios van encerrados entre /* y */ y pueden ocupar varias líneas, o empiezan con // ocupando una sola línea.

El formato general para la definición de un modelo atómico es:

```
atomic model model_name
      inports: input ports list;
      outports: output ports list;
      state variables: state variable list;
      phase: phase definition list;
      initial condition: state value assignment list;
      internal transition: internal state transition list;
      external transition: external state transition list;
      output function: output function list;
end model_name
```

input ports list y *output ports list* se definen mediante *constant_lists*.

state variable list se define mediante *variable_list*.

El estado inicial se representa como una lista de *assignment_statements* separadas por un punto y coma.

La forma general de *internal state transition* es:

"current_state => next_state".

La forma general de *external state transition* es:

*“current_state * input_event => next_state”*.

current_state y *next_state* se escriben mediante expresiones booleanas conjuntivas.

La función de salida se define como:

“current_state => output_event”.

Las reglas BNF correspondientes son las siguientes:

```
state_tran_list -> state_tran; || state_tran;  state_tran_list
state_tran -> internal_tran || external_tran
internal_tran -> state_cond => state_cond
external_tran -> state_cond  *  string_constant => state_cond
state_cond -> boolean_expr
boolean_expr -> conjunctive_expr ||
conjunctive_expr   ||(logical OR)   boolean_expr
conjunctive_expr -> statements ||
                  statements   &&(logical AND)   conjunctive_expr
output_function -> state_cond => string_constant
```

La definición de un modelo acoplado consta de un nombre, seguido de una sección de declaración de componentes, y una sección de declaración de enlaces.

```
coupled model model_name
      components: component model type list
      internal coupling: internal coupling list
      external input coupling: external input coupling list
      external output coupling: external output coupling list
end model_name
```

La *component model type list* se construye como una lista de definiciones de tipo de modelo, separadas por un punto y coma. La sintáxis de la definición de tipo de modelo es *“component_model model_name”*. El símbolo reservado ”→” representa la conexión entre componentes. El formato general de conexión es “*from_model.port_name → to_model.port_name”*. Las reglas BNF son:

```
component_type_def -> component_var
                      || component_var  component_type_def
component_var -> string_constant  string_constant;
coupling_list -> coupling_assignment
                      || coupling_assignment  coupling_list
coupling_assignment -> dotted_string_const  dotted_string_const
dotted_string_const ->  string_constant.string_constant
```

2.5 Conclusiones

Las herramientas DEVSJAVA y CD++ disponen de una biblioteca de clases que les permite el modelado de cualquier sistema mediante el formalismo DEVS. Se trata de herramientas de uso gratuito para fines educativos.

DEVSJAVA está implementada con el lenguaje Java y aquí se han descrito los pasos necesarios para su instalación y la creación de un nuevo proyecto, con el entorno de desarrollo integrado (IDE) Eclipse. La instalación es muy sencilla, ya que Eclipse contiene todo lo necesario para programar con

Java. Al crear un nuevo proyecto hay que añadirle siempre la biblioteca de clases de DEVSJAVA.

CD++ usa el lenguaje C++. En su versión *CD++Builder*, se distribuye como un *plugin* para Eclipse, incluyendo también las aplicaciones necesarias para su funcionamiento. También se ha descrito paso a paso su instalación y la creación de un nuevo proyecto con Eclipse. En la instalación es necesario instalar también el entorno *Cygwin*, para la programación con C++. Al crear un proyecto no es necesario añadirle nada, porque CD++ dispone de su propia perspectiva dentro de Eclipse.

El Lenguaje de Definición DEVS propuesto por Bernard P. Zeigler et al, está descrito en el Capítulo 17 del texto "Theory of Modelling and Simulation" de Bernard P. Zeigler et al, [Zeigler00]. Mediante la utilización de un conjunto de palabras reservadas y unas reglas gramaticales BNF (*Backus Normal Form*), es posible la descripción directa de los modelos DEVS.

3
Modelado mediante Autómata Celular

3.1 Introducción

Se ha elegido un modelo basado en autómata celular, para poder comprobar la facilidad de las herramientas en el modelado, simulación y visualización de un espacio celular, y la aplicación de las reglas de comportamiento de las células en dicho espacio.

El modelo con el que se va a realizar el análisis comparativo está tomado del artículo *“Modelling conservation quantities using Cellular Automata”* de K. Breitenecker et al, [Breitenec09], publicado en los Proceedings de MATHMOD 09. En él se analizan las características del proceso de propagación de los contaminantes, en un área que ha sido expuesta a elementos radioactivos. En dicho trabajo se destaca que los procesos más importantes y mejor conocidos, de todos los que intervienen en la migración de cualquier contaminante, son la *difusión* y la *advección*.

Existen otros procesos, como la *absorción* y la *adsorción*, que también influyen significativamente en la migración. Aunque son procesos muy bien observados es complicado expresarlos por medio de una simple fórmula matemática.

Por tanto, no es posible modelar el desplazamiento con un modelo de caja blanca (white-box). Pero como el comportamiento y las interacciones de todos los procesos son bien conocidos, será suficiente un modelo de caja negra (black-box), como el que ofrece un autómata celular. Para ello, se toman como procesos principales la *difusión* y la *advección*, pero las reglas de comportamiento del modelo se hallarán teniendo en cuenta a todos los demás procesos.

La *difusión* es un proceso de equilibrio físico, mediante el cual, las partículas de las áreas de alta concentración se desplazan hacia las áreas de menor concentración, para llegar a conseguir un equilibrio.

La *advección* es un mecanismo de transporte de una sustancia por medio del movimiento de un fluido.

El comportamiento elegido para la *difusión* es el de la regla de vecindad de *Von-Neumann* (*von-Neumann neighbourhood*). Cada célula envía contaminantes a sus cuatro células vecinas, pero también los recibe de ellas.

En la *advección* la célula se comporta según una simplificación de la regla de vecindad de *Moore* (*Moore neighbourhood*). Una célula recibe

contaminantes de su célula vecina superior, y se los envía a su vez a su célula vecina inferior. La dirección de desplazamiento se corresponderá con la inclinación natural que posea el terreno.

Los comportamientos de las células con la *difusión* y la *advección* se muestran en la Figura 3.1.

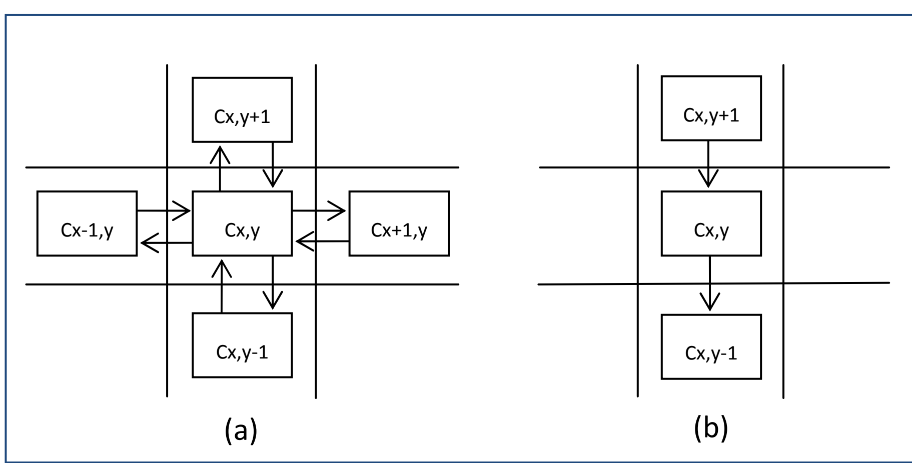


Figura 3.1 Comportamiento de la difusión (a) y de la advección (b).

En las reglas de comportamiento ya están consideradas las interacciones con los demás procesos. Las reglas propuestas para el modelo son las siguientes:

$$N_{dif}(x,y,t+1) = N(x,y,t) - 4dN(x,y,t) + d[N(x-1,y,t) + N(x+1,y,t) + N(x,y-1,t) + N(x,y+1,t)]$$

$$N_{adv}(x,y,t+1) = (1-v)N(x,y,t) + vN(x,y+1,t)$$

donde

$N(x,y,t)$ Indica el valor del nivel de contaminantes de la célula *(x, y)*, en el instante *t*.

$N_{dif}(x,y,t+1)$ Indica el valor del nivel de contaminantes, debido a la *difusión*, de la célula *(x, y)*, en el instante *t+1*.

$N_{adv}(x,y,t+1)$ Indica el valor del nivel de contaminantes, debido a la *advección*, de la célula *(x, y)*, en el instante *t+1*.

d Es el coeficiente de difusión.

v Es el coeficiente de advección.

3.2 Definición del Modelo

Este modelo es un ejemplo de autómata celular bidimensional. Para realizar el modelado, el terreno se define como un espacio celular bidimensional, que no cambia de forma durante todo el tiempo que dura la simulación. El espacio celular está compuesto por un número determinado de células, distribuidas uniformemente a lo largo de los ejes X e Y, cada una de las cuales está acoplada lateralmente a aquellas que están más próximas.

Cada célula dispone de una variable de estado en la que almacena el valor del nivel de contaminantes por unidad de área, que posee en cada momento. Todas actualizan su nivel por medio de una misma regla. Esta regla utiliza los niveles de las celulas vecinas y los coeficientes de difusión y advección. Todas las células actualizan su nivel a la vez, en cada paso del tiempo de la simulación. Los valores de la variable nivel para el momento *t+1*, se consiguen actualizando los valores del momento *t*.

La célula dispone de cuatro puertos de entrada y cinco puertos de salida, según muestra la Figura 3.2.

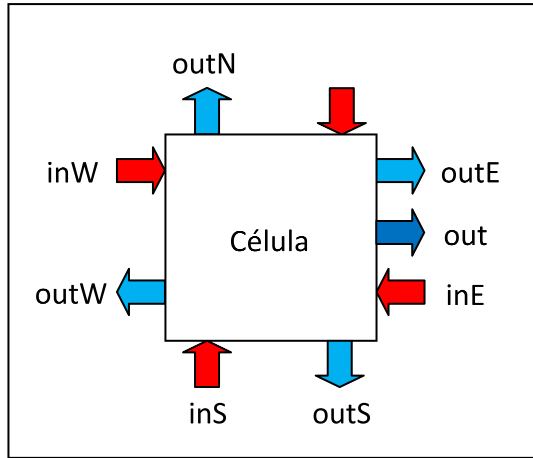


Figura 3.2: Puertos de entrada y salida de un célula.

A través de los puertos de entrada (inN, inE, inS, inW) la célula recibe los valores del nivel que poseen sus células vecinas. Por los puertos de salida (outN, outE, outS, outW) envía, a su vez, el valor de su nivel a sus vecinas. El puerto de salida out sirve para enviar el nivel de cada célula al componente del modelo que permite la visualización de la simulación, y que genera las gráficas y los valores estadísticos.

La conexión de cada célula con sus vecinas se realiza de forma lateral, como se muestra en la Figura 3.3. En ella se pueden apreciar los puertos que participan en cada una de las conexiones.

La simulación comienza con una célula, situada en el centro del espacio celular, que posee un nivel de contaminantes por unidad de área, igual a 1 y el resto de las células con un nivel igual a 0. En cada ciclo de simulación las células actualizan sus valores de la variable nivel.

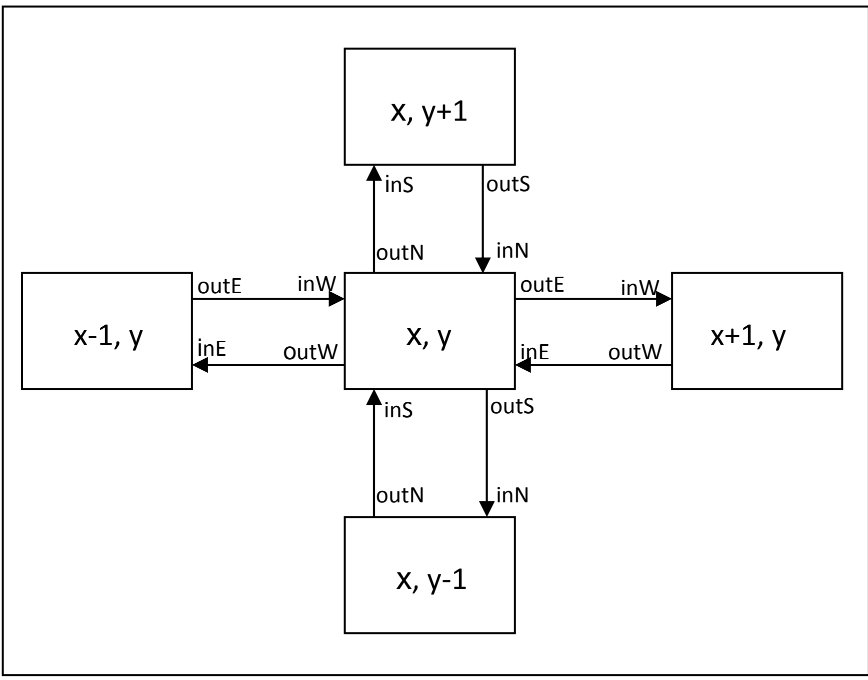


Figura 3.3: Conexiones entre los puertos de una célula con los de cada una de sus vecinas.

En el primer ciclo de la simulación las células vecinas de la célula con nivel 1 recibirán este valor, y al hacer la actualización pasarán a tener un valor distinto de 0. En el siguiente ciclo ya serán más las células con valores de contaminante superiores a 0. De esta manera, por medio de la regla de comportamiento, el nivel de las células irá variando dentro del espacio celular.

La regla para actualizar el valor del nivel de las células en cada paso de la simulación, se obtiene mediante la combinación de las dos reglas de comportamiento del autómata celular, que se han visto en el Apartado anterior. Es la siguiente:

$$N(x,y,t+1) = N(x,y,t) + d[N(x-1,y,t) + N(x+1,y,t) + N(x,y-1,t) + N(x,y+1,t) - 4N(x,y,t)] + v[N(x,y+1,t) - N(x,y,t)]$$

Por tanto, el valor del nivel de la célula *(x, y)* en el instante *t+1*, ***N(x,y,t+1)***, es **igual** al valor que tenía en el instante *t, **N(x,y,t)***, **más** el valor que recibe por *difusión* de sus cuatro células vecinas, menos el valor que envía por *difusión* a esas mismas cuatro células vecinas, ***d[N(x-1,y,t) + N(x+1,y,t) + N(x,y-1,t) + N(x,y+1,t) – 4N(x,y,t)]***, **más** el valor que recibe por *advección* de su vecina superior, menos el valor que envía por *advección* a su vecina inferior, ***v[N(x,y+1,t) – N(x,y,t)]***.

Como para los objetivos de este Proyecto no se precisan coeficientes de *difusión* y *advección* reales, se han elegido valores que muestran el comportamiento correcto del modelo en la simulación durante un tiempo adecuado, y permiten la visualización. Los valores de los coeficientes de *difusión* y *advección* que van a ser utilizados en el modelado son los siguientes:

Coeficiente de *difusión*: *d* = 0.03 unidades de área / unidad de tiempo

Coeficiente de *advección*: *v* = 0.08 unidades de área / unidad de tiempo

En los valores del nivel de contaminantes por unidad de área, se han despreciado los valores inferiores a 10^{-6}, para que no se prolongue demasiado la simulación. Si fuera necesario utilizar alguno de los dos modelos para un estudio riguroso del problema, no habría más que cambiar estos valores por los que se consideraran correctos. También se deberán aplicar las unidades de área y de tiempo adecuadas al estudio concreto que se vaya a realizar.

3.3 Descripción con el Lenguaje de Definición DEVS

Según las reglas de especificación del Lenguaje de Definición de Zeigler, que se han visto en el Apartado 2.4, el modelo celular se puede describir de la forma siguiente:

```
begin description

overall model : modeloCelular

atomic model Celula          //sólo existe un modelo atómico
      inports: inN; inE; inS; inW;
      outports: outN; outE; outs; outW; out;
      state variables:
                  // estados de la célula:
            status in {active, passive};
                  //nivel de contaminantes de la célula:
            nivel in {float};       //cualquier valor entre 0 y 1
                  //posición (x,y) en el espacio celular:
            x in {integer};
            y in {integer};
      initial condition:
                  // todas las células menos la infectada:
            status := passive;
            nivel := 0;
                  /* la célula infectada que ocupa la posición
                  central del espacio celular tendrá los valores
                  iniciales: */
            status := active;
            nivel := 1;
      internal transition:
            {status = active} => {status := passive};
```

```
    external transition:

            /* Si se recibe un evento en uno o más puertos
            de salida se produce un cambio de estado */

        {status = passive} * (inN || inE || inS || inW)

                => {status := active};

    output function:

            /* la salida se realiza simultáneamente en los
            cinco puertos de salida */

        {status = active}

                => outN && outE && outS && outW && out;

end Celula


coupled model EspacioCelular

        /* el espacio celular se compone de n x m células que
        se definen como cel(x,y) para indicar su posición */

    components: Celula cel(0,0); Celula cel(1,0); ...

                Celula cel(x,y); ... Celula cel(n,m);

    internal coupling:

            //para toda cel(x,y) del espacio celular

        cel(x,y).outN -> cel(x,y+1).inS;

        cel(x,y).outE -> cel(x+1,y).inW;

        cel(x,y).outS -> cel(x,y-1).inN;

        cel(x,y).outW -> cel(x-1,y).inE;

        cel(x,y+1).outS ->  cel(x,y).inN;

        cel(x+1,y).outW ->  cel(x,y).inE;

        cel(x,y-1).outN ->  cel(x,y).inS;

        cel(x-1,y).outE ->  cel(x,y).inW;

    external input coupling:

            // Ninguna


    external output coupling:

            // para toda cel(x,y) del espacio celular

        cel(x,y).out -> EspacioCelular.out;

end EspacioCelular
```

3.4 Descripción y Simulación con DEVSJAVA

Los archivos con el código de las clases del modelo se pueden ver en el Anexo A, al final de esta Memoria. También están incluidos en el CD del Proyecto, dentro de la carpeta *Celular*.

Para empezar a realizar el modelado con DEVSJAVA se crea un proyecto nuevo en Eclipse, como se ha explicado en el Apartado 2.2.1. Se le pone el nombre de *Celular*. A continuación, en el proyecto recién creado, se crea un paquete nuevo con el nombre *celular*, que es el que va a contener las clases del modelo.

Como se trata de un espacio celular de dos dimensiones, es conveniente importar la carpeta *twoDCellSpace*, que está en el archivo *IllustratinCode.zip*, el cual se obtiene al bajar el software de DEVSJAVA de la página de descargas. Se crea un nuevo paquete que contenga las clases de dicha carpeta *twoDCellSpace*, porque serán muy útiles para escribir las clases del modelo.

Al estar compuesto el modelo por un modelo atómico que define a la célula, y un modelo acoplado que define al espacio celular, se necesita crear una clase para cada uno de ellos. Estas clases se llaman *CelularCell* el modelo atómico, y *CelularSpace* el modelo acoplado. Una vez creadas estas dos clases, se tiene la vista del entorno Eclipse que muestra la Figura 3.4.

Para la implementación de las clases *CelularCell* y *CelularSpace* se han utilizado como guía de referencia el modelo *ForestFire,* de la carpeta *Examples*,

la cual se encuentra en la página de descargas de DEVSJAVA, y el modelo *Earthquake*, que se obtiene con el enlace *StudentProjects* que figura en esa misma página. Ambos son modelos celulares bidimensionales.

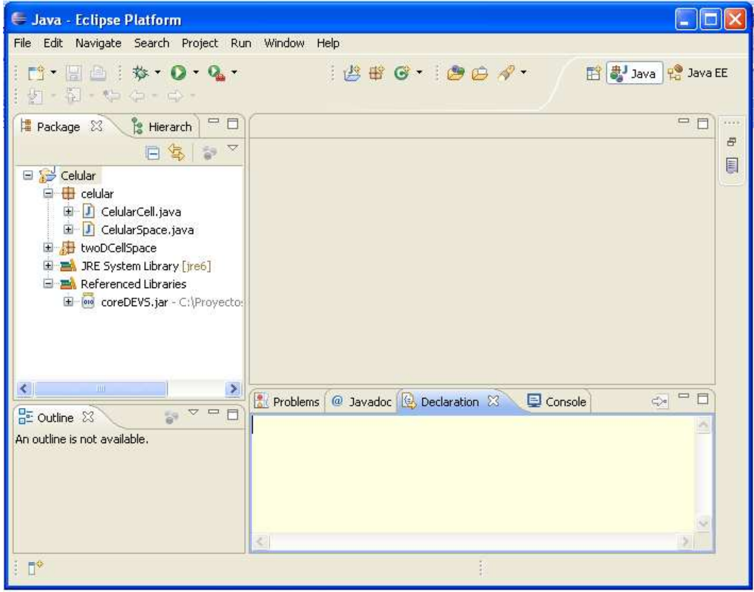


Figura 3.4: Aspecto de Eclipse después de crear el proyecto y las clases necesarias para el modelado.

La clase *CelularCell* hereda de *TwoDimCell*, del paquete *twoDCellSpace*, y contiene los siguientes métodos:

- Método constructor: *public CelularCell(double nivel, int xcoord, int ycoord)*

  Crea una nueva célula con *nivel* como el valor del nivel de contaminante que contiene, y con *xcoord* e *ycoord* como los valores de las coordenadas de la célula en el espacio celular.

- Método de inicialización: *public void initialize()*

  Todas las células son inicializadas en el estado *passive*, si su *nivel* es igual a 0. Si el nivel es distinto de 0 se inicializa en el estado *active*.

- Método de transición externa: *public void deltext(doublé e, message x)*

  Al recibir un mensaje por uno o más puertos de entrada se actualiza el valor del parámetro *nivel*, y la célula pasa al estado *active*.

- Método de transición interna: *public void delint()*

  La célula pasa al estado *passive*.

- Método de salida: *public message out()*

  La célula envía el parámetro *nivel* a sus vecinas y a una instancia de la clase *CellGridPlot*. Esta clase pertenece al paquete *SimView*, que está en la biblioteca de clases de DEVSJAVA. Esta instancia se creará en la clase *CelularSpace*, y será la encargada de visualizar la simulación.

La clase *CelularSpace* hereda de *TwoDimCellSpace*, del paquete *twoDCellSpace*, y contiene los siguientes métodos:

- Método constructor: *public CelularSpace(int xDim, int yDim)*

  Crea un espacio celular de dimensiones *xDim* por *yDim* y lo llena de células de la clase *CelularCell*, todas con un nivel igual a 0, menos la célula central que tiene un nivel igual a 1. Conecta todas las células entre sí mediante la invocación del método *doNeighborToNeighborCoupling()*. Crea una instancia de la clase *CellGridPlot*, del paquete *SimView* que pertenece a la biblioteca de clases de DEVSJAVA. Esta instancia será la encargada de la visualización de la simulación. Conecta el puerto de salida *out* de cada célula, con el puerto *drawCellToScale* de dicha instancia.

- Método principal: *public static void main(String[ ] args)*

  Crea un espacio celular de la clase *CelularSpace* y controla la simulación. Imprime en pantalla los valores de los tiempos de inicio, de final y el tiempo de ejecución.

Los valores de los parámetros *xDim* e *yDim,* necesarios para la creación del espacio celular, se han establecido en 20 en ambos casos. Valores mayores hacen que la simulación se ralentice mucho, e incluso no llegue a terminar, debido a errores de falta de memoria.

El elemento de la clase *CellGridPlot* se construye con una dimensión mayor que la del espacio celular, para poder observar claramente la evolución de la simulación.

Para ejecutar la simulación por primera vez hay que seleccionar la clase *CelularSpace.java*, hacer un click en el botón derecho del ratón, seleccionar *Run As*, y pulsar en *Java Application*. Esto hará que se ejecute el método *main* de la clase y que comience la simulación. Las siguientes ocasiones en que se desee ejecutar la simulación, será suficiente con tener seleccionado el paquete *celular*, o una de sus clases, y pulsar el botón *Run*.

Dentro del método *out* de *CelularCell.java* se ha establecido una clasificación de los mensajes que salen del puerto *out*. Esta clasificación está basada en los rangos de los valores del parámetro *nivel*. Según sea el rango de su valor, la célula envía un color al elemento *CellGridPlot* para que la represente, y de este modo se consigue una mejor visualización del proceso de

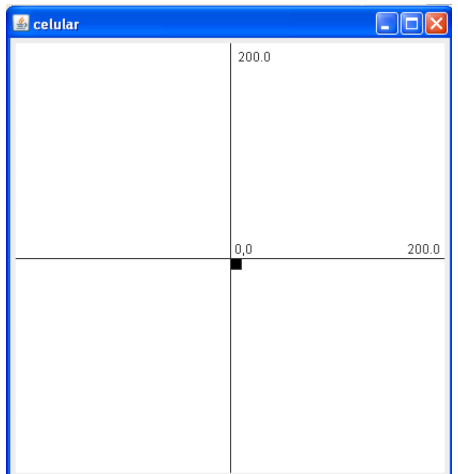
simulación. Si fuera necesario, se puede variar la clasificación modificando el método *out* y estableciendo una nueva. Los valores que se han establecido aquí pueden verse en la Figura 3.5.

nivel	color
> 0.5	Negro
> 0.1	Gris oscuro
> 0.01	Rojo
> 0.0075	Naranja
> 0.005	Amarillo
> 0.0025	Verde
> 0.001	Cyan
> 0.0005	Azul
> 0.00005	Magenta
> 0.000001	Rosa

Figura 3.5: Clasificación de los colores de la visualización según los valores del parámetro nivel.

Una vez arrancada la simulación, aparece una ventana correspondiente a *CellGridPlot*, en donde se sigue la visualización.

Seguidamente se muestran imágenes de algunos momentos de la simulación.



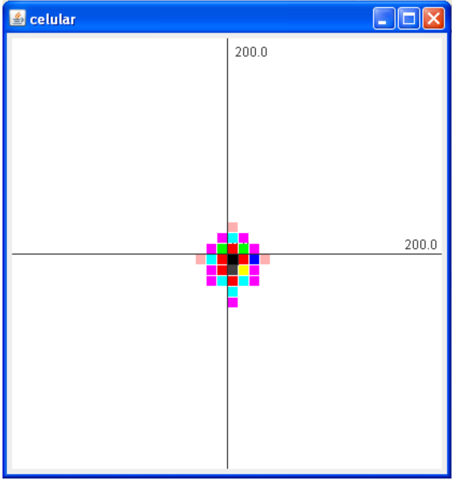


Figura 3.6: Primeros pasos de la simulación.

En la Figura 3.6 se ven los primeros momentos de la simulación. En la imagen de la izquierda se ve el inicio. La célula del centro de color negro, indica un alto valor del nivel de contaminante en ese punto. El resto del espacio es de color blanco por lo que está libre de contaminantes. La imagen de la derecha permite observar cómo el contaminante se empieza a desplazar hacia las células vecinas. El desplazamiento es mayor hacia las células inferiores debido a la *advección*.

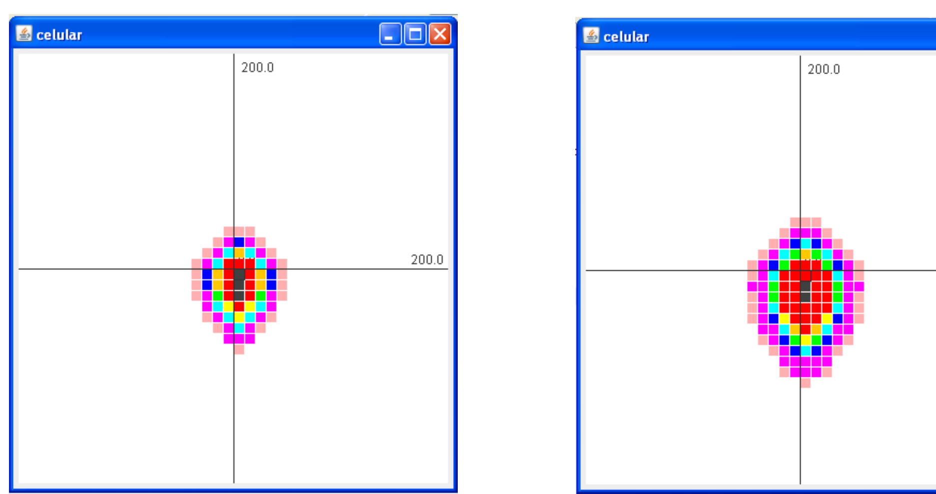
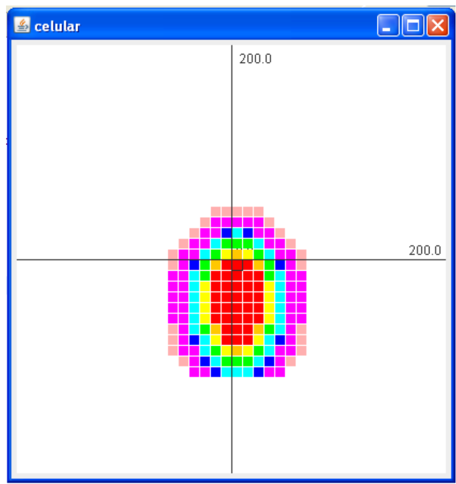


Figura 3.7: Desplazamiento de los contaminantes.

En la Figura 3.7 se observa como la mancha se va extendiendo hacia los cuatro lados, y se desplaza hacia abajo. A medida que se va extendiendo, los valores del nivel de contaminante son más pequeños en las células más alejadas del origen. Y los valores en origen también decrecen.

En la Figura 3.8 podemos ver cómo la mancha formada por los contaminantes se sigue extendiendo. Los niveles siguen decreciendo, y los contaminantes empiezan a abandonar el área contaminada en su desplazamiento.



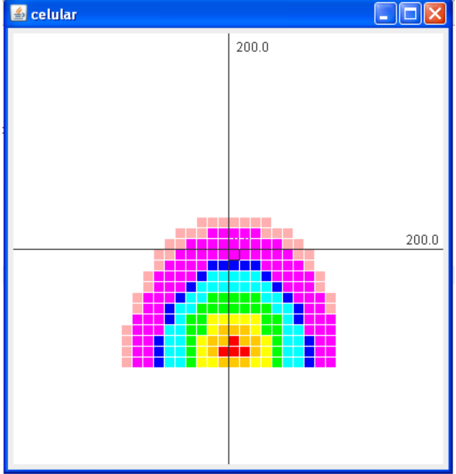
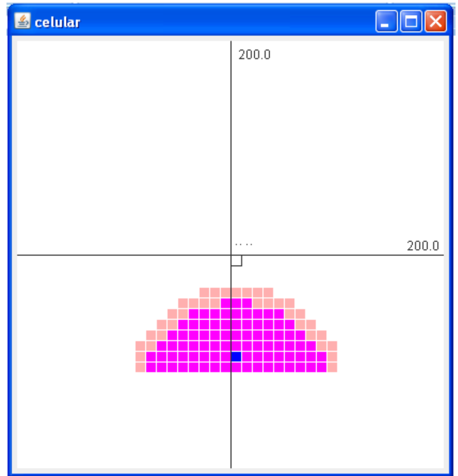


Figura 3.8: Los contaminantes abandonan el espacio celular.

La Figura 3.9 muestra cómo los últimos niveles de contaminante son muy bajos, y el espacio celular va quedando de color blanco por la desaparición de células contaminadas.



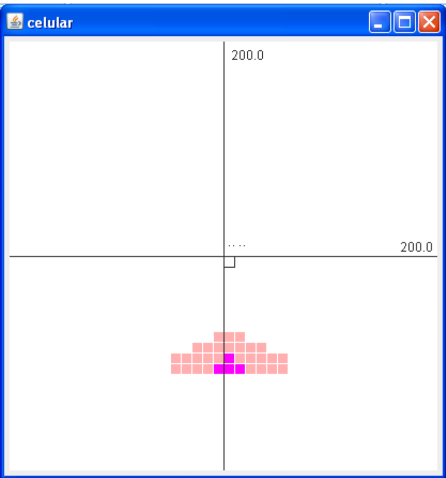


Figura 3.9: Desaparición de los contaminantes.

En la Figura 3.10 podemos ver el espacio celular y el aspecto que muestra Eclipse una vez terminada la simulación. En la consola están impresos los tiempos inicial, final y de ejecución.

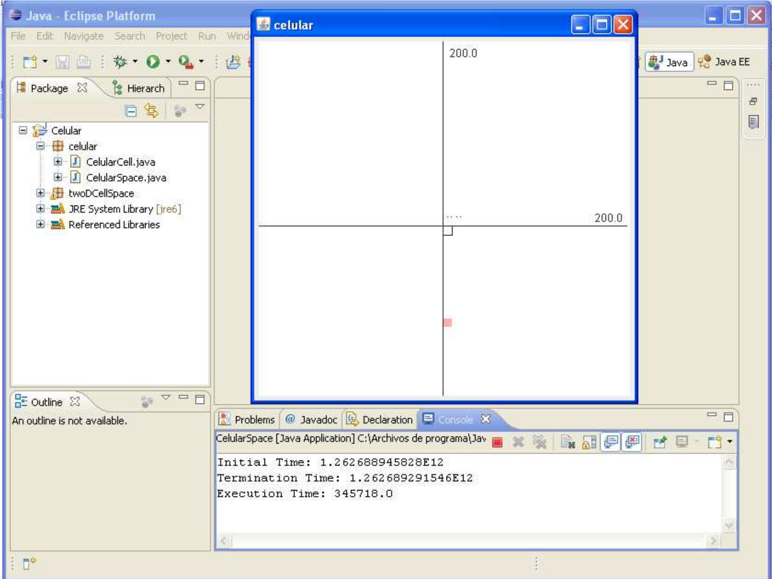


Figura 3.10: Aspecto de Eclipse al finalizar la simulación.

3.5 Descripción y Simulación con CD++

Como ya se comentó en el Apartado 2.3, en CD++ los modelos basados en células, o modelos Cell-DEVS, son un caso especial de modelos acoplados. Se pueden definir mediante un lenguaje de especificación, que es una ampliación del lenguaje de los modelos acoplados, y que incluye los parámetros necesarios para generar un espacio celular completo. El comportamiento de este espacio celular se controla mediante un conjunto de reglas, las cuales se escriben con el lenguaje de especificación. Para generar el modelo se crea un archivo con extensión *.ma*, en el que se describe su composición y su comportamiento.

Por tanto, hay que empezar creando un proyecto nuevo, llamado *celularCD++,* como está explicado en el Apartado 2.3.1. En el proyecto se crea el archivo *celular.ma*, que se escribe por medio del lenguaje de especificación

citado, el cual puede verse en el capítulo 4 del Manual de Usuario de CD++ [Wainer05].

A continuación se reproduce el código del archivo *celular.ma*, y se comentan su estructura y algunos de sus componentes:

```
[top]
components : celular

[celular]
type : cell
width : 40
height : 40
delay : transport
defaultDelayTime : 100
border : nonwrapped
neighbors : celular(-1,1)   celular(0,1)   celular(1,1)
neighbors : celular(-1,0)   celular(0,0)   celular(1,0)
neighbors : celular(-1,-1)  celular(0,-1)  celular(1,-1)
initialvalue : 0
initialrowvalue :  20  0000000000000000000100000000000000000000
localtransition : celular-rule


[celular-rule]

%  Coeficiente de difusión: d = 0.05
%  Coeficiente de advección: v = 0.08

rule :
0 100 {((0,0)>0 or (-1,0)>0 or (1,0)>0 or (0,-1)>0 or (0,1)>0)
      and   ((0,0)+ 0.05*((-1,0)+(1,0)+(0,-1)+(0,1)+(-4)*(0,0))+
            0.08*((-1,0)+(-1)*(0,0))) < 0 }

rule :
{(0,0)+ 0.05*((-1,0)+(1,0)+(0,-1)+(0,1)+(-4)*(0,0))+
      0.08*((-1,0)+(-1)*(0,0)) } 100
{(0,0)>0 or (-1,0)>0 or (1,0)>0 or (0,-1)>0 or (0,1)>0 }

rule : 0 100 { t }
```

La expresión `[top]` es obligatoria al principio de todos los modelos acoplados y define al modelo. A continuación de `components` se detalla la lista de los componentes del modelo. En este caso solamente hay un componente, llamado `celular`.

Después se detallan las características del componente. Primero se escribe su nombre entre corchetes, `[celular]`, seguido de las cláusulas que definen su comportamiento.

La cláusula `type : cell`, define el tipo de simulador abstracto que se va a utilizar.

Las siguientes, `width : 40` y `height : 40`, indican el ancho y el alto, respectivamente, del espacio celular, que en este caso va a ser de 40 por 40.

Siguen tres cláusulas correspondientes al tipo y tamaño del retraso y el tipo de frontera del espacio celular, las dejamos con los valores que aparecen por defecto.

Después se indican las conexiones entre las celdas mediante las cláusulas `neighbors`. Se define que cada célula `celular(0,0)` está conectada a las ocho células que la rodean.

El valor inicial de cada célula se define mediante `initialvalue : 0`, y utilizamos la siguiente, `initialrowvalue`, para indicar el valor 1 de la célula central en nuestro modelo. Para ello se escribe 20 como el número de la fila, seguido por los valores que tendrán las células de esa fila. Estos valores se describen mediante una línea compuesta de una sucesión de ceros, con un uno intercalado en la posición número 20.

La siguiente cláusula `localtransition`, indica el nombre del grupo que contiene las reglas que definen el comportamiento del modelo. En este caso

será **celular-rule**, y lo definiremos a continuación escribiendo **[celular-rule]** seguido de las reglas. Cada una de ellas empieza por **rule**. Las reglas se ejecutan en el orden en que están escritas, y si se encuentra una que se adapta a la situación no se sigue mirando las demás. Las reglas siguen la estructura: VALUE DELAY {CONDITION}, donde VALUE es el valor que pasará a tener la célula si se cumple {CONDITION}, y será retrasada el tiempo que se especifica en DELAY. Para este modelo el valor de DELAY no es determinante, y se ha dejado 100, que es el valor que aparece por defecto en otros modelos observados.

Antes de las reglas hay una línea comentada en la que se recuerdan los valores de los coeficientes de *difusión* y *advección*, que aparecen posteriormente en las reglas. Para comentar una línea en este lenguaje hay que comenzarla con el símbolo %. El lenguaje no permite efectuar la operación resta, sino que hay que hacerlo mediante la suma de números negativos, encerrados entre paréntesis. Cuando VALUE no es simplemente un valor, sino que se trata de una expresión, debe ir encerrada entre llaves.

La primera regla tiene dos condiciones. La primera es que la célula tenga alguna vecina con un valor mayor que cero. La segunda condición es que la aplicación de la regla de comportamiento del modelo genere un resultado negativo. En ese caso se le asigna a la célula el valor cero, ya que no tienen sentido los valores negativos en este modelo.

La segunda regla utiliza la condición de que la célula tenga alguna vecina con un valor mayor que cero. Asigna el valor que obtiene aplicando la regla de comportamiento del modelo.

Mediante la tercera regla se asigna el valor cero a todas las demás células, que no se han visto afectadas por las reglas anteriores.

Para la implementación del archivo *celular.ma* han sido muy útiles los modelos *life2d.zip* y *DiffusionLimitedAggegation.zip*, que se encuentran entre los ejemplos de la página de CD++ en la Universidad de Carleton.

Una vez creado el archivo *celular.ma*, se crea un nuevo archivo con el nombre de *celularLOG.log*. Para efectuar la simulación se pulsa el botón *Simu*, se cargan, en la ventana que aparece, los archivos *celular.ma* y *celularLOG.log*, y se pulsa *Proceed.* Una vez cargados los datos se puede crear un archivo *celular.bat* y guardarlo, para poder utilizarlo todas las veces que se necesite repetir la simulación.

El aspecto de Eclipse durante la simulación se muestra en la Figura 3.11. En ella puede verse la ventana para la simulación, con los datos necesarios cargados con el archivo *celular.bat*.

Cuando termina la simulación y para ejecutar la visualización se pulsa el botón *Animate Cell-Dev Simulation*, el cual abre una ventana de *CD++Modeler* para modelos celulares. Luego se pulsa *Add Model* y se buscan los archivos *celular.ma* y *celularLOG.log*. Se hace doble click sobre

celular@celularLOG.log, que ha aparecido en la columna *Available*. Se pulsa *Load Model* y con ello se carga el modelo.

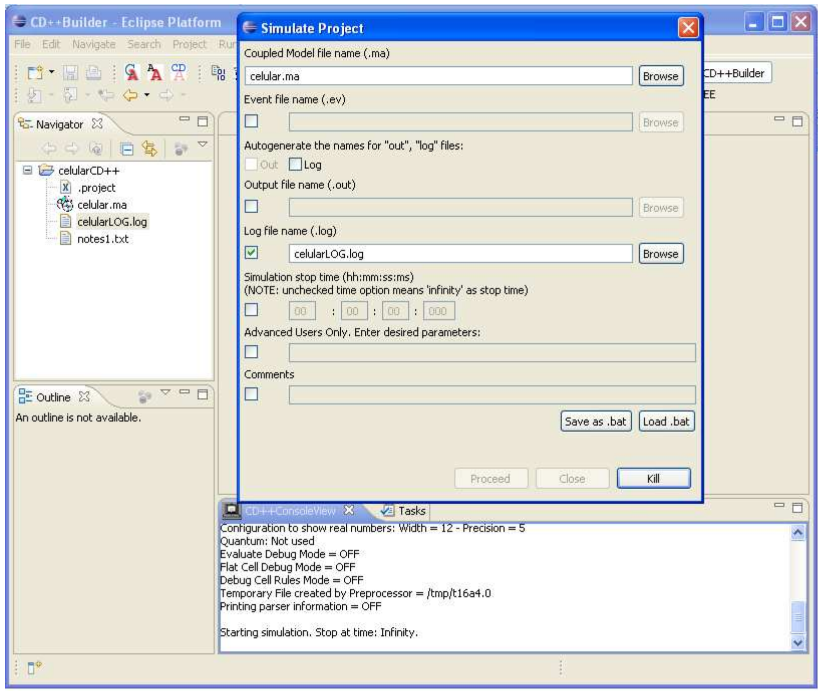


Figura 3.11: Aspecto de Eclipse durante la simulación.

Para que la visualización se lleve a cabo con los colores seleccionados, hay que pulsar *Modify Palette*. En la ventana que aparece, se configura la paleta de colores que se desee y se guarda. Cuando se quiera visualizar la simulación, sólo habrá que cargar la paleta guardada, mediante el botón *Load*. Para este modelo se ha creado la paleta *celular.pal*, en la que se hace una clasificación de colores. Esta clasificación está basada en los valores del parámetro *nivel*. Según sea el valor, la célula será representada de un color determinado, consiguiendo así una mejor visualización del proceso de simulación. Esta clasificación se ha efectuado atendiendo a criterios de mejor visualización para este Proyecto. Si fuera necesario se puede variar,

modificando el archivo *celular.pal* y estableciendo una nueva clasificación. Los valores que se han establecido aquí se muestran en la Figura 3.12.

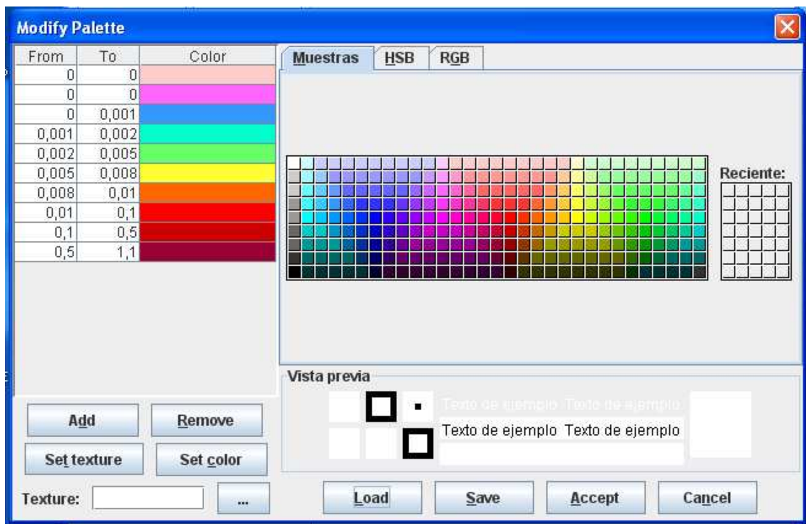


Figura 3.12: Paleta de colores para la visualización.

Al pulsar *Play* comenzará la simulación. Si se quiere, se puede deseleccionar la casilla de *Show Values*, para ver mejor la simulación.

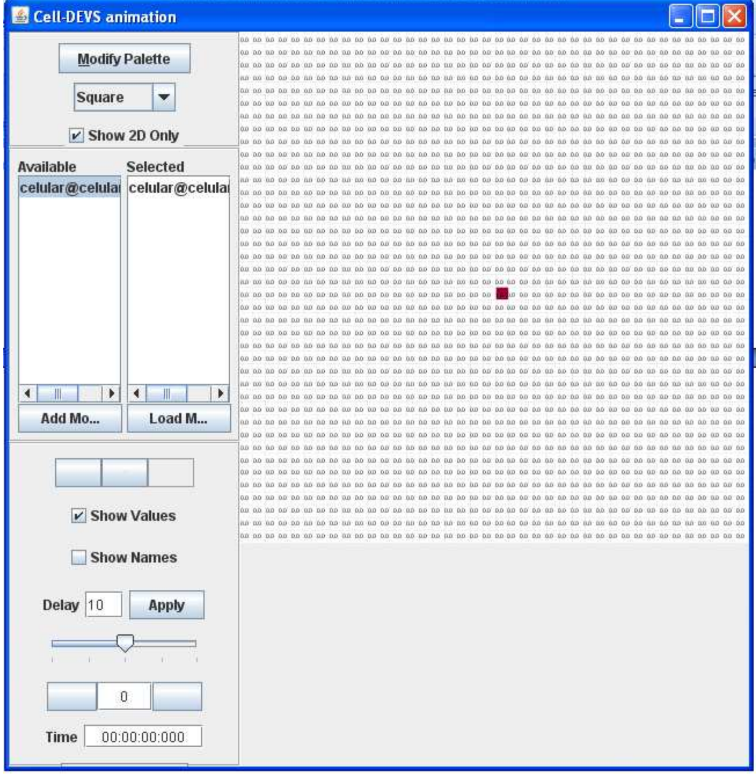


Figura 3.13: La aplicación *Cell-DEVS animation* antes del comienzo de la visualización.

En la Figura 3.13 se muestra el aspecto de la aplicación *Cell-DEVS animation*, con el modelo cargado, antes de comenzar la visualización. Los botones para ejecutar o parar la simulación son *Play, Stop* y *Pause*. Están situados en ese orden, de izquierda a derecha, encima de la casilla *Show Values*. Al final de esa columna, se puede ver el tiempo de duración, y encima de él, el número de ciclos.

A continuación se muestran unas imágenes del proceso de la visualización, en las que se observa el comportamiento del modelo.

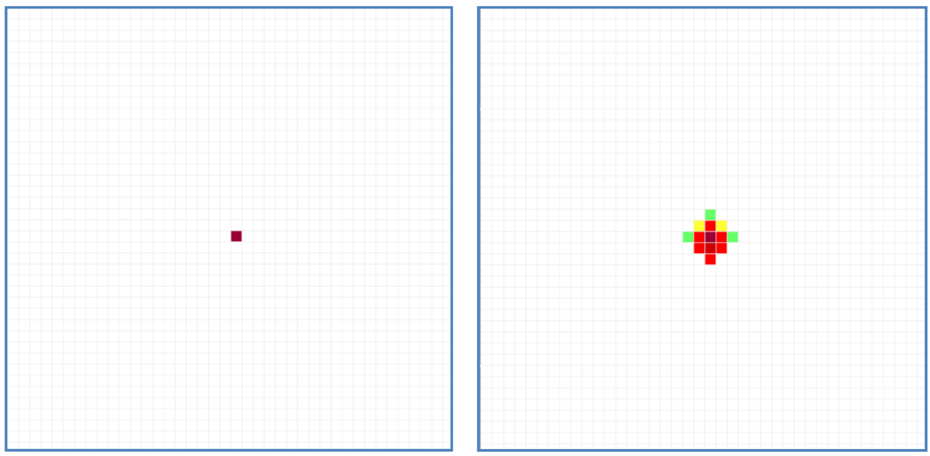


Figura 3.14: Primeros pasos de la simulación.

En la Figura 3.14, se pueden ver los primeros pasos de la simulación. En la izquierda se ve a la célula infectada, situada en el centro del espacio celular. El resto de las células aún no han sido afectadas.

En la Figura 3.15, se observa cómo la mancha va aumentando su tamaño y se desplaza hacia abajo, por efecto de la *advección*. Las células más alejadas del punto central tienen valores más pequeños que las que están en el centro.

La contaminación se va extendiendo, debido al efecto de equilibrio que produce la *difusión*.

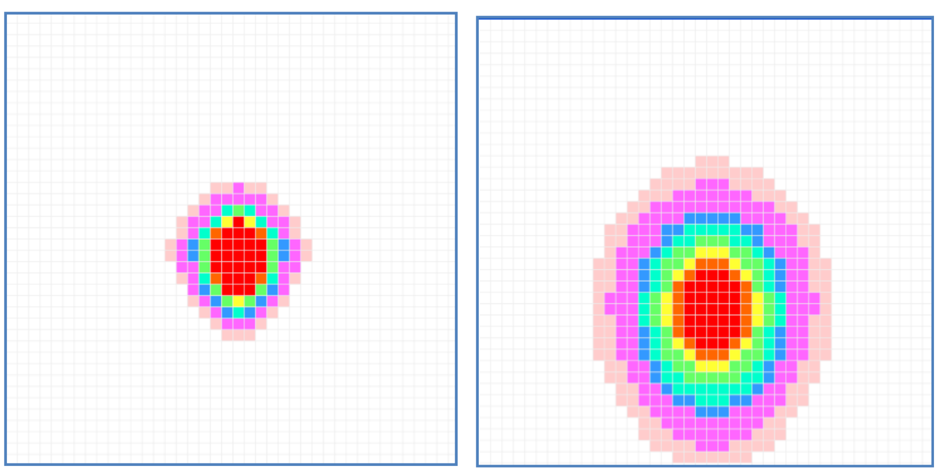


Figura 3.15: Desplazamiento de los contaminantes.

En la Figura 3.16 se observa como la mancha sigue aumentando de tamaño, y al mismo tiempo los niveles máximos son cada vez más pequeños. En su desplazamiento, los contaminantes empiezan a abandonar el área contaminada.

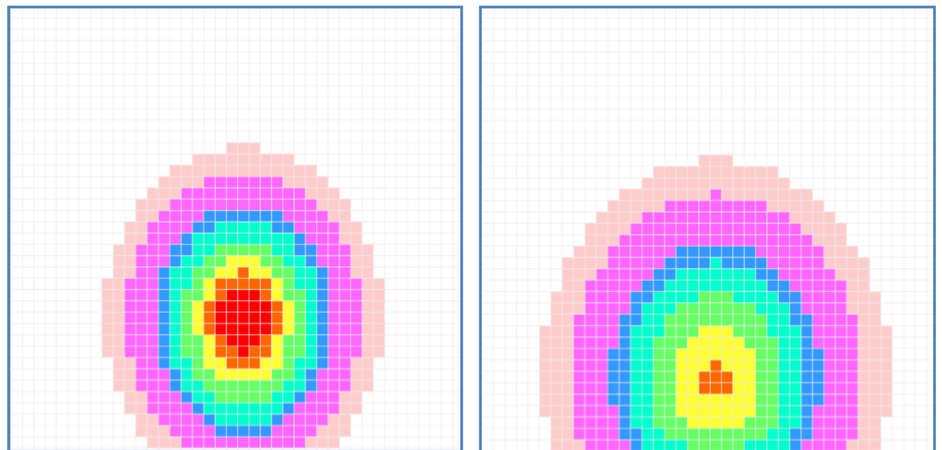


Figura 3.16: Los contaminantes abandonan el espacio celular.

La Figura 3.17 muestra cómo las últimas células contaminadas poseen valores muy bajos de contaminante, y el espacio celular va quedando limpio, por el desplazamiento de las células contaminadas.

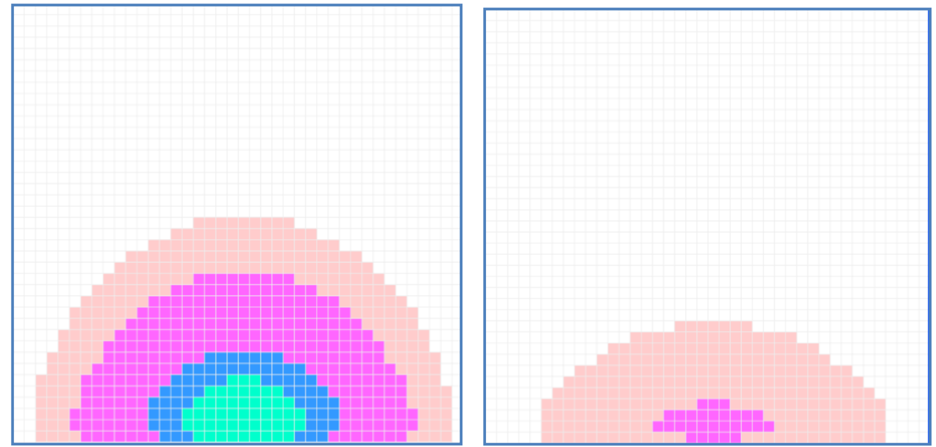


Figura 3.17: Desaparición de los contaminantes.

La Figura 3.18 muestra el aspecto de la aplicación *Cell-DEVS animation* al final de la simulación. En ella se pueden ver el espacio celular totalmente vacío, y los valores del número de ciclos y el tiempo de ejecución.

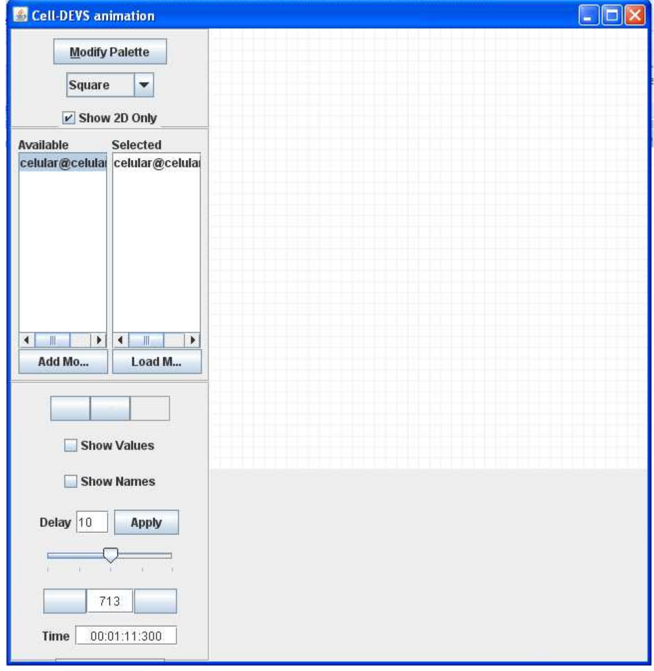


Figura 3.18: Aspecto de *Cell-DEVS animation* al final de la simulación.

Los archivos con el código del modelo realizado con CD++ están en el CD que acompaña a esta Memoria del Proyecto, dentro de la carpeta *cellularCD++.*

3.6 Conclusiones

Se ha visto el desarrollo de las herramientas DEVS en el modelado mediante un autómata celular, en un espacio bidimensional.

El modelo está compuesto de un modelo atómico, que representa a cada una de las células, y un modelo acoplado, representando al espacio celular.

La descripción mediante el Lenguaje de Definición DEVS contiene los dos modelos y detalla su estructura.

El modelado con las dos herramientas DEVS es sencillo en ambos casos. Pero con CD++ es casi inmediato, debido a que no hay que escribir código. La herramienta de visualización de CD++ también ofrece más rendimiento y posibilidades gráficas.

El resultado obtenido del modelado y la simulación con ambas herramientas es muy similar.

4

Modelado de un Sistema Logístico

4.1 Introducción

El sistema logístico que se va a modelar, es el que simula el movimiento de los usuarios de una gasolinera. El comportamiento de un usuario será el siguiente:

1. Accede a la instalación y se sitúa en el surtidor que está libre o, si no hay ninguno libre, en el que tenga menos vehículos esperando en la cola.
2. Rellena el depósito con la cantidad de combustible que desea y, sin mover el vehículo, se dirige a la zona de cajas para abonar el importe.
3. Si las cajas están ocupadas, espera en la cola hasta que llegue su turno. Realiza el pago, regresa a su coche, y abandona la gasolinera.

En este modelo se van a poder observar los procesos con cola, y el paso de mensajes entre ellos. También será necesaria la utilización de funciones estadísticas.

4.2 Definición del Modelo

El modelo tendrá una estructura formada por los siguientes módulos:

- *Marco experimental*: Genera las entidades que entran en el sistema y permite hacer un seguimiento de la simulación.
- *Entrada*: Recibe las entidades que genera el *marco experimental*, selecciona el *surtidor* libre, o con menor ocupación, y envía hacia allí la entidad recién llegada.
- *Surtidor*: Es un proceso con una cola. Las entidades llegan a él desde la *entrada*. No se libera hasta que la entidad abandona la *caja* de pago. El modelo consta de tres *surtidores.*
- *Cajas*: Estará compuesto por las *cajas* y una *cola* común. Las entidades llegan desde el *surtidor*. El modelo cuenta con dos *cajas*.

Además, el modelo dispondrá de elementos para generar datos estadísticos sobre los tiempos de espera en las colas, los tiempos que duran los procesos, el número de vehículos que han sido atendidos, etc.

El esquema del modelo se reproduce en la Figura 4.1. En ella se pueden ver los módulos y sus conexiones. Los nombres de los módulos en la Figura son

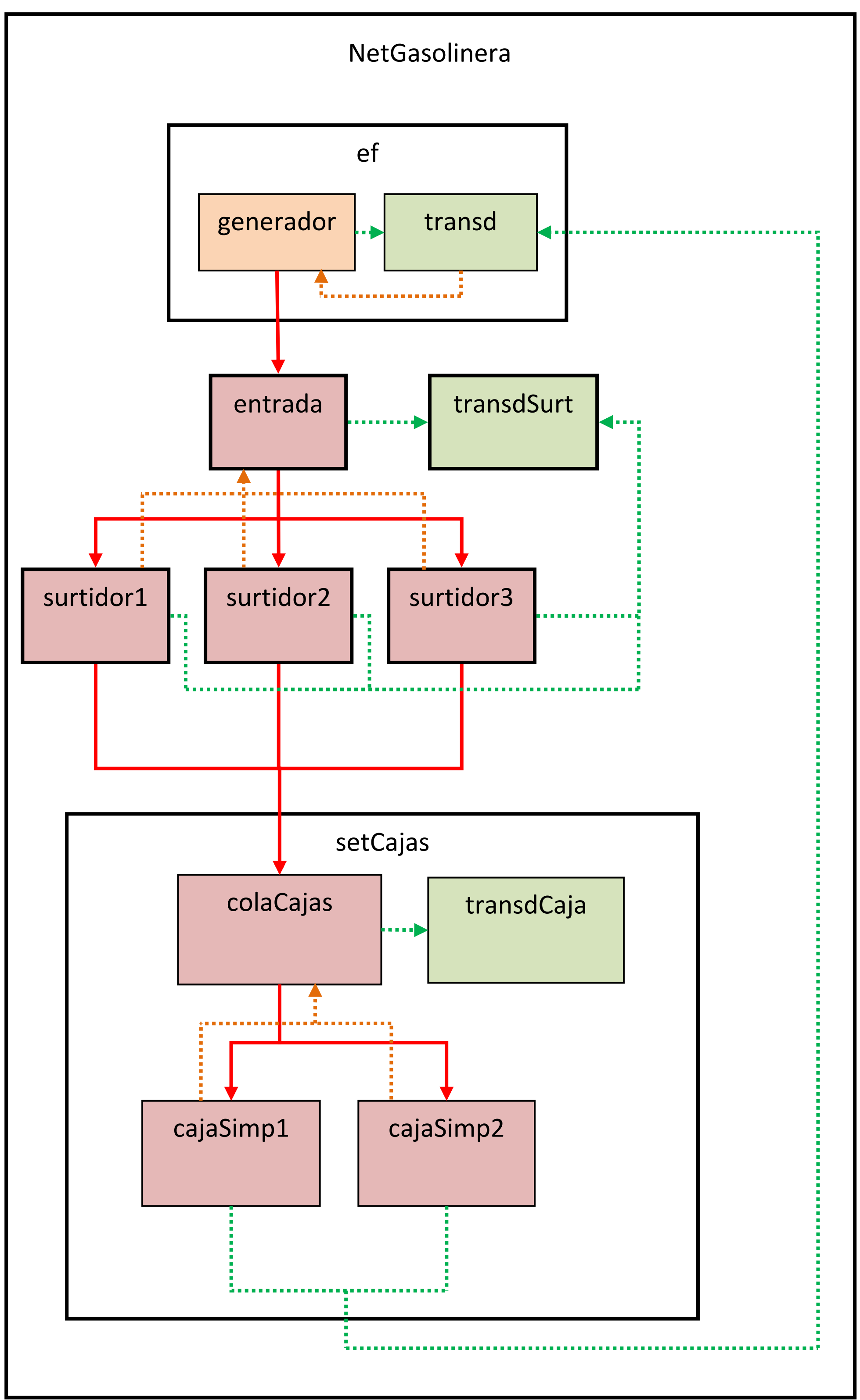


Figura 4.1: Estructura del modelo gasolinera.

los que se van a utilizar en el resto de la descripción del modelo. Con flechas continuas de color rojo, se representa el recorrido de las entidades por el sistema, desde que son creadas por el *generador*, hasta que terminan el proceso de las *cajas*. Las flechas de puntos de color verde indican el movimiento de la información, desde los módulos, hacia los *transductores* encargados de generar la información: *transd, transdSurt* y *transdCaja*. La información necesaria para la simulación, que se transmite entre *transd* y *generador*, entre los *surtidores* y la *entrada*, y entre las *cajas* y la *colaCajas*, se representa mediante flechas discontinuas de color naranja.

Con la intención de hacer un modelo más variado, los modelos atómicos *entrada*, *transdSurt* y los *surtidores*, no se han englobado en un modelo acoplado. A continuación se describen el comportamiento y las características de cada uno de los módulos y sus componentes.

- *NetGasolinera*: Es el modelo acoplado principal que acoge a todos los demás modelos que forman el sistema.
- *ef* (marco experimental): Se trata de un modelo acoplado que está formado por los dos modelos atómicos *generador* y *transd*.
    - *generador*: Es el elemento encargado de crear las entidades e introducirlas en el sistema. Al crear las entidades, les asigna un valor que las identifica y las diferencia de las demás, durante toda la simulación. Los intervalos de tiempo entre creación de las entidades, siguen una función de distribución de probabilidad *exponencial*, de

parámetro 5. Esta función se adapta muy bien al caso que estamos estudiando, y simula la llegada de vehículos a la gasolinera con una media de cinco minutos entre ellos. Este modelo atómico dispone de un único puerto de salida, llamado *out*, conectado con el puerto de entrada *ariv*, de *transd,* por el que le comunica el valor de las entidades que genera, y con el puerto de salida *out*, de *ef*, por donde salen las entidades hacia el módulo *entrada*. Tiene sólo un puerto de entrada, *stop*, conectado con el puerto *out*, de *transd*, y que sirve para que finalice la generación, cuando recibe un mensaje en él.

- *transd*: Tiene como misión realizar un seguimiento de la simulación, e ir imprimiendo los tiempos de llegada y salida de las entidades, y el número de entidades que han llegado y salido. Al término de la simulación, imprime los resultados finales. Controla el tiempo de la simulación y, por medio de un mensaje, le indica a *generador* cuando debe finalizar su tarea de crear entidades. Tiene dos puertos de entrada: *ariv* y *solved*. A través de *ariv*, recibe de *generador* el valor de las entidades que entran al sistema, y en *solved*, recibe de *setCajas* el valor de las entidades que han terminado de ser procesadas y abandonan el sistema. Registra los tiempos de llegada de los mensajes, para poder hacer el seguimiento de la simulación. Por su único puerto de salida, *out,* es por donde envía el mensaje a *generador* de que debe finalizar la creación de entidades. El tiempo

de duración de la simulación es un parámetro de este elemento. En este modelo se fija en 1440 minutos, que son 24 horas.

- *entrada*: Es un modelo atómico de un proceso, que recibe las entidades que provienen de *ef*. Es la entrada al sistema, y su cometido es distribuir equitativamente las entidades entre los *surtidores*. Para ello, debe tener información en todo momento del estado de ocupación de todos los *surtidores*. Tiene un puerto de entrada, *in*, por donde recibe las entidades y otros tres puertos de entrada, *inQ1, inQ2* e *inQ3*, por los que le llega la información que le envía cada *surtidor*, con el número de entidades que están en él. Cada vez que llega o sale una entidad de un *surtidor*, éste envía un mensaje a *entrada* actualizando la información. Por sus tres puertos de salida, *out1, out2* y *out3*, envía las entidades a cada uno de los *surtidores*, y también los valores que las identifican a *transdSurt*. Las entidades no se detienen en este modelo, llegan e inmediatamente son enviadas al *surtidor* adecuado.

- *surtidor*: Este modelo atómico está construido como un proceso con una cola. La cola es de tipo FIFO (*First In First Out*). Es uno de los elementos más complejos del modelo. Debe enviar información a *entrada* de las variaciones que haya en la cola, y a *transdSurt* de las entidades que comienzan el proceso, para que pueda reflejar los tiempos de permanencia en la cola. Al finalizar el tiempo de procesado, retiene a la entidad hasta que recibe el mensaje con la información de que dicha entidad ha terminado su proceso en *setCajas*. Entonces inicia el proceso de la siguiente

entidad en la cola. Tiene dos puertos de entrada, *in* e *inEnd*. En el puerto *in* recibe las entidades que le envía *entrada*. Al puerto *inEnd* llega el mensaje de que, la entidad que tiene retenida, ha terminado el proceso en *setCajas*. Los puertos de salida son tres: *out*, *outQ* y *transd*. Por el puerto *out* salen las entidades hacia *setCajas*. El puerto *outQ* envía a *entrada* el número de entidades que contiene, cada vez que se produce una variación. El puerto *transd* le envía a *transdSurt* la identificación de las entidades, cuando abandonan la cola y empiezan a ser procesadas. Para poder efectuar todo ese intercambio de mensajes, el modelo *surtidor* necesita realizar transiciones a varios estados diferentes. Además del estado *passive*, en el que permanece mientras no haya entidades ni en la cola, ni en proceso, ni esperando a que termine el proceso de *setCajas*, existen otros estados: el estado *busy*, en el que permanece durante el tiempo que dura el proceso; el estado *wait*, donde espera, después de haber procesado una entidad, a que se reciba el mensaje que le comunica que ha finalizado el proceso en *setCajas*; el estado *end*, con tiempo de permanencia cero, al que pasa después de *wait*, cuando recibe el mensaje de *setCajas*, y en el que comprueba si hay entidades esperando en la cola; el estado *sendOutQ*, con tiempo de permanencia cero, en el cual envía a *transd* el mensaje de que inicia el proceso de una entidad, si ésta ha llegado cuando el *surtidor* estaba en el estado *passive*, y por tanto la cola estaba vacía; el estado *sendOutQ2*, igual que el anterior, pero para el caso de que la entidad estuviera esperando en la cola; el estado *sendQ*, de tiempo cero, en el que envía a

entrada el tamaño de la cola, si una entidad llega mientras *surtidor* está en el estado *busy*; el estado *sendQ2*, igual que el anterior, pero para el caso en el que la llegada de la entidad se produce mientras el *surtidor* permanece en el estado *wait*. El diagrama del recorrido de una entidad y los cambios de estado que se producen, puede verse en la Figura 4.2. En este diagrama de transición de estados se representan los estados por rectángulos, y los demás modelos con los que se relaciona *surtidor*, están representados por elipses. Las flechas discontinuas indican los mensajes que son enviados entre los modelos, y que salen o entran al modelo *surtidor*. El tiempo de proceso sigue una distribución de probabilidad *uniforme*, con parámetros 2 y 5. De esta manera se especifica un tiempo mínimo de dos minutos y uno máximo de cinco. En este tiempo no se incluye el de permanencia en la cola, pero sí el que necesitará invertir el usuario para ir y volver caminando a las cajas de pago, no el tiempo que se invierta en las *cajas*, que se incluirá allí. El tiempo mínimo sería el de alguien que utiliza habitualmente el surtidor y no llena el depósito, y el tiempo máximo reflejaría el que invertiría alguien menos experto, que llenara el depósito de su vehículo.

- <u>*transdSurt*</u>: Es el modelo atómico responsable de generar la información estadística sobre las colas de los *surtidores*. Tiene cuatro puertos de entrada: *ariv*, *solved1*, *solved2* y *solved3*. En el puerto *ariv* recibe, desde *entrada,* los valores de las entidades que ésta envía hacia los *surtidores*. En los puertos *solved1*, *solved2* y *solved3*, recibe los valores de las entidades

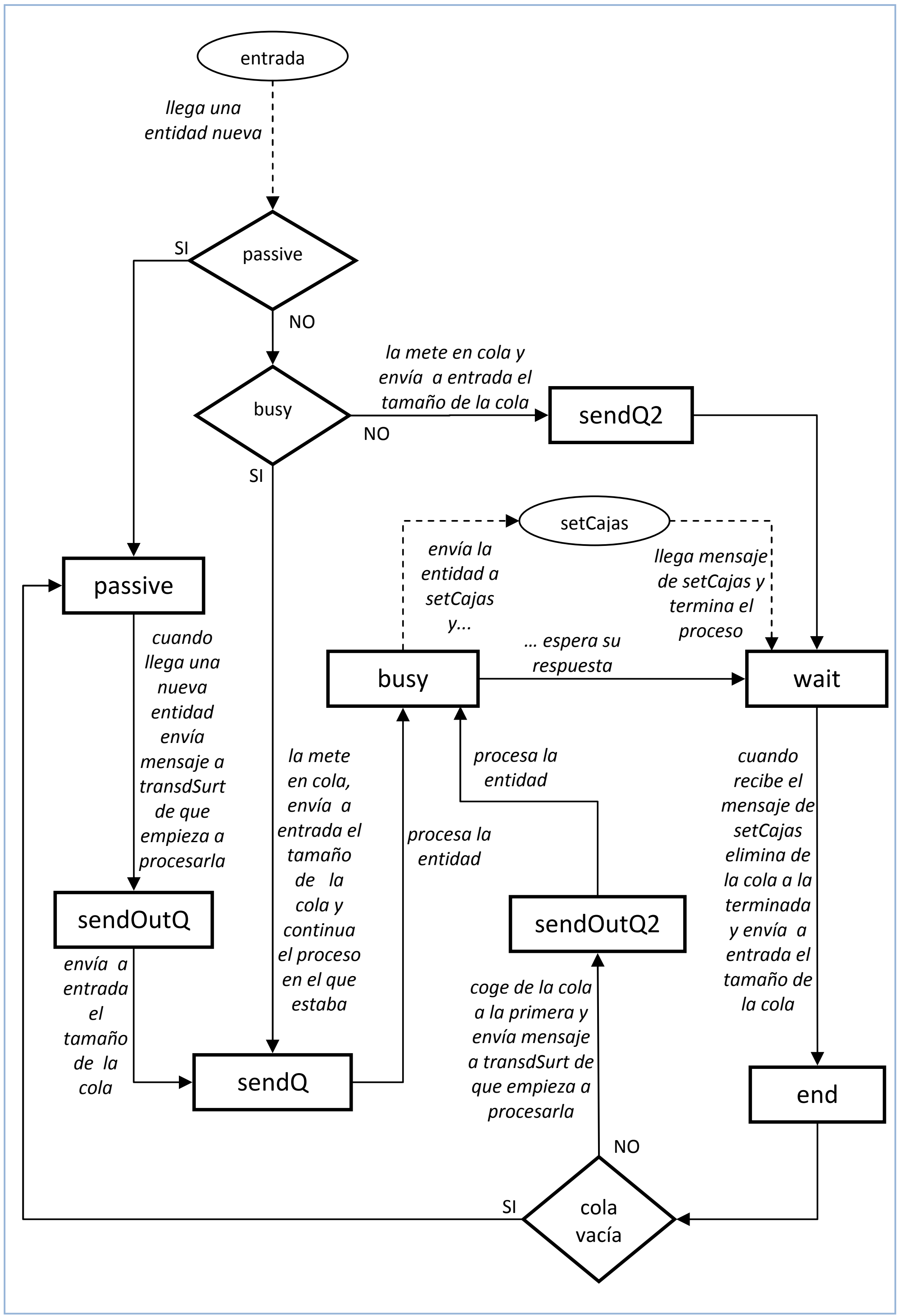


Figura 4.2: Diagrama de transición de estados del modelo atómico *surtidor*.

que abandonan la cola de *surtidor1*, *surtidor2* y *surtidor3*, respectivamente. No tiene puertos de salida. Al final de la simulación, imprime los datos estadísticos de las colas de los tres *surtidores*.

- <u>*setCajas*</u>: Es un modelo acoplado, que contiene los modelos atómicos: *colaCajas*, *cajaSimp1*, *cajaSimp2* y *transdCaja*.
    - <u>*colaCajas*</u>: Es un modelo atómico compuesto por un proceso con una cola. La cola es de tipo FIFO (*First In First Out*). Las entidades llegan desde los *surtidores*, y esperan en la cola hasta que una de las *cajas* queda libre. Junto con *surtidor*, es otro de los modelos complejos del sistema. Debe conocer el estado de las *cajas* en todo momento, para enviar una entidad de la cola a la *caja* que esté libre. Envía información a *transdCaja* de las entidades que envía hacia las cajas, para que pueda reflejar los tiempos de permanencia en la cola. Tiene tres puertos de entrada: *in*, *estado1* y *estado2*. En el puerto *in* recibe las entidades que llegan desde los *surtidores*. A los puertos *estado1* y *estado2* llegan los mensajes que le envían *cajaSimp1* y *cajaSimp2*, respectivamente. En estos mensajes le indican su estado de ocupación, es decir, si están procesando una entidad, o si están libres. Tiene tres puertos de salida: *out1, out2* y *outQ*. Desde *out1* envía las entidades hacia *cajaSimp1*, y desde out2 hacia *cajaSimp2*. Por *outQ*, envía el número total de personas que hay en *setCajas*. Para llevar a cabo el intercambio de los mensajes, es necesario que el modelo realice transiciones a varios estados

diferentes. Además del estado *passive,* en el que la *caja* permanece inactiva, existen otros estados: el estado *busy*, de tiempo de proceso cero, en el que selecciona una *caja* libre, y envía hacia ella y hacia *transdCajas*, a la primera entidad de la cola; el estado *wait*, donde permanece, si hay alguna entidad en la cola y las *cajas* están ocupadas, hasta que reciba un mensaje de alguna de las *cajas* indicándole que ha quedado libre; el estado *sendQ*, de tiempo cero, en el que envía el número total de personas que hay en *setCajas*; el estado *sendQ2*, igual que el anterior, pero que se activa si el mensaje de que ha quedado una *caja* libre se recibe durante el estado *passive*. El diagrama de transición de estados puede verse en la Figura 4.3, en la que los estados están representados por rectángulos y los demás modelos con los que se relaciona *colaCajas* están representados por elipses. Las flechas discontinuas indican los mensajes que son enviados entre los modelos y que salen o entran al modelo *colaCajas*.

- *cajaSimp*: Es un modelo atómico de un proceso. Recibe las entidades que le envía *colaCajas*, las procesa, y al terminar, las envía fuera del sistema. También envía a *transd* el valor de cada entidad que termina de procesar, y al *surtidor* de donde procedía la entidad, el mensaje de que ya puede liberarse y procesar una nueva entidad. También le comunica a *colaCajas* que ha quedado libre. Para ello cuenta con un puerto de entrada, *in*, al que llegan las entidades, y

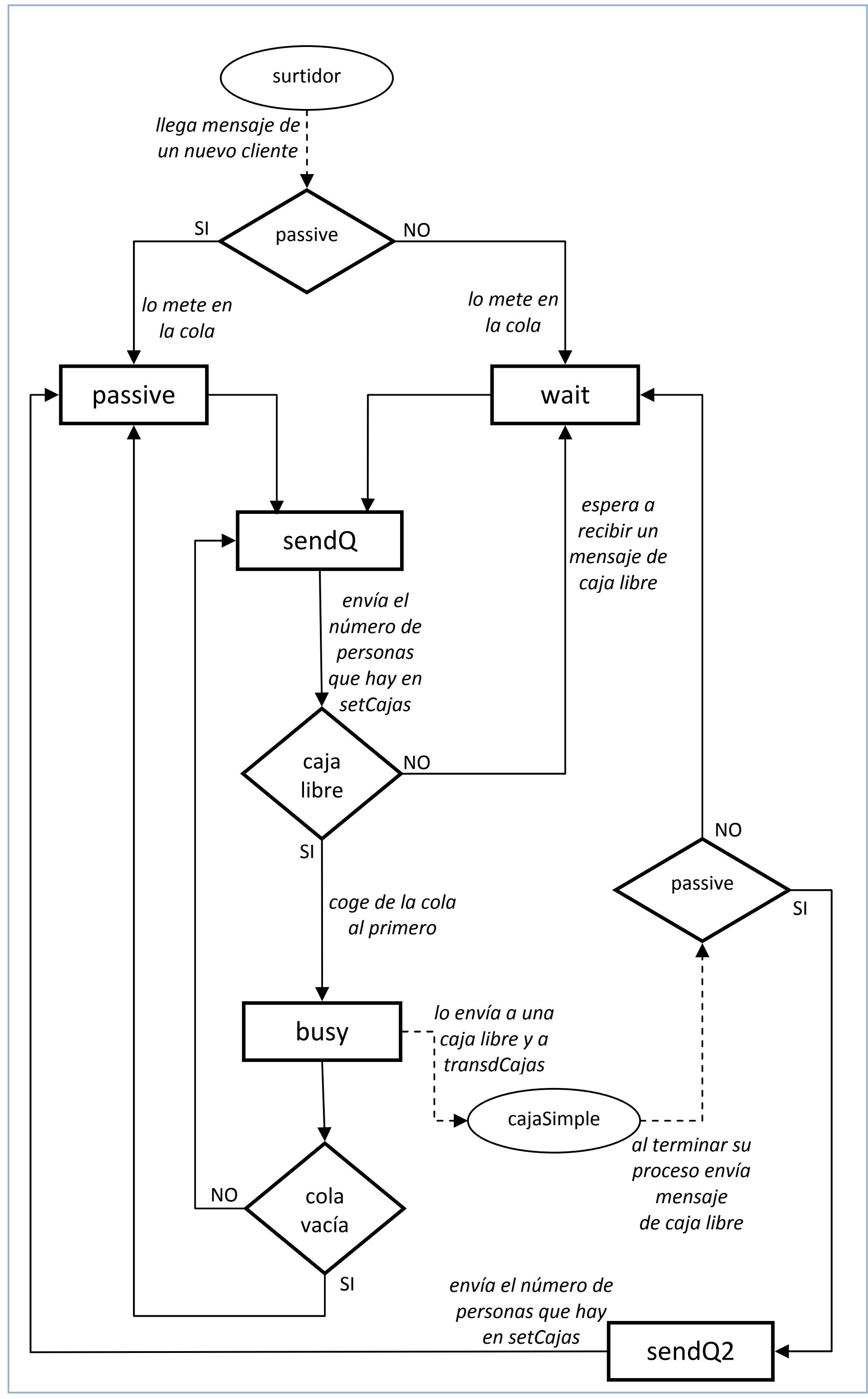


Figura 4.3: Diagrama de transición de estados del modelo atómico *colaCajas*.

cinco puertos de salida: *out*, *estado*, *out1*, *out2* y *out3*. Por el puerto *out*, envía a la entidad fuera del sistema, y al mismo tiempo, le comunica a *transd* el valor de la entidad. El puerto *estado* es por donde le envía un mensaje a *colaCajas*. Los puertos *out1*, *out2* y *out3*, se comunican con *surtidor1*, *surtidor2* y *surtidor3*, respectivamente. El tiempo que dura el proceso sigue una función de distribución de probabilidad *uniforme*, de parámetros 0,5 y 2. El valor mínimo correspondería a una persona que paga en metálico, y el máximo a alguien que paga con tarjeta, pide recibo y tiene un comportamiento lento. El diagrama de transición de estados se puede observar en la Figura 4.4, en donde los estados están representados por rectángulos, y los demás modelos están representados por elipses. Las flechas discontinuas indican los mensajes que son enviados entre los modelos, y que salen o entran al modelo *cajaSimp*.

- *transdCaja*: Es el modelo atómico encargado de la generación de la información estadística sobre la cola de *colaCajas*. Tiene dos puertos de entrada *ariv* y *solved*. En el puerto *ariv* recibe los valores de las entidades que llegan a *colaCajas* y en *solved* el de las entidades que salen de *colaCajas*. No tiene puertos de salida. Imprime la información al finalizar la simulación.

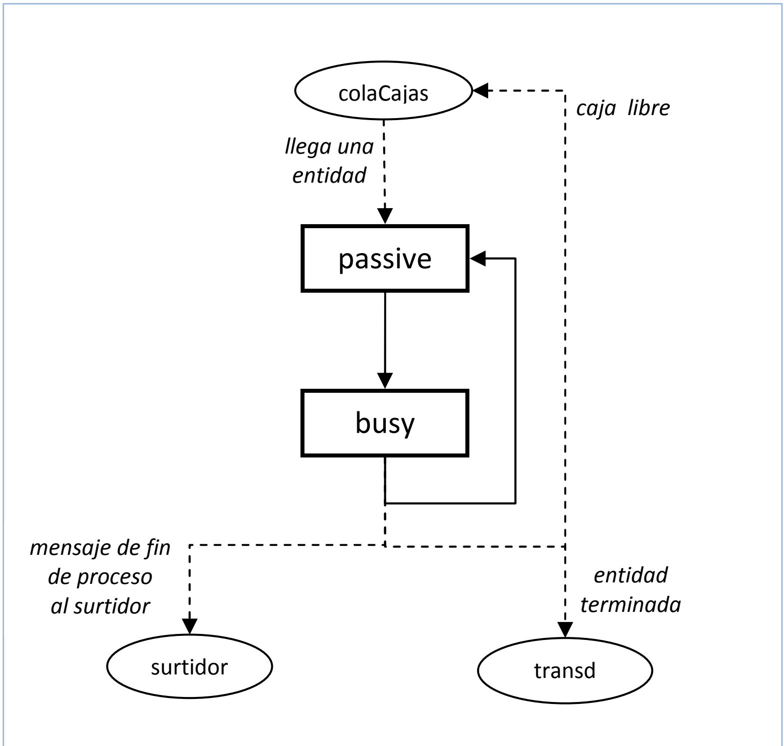


Figura 4.4: Diagrama de transición de estados del modelo atómico *cajaSimp*.

Como se ha visto, se trata de realizar el modelado de una gasolinera compuesta por tres surtidores, y dos cajas con una cola común. La simulación durará 24 horas, y los tiempos de llegada de vehículos siguen una función *exponencial (5)*. Los surtidores tienen un tiempo de proceso según una función *uniforme (2, 5)*, y el de las cajas es *uniforme (0.5, 2)*. Todos los valores son en minutos.

El modelo podría tener más surtidores o cajas, pero se ha considerado que los que hay son una cantidad suficiente, y permiten mayor claridad que con un número más elevado. De cualquier manera, al ser modulable, se podrían añadir más elementos al modelo sin mucha dificultad, en caso de ser necesario.

El generador de entidades está incluido dentro del sistema por una cuestión de simplicidad, pero también podría funcionar con un generador externo conectado directamente al puerto de entrada *in*, del modelo acoplado *NetGasolinera.*

4.2.1 Definición formal DEVS

A continuación se definen todos los modelos atómicos, según el formalismo DEVS paralelo, que se mostró en el Apartado 1.1. En este modelo hay componentes que tienen más de un puerto de entrada o de salida, y además, hay algún modelo en el que se pueden recibir varios mensajes simultáneamente.

- generador:

$DEVS_{\Delta} = < X_M, S, Y_M, \delta_{ext}, \delta_{int}, \delta_{cont}, \lambda, ta >$

Δ : Parámetro del modelo. Tiempo entre salidas. Se calcula cada vez, mediante la función de probabilidad *exponencial* con parámetro igual a 5.

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$

InPorts = { "*stop*" }

X_{stop}: R — R representa al conjunto de los números reales. (En realidad, el puerto *stop* acepta cualquier tipo de mensaje)

$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$

OutPorts = {"*out*"}

Y_{out}: R^+ — R^+ representa al conjunto de los números reales positivos.

S = (*fase, σ, contador*) = {*"activo", "pasivo"*} x R^+_0 x R^+

fase puede ser "activo" o "pasivo".

σ: R^{+}_{0}, valores reales positivos incluyendo el cero.

contador: R^{+}, valores reales positivos. Cada mensaje que envía, contiene el valor de la variable *contador.*

δ_{ext} (*"activo", σ, contador, e*, (*"stop"*, x)) = (*"pasivo"*, ∞, *contador*)

δ_{int} (*"activo", σ, contador*) = (*"activo"*, Δ, *contador+1*)

δ_{cont} (*s*, x) = δ_{ext} (δ_{int} (*s*), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno, y a continuación el externo.

λ (*"activo", σ, contador*) = ("*out*", *contador)*

ta (*fase, σ, contador*) = *σ*

- <u>*transd*</u>:

$DEVS_{\Delta} = < X_M, S, Y_M, \delta_{ext}, \delta_{int}, \delta_{cont}, \lambda, ta >$

Δ : Parámetro del modelo. Tiempo que permanece activo.

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$

InPorts = { "*ariv*", "*solved*"}

X_{ariv}: X_{solved}: R^{+}

$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$

OutPorts = {"*out*"}

Y_{out}: {1}

S = (*fase, σ,* L_a, L_s) = {*"activo", "pasivo"*} x R^{+}_{0} x V^{+} x V^{+}

L_a y L_s almacenan una secuencia finita de valores en forma de lista, pertenecientes a R^{+}. Esta secuencia se representa con V^{+}.

δ_{ext} (*"activo", σ,* L_a, L_s, *e*, (*"ariv"*, x)) = (*"activo", σ – e,* L_a + x , L_s)

δ_{ext} (*"activo", σ,* L_a, L_s, *e*, (*"solved"*, x)) = (*"activo", σ – e,* L_a, L_s + x)

L_a + x indica que la entidad x se añade a la lista L_a

δ_{int} (*"activo", σ, L_a, L_s*) = (*"pasivo",* ∞, L_a, L_s)

δ_{cont} (s, x) = δ_{ext} (δ_{int} (s), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación el externo.

λ (*"activo", σ, L_a, L_s*) = (*"out"*, 1) (Envía un mensaje que contiene un uno)

ta (*fase, σ, L_a, L_s*) = *σ*

- <u>*entrada*</u>:

DEVS = < X_M, S, Y_M, δ_{ext}, δ_{int}, δ_{cont}, λ, ta >

$X_M = \{(p,v) | p \in InPorts, v \in X_p\}$

InPorts = { *"in", "inQ1", inQ2", "inQ3"*}

X_{in} : X_{inQ1} : X_{inQ2} : X_{inQ3}: R^+

$Y_M = \{(p,v) | p \in OutPorts, v \in Y_p\}$

OutPorts = {*" out1"*, *"out2"*, *"out3"*}

Y_{out1} : Y_{out2} : Y_{out3} : R^+

S = (*fase, σ, q1, q2, q3*) = {*"activo", "pasivo"*} x R^+_0 x R^+_0 x R^+_0 x R^+_0

q1, q2, q3 son las variables que almacenan los valores del número de clientes que hay en cada surtidor.

δ_{ext} (*"pasivo", σ, q1, q2, q3, e*, (*"in"*, x)) = (*"activo"*, 0, *q1*, *q2*, *q3*)

δ_{ext} (*"pasivo", σ, q1, q2, q3, e*, (*"inQ1"*, x)) = (*"pasivo", σ – e, x, q2, q3*)

δ_{ext} (*"pasivo", σ, q1, q2, q3, e*, (*"inQ2"*, x)) = (*"pasivo", σ – e, q1, x, q3*)

δ_{ext} (*"pasivo", σ, q1, q2, q3, e*, (*"inQ3"*, x)) = (*"pasivo", σ – e, q1, q2, x*)

δ_{int} (*"activo", σ, q1, q2, q3*) = (*"pasivo",* ∞, *q1, q2, q3*)

δ_{cont} (*s,* x) = δ_{ext} (δ_{int} (*s*), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación el externo.

$$\lambda(\text{"activo"}, \sigma, q1, q2, q3) = \begin{cases} (\text{"out1"}, x) & \text{si } (q1<q2 \ \& \ q1<q3) \\ (\text{"out2"}, x) & \text{si } (q2<q1 \ \& \ q2<q3) \\ (\text{"out3"}, x) & \text{si } (q3<q1 \ \& \ q3<q2) \end{cases}$$

x es el valor que se acaba de recibir en el puerto "*in*".

Si no hay una cola menor que las demás, se utiliza una función de probabilidad para seleccionar el puerto por el que se envía la entidad.

ta (*fase, σ, q1, q2, q3*) = *σ*

- *surtidor*:

$DEVS_{\Delta}$ = < X_M, S, Y_M, δ_{ext}, δ_{int}, δ_{cont}, λ, ta >

Δ : Parámetro del modelo. Tiempo de proceso. Se calcula cada vez mediante la función de probabilidad *uniforme* con parámetros 2 y 5.

etiqueta: Parámetro del modelo que identifica a cada *surtidor*.

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$

InPorts = { *"in", "inEnd"*}

X_{in} : X_{inEnd} : R^+

$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$

OutPorts = {" *out*", "*outQ*", "*transd*"}

Y_{out1} : Y_{outQ} : Y_{transd} : R^+

S = (*fase*, σ, *q*, *numQ*, *resto*) =

= {"*pasivo*","*busy*","*wait*","*end*","*sendQ*","*sendQ2*","*sendOutQ*","*sendOutQ2*"} x R^+_0 x V^+ x R^+_0 x R^+_0

q almacena una secuencia finita de valores que se encuentran en cola, pertenecientes a R^+. Esta secuencia se representa con V^+.

numQ almacena el valor del número de elementos que hay en *q*.

resto almacena el tiempo de proceso que falta.

δ_{ext} ("*pasivo*", σ, *q*, *numQ*, *resto*, *e*, ("*in*", x)) =
= ("*sendOutQ*", 0, *q* • x, *numQ + 1*, Δ)

δ_{ext} ("*busy*", σ, *q*, *numQ*, *resto*, *e*, ("*in*", x)) =
= ("*sendQ*", 0, *q* • x, *numQ + 1*, $\sigma - e$)

δ_{ext} ("*wait*", σ, *q*, *numQ*, *resto*, *e*, ("*in*", x)) =
= ("*sendQ2*", 0, *q* • x, *numQ + 1*, *resto*)

δ_{ext} ("*wait*", σ, *v* • *q**, *numQ*, *resto*, *e*, ("*inEnd*", x)) =
= ("*end*", 0, *q**, *numQ – 1*, *resto*)

El símbolo • representa la concatenación.
q • x indica que la entidad x se añade al final de la cola *q*.
v • *q** indica que la cola *q* está compuesta por el elemento *v*, que es el que se ha terminado de procesar, y por la cola *q**.

δ_{int} ("*sendQ*" , σ, *q*, *numQ*, *resto*) = ("*busy*", *resto*, *q*, *numQ*, *resto*)

δ_{int} ("*busy*" , σ, *q*, *numQ*, *resto*) = ("*wait*", ∞, *q*, *numQ*, *resto*)

δ_{int} ("*sendQ2*" , σ, *q*, *numQ*, *resto*) = ("*wait*", ∞, *q*, *numQ*, *resto*)

δ_{int} ("*end*", σ, *q*, *numQ*, *resto*) = { ("*pasivo*", ∞, *q*, *numQ*, *resto*) si *numQ* = 0 ; ("*sendOutQ2*", *0*, *q*, *numQ*, Δ) en otro caso }

δ_{int} ("*sendOutQ*", σ, *q*, *numQ*, *resto*) = ("*sendQ*", *0*, *q*, *numQ*, *resto*)

δ_{int} ("*sendOutQ2*", σ, *q*, *numQ*, *resto*) = ("*busy*", *resto*, *q*, *numQ*, *resto*)

δ_{cont} (*s*, x) = δ_{ext} (δ_{int} (*s*), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación el externo.

λ (*"sendQ", σ, q, numQ, resto*) = (*"outQ", numQ*)

λ (*"sendQ2", σ, q, numQ, resto*) = (*"outQ", numQ*)

λ (*"end", σ, q, numQ, resto*) = (*"outQ", numQ*)

λ (*"busy", σ, v • q*, numQ, resto*) = (*"out", v + etiqueta*)

λ (*"sendOutQ", σ, v • q*, numQ, resto*) = (*"transd", v*)

λ (*"sendOutQ2", σ, v • q*, numQ, resto*) = (*"transd", v*)

ta (*fase, σ, q, numQ, resto*) = *σ*

- *transdSurt*:

$DEVS_{\Delta} = < X_M, S, Y_M, \delta_{ext}, \delta_{int}, \delta_{cont}, \lambda, ta >$

Δ : Parámetro del modelo. Tiempo que permanece activo.

$X_M = \{(p,v) \mid p \in InPorts,\ v \in X_p\}$

InPorts = { *"ariv", "solved1", "solved2", "solved3"*}

$X_{ariv}: X_{solved1} : X_{solved2} : X_{solved3} : R^+$

$Y_M = \{\ \}$

S = (*fase, σ, L_a, L_{s1}, L_{s2}, L_{s3}*) = {*"activo", "pasivo"*} x R^+_0 x V^+ x V^+ x V^+ x V^+

> L_a, L_{s1}, L_{s2} y L_{s3} almacenan una lista finita de valores, pertenecientes a R^+. Esta secuencia se representa con V^+.

δ_{ext} (*"activo", σ, L_a, L_{s1}, L_{s2}, L_{s3} , e*, (*"ariv"*, x)) =
= (*"activo", σ – e, L_a + x , L_{s1}, L_{s2}, L_{s3}*)

δ_{ext} (*"activo", σ, L_a, L_{s1}, L_{s2}, L_{s3}, e*, (*"solved1"*, x)) =
= (*"activo", σ – e, L_a, L_{s1} + x, L_{s2}, L_{s3}*)

δ_{ext} (*"activo", σ, L_a, L_{s1}, L_{s2}, L_{s3}, e*, (*"solved2"*, x)) =
= (*"activo", σ – e, L_a, L_{s1}, L_{s2} + x, L_{s3}*)

δ_{ext} (*"activo", σ, L_a, L_{s1}, L_{s2}, L_{s3}, e*, (*"solved3"*, x)) =
= (*"activo", σ – e, L_a, L_{s1}, L_{s2}, L_{s3}* + x)

> L_a + x indica que la entidad x se añade a la lista L_a

δ_{int} (*"activo", σ, L_a, L_{s1}, L_{s2}, L_{s3}*) = (*"pasivo"*, ∞, *L_a, L_{s1}, L_{s2}, L_{s3}*)

δ_{cont} (*s*, x) = δ_{ext} (δ_{int} (*s*), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación el externo.

λ (*"activo", σ, L_a, L_{s1}, L_{s2}, L_{s3}*) = { }

ta (*fase, σ, L_a, L_{s1}, L_{s2}, L_{s3}*) = *σ*

- <u>*colaCajas*</u>:

DEVS = < X_M, S, Y_M, δ_{ext}, δ_{int}, δ_{cont}, λ, ta >

X_M = {(*p,v*) | *p* ϵ*InPorts, v* ϵX_p}

InPorts = { *"in"*, *"estado1"*, *"estado2"*}

X_{in} : $X_{estado1}$: $X_{estado2}$: R^+

Y_M = {(*p,v*) | *p* ϵ*OutPorts, v* ϵY_p}

OutPorts = {"*out1*", "*out2*", "*outQ*"}

Y_{out1} : Y_{out2} : Y_{outQ} : R^+

S = (*fase, σ, q, numQ, pasivo1, pasivo2*) =

= {*"pasivo","busy","wait","sendQ","sendQ2"*} x R^+_0 x V^+ x {*true*, *false*} x {*true*, *false*}

q almacena una secuencia finita de valores que se encuentran en cola, pertenecientes a R^+. Esta secuencia se representa con V^+.

numQ almacena el valor del número de elementos que hay en *q*.

pasivo1 y *pasivo2* son variables booleanas que indican si está libre (*true*) u ocupada (*false*) cada una de las cajas.

δ_{ext} ("*pasivo*",*σ, q,numQ, pasivo1, pasivo2, e*,((*"in"*,x_1),(*"in"*,x_2),…,(*"in"*,x_n))) =
= ("*sendQ*", *0, q* • {x_1, x_2,…,x_n}, *numQ + n,pasivo1,pasivo2*)

δ_{ext} ("*pasivo*", *σ, q, numQ, pasivo1, pasivo2, e*,(*"estado1"*, x)) =
= ("*sendQ2*", 0, *q, numQ, true, pasivo2*)

δ_{ext} (*"pasivo"*, *σ, q, numQ, pasivo1, pasivo2, e,*(*"estado2"*, x)) =
= (*"sendQ2"*, 0, *q, numQ, pasivo1, true*)

δ_{ext} (*"wait"*, *σ, q, numQ, pasivo1, pasivo2, e,* ((*"in"*,x_1),(*"in"*,x_2),…,(*"in"*,x_n))) =
= (*"sendQ", 0, q* • {x_1, x_2,…,x_n}*, numQ + n,pasivo1,pasivo2*)

δ_{ext} (*"wait"*, *σ, q, numQ, pasivo1, pasivo2, e,*(*"estado1"*, x)) =
= (*"sendQ"*, 0, *q, numQ, true, pasivo2*)

δ_{ext} (*"wait, σ, q, numQ, pasivo1, pasivo2, e,*(*"estado2"*, x)) =
= (*"sendQ"*, 0, *q, numQ, pasivo1, true*)

δ_{ext} (*"busy"*, *σ, q, numQ, pasivo1, pasivo2, e,* ((*"in"*,x_1),(*"in"*,x_2),…,(*"in"*,x_n))) =
=(*"busy",σ – e, q* • {x_1, x_2,…,x_n}*,numQ + n,pasivo1,pasivo2*)

δ_{ext} (*"busy"*, *σ, q, numQ, pasivo1, pasivo2, e,*(*"estado1"*, x)) =
= (*"busy", σ – e, q, numQ, true, pasivo2*)

δ_{ext} (*"pasivo"*, *σ, q, numQ, pasivo1, pasivo2, e,*(*"estado2"*, x)) =
= (*"busy", σ – e, q, numQ, pasivo1, true*)

(*"in",x_1*),(*"in",x_2*),…,(*"in",x_n*) representa la bolsa de eventos de entrada que pueden llegar simultáneamente al puerto *"in"*

q • {x_1, x_2,…,x_n} indica que todas las entidades se agregan a la cola.

δ_{int} (*"sendQ", σ, v • q*, numQ, pasivo1, pasivo2*) =

$$= \begin{cases} (\text{"busy"}, 0, q^*, numQ - 1, pasivo1, pasivo2) & \text{si } (pasivo1 = true \mid\mid pasivo2 = true) \\ (\text{"wait"}, \infty, v \bullet q^*, numQ, pasivo1, pasivo2) & \text{en otro caso} \end{cases}$$

δ_{int} (*"sendQ", σ, q, numQ, pasivo1, pasivo2*) =
= (*"pasivo", ∞, q, numQ, pasivo1, pasivo2*)

δ_{int} (*"busy", σ, q, numQ, pasivo1, pasivo2*) =

$$= \begin{cases} (\text{"pasivo"}, \infty, q, numQ, pasivo1, pasivo2) & \text{si } numQ = 0 \\ (\text{"sendQ"}, 0, q, numQ, pasivo1, pasivo2) & \text{si } ((numQ \neq 0)\ \&\&\ (pasivo1=true \mid\mid pasivo2=true)) \\ (\text{"wait"}, \infty, q, numQ, pasivo1, pasivo2) & \text{en otro caso} \end{cases}$$

$\delta_{cont}(s, x^b) = \delta_{ext}(\delta_{int}(s), 0, x^b)$

En caso de que llegue una bolsa de eventos externos en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación los externos.

λ (*"sendQ", σ, q, numQ, pasivo1, pasivo2*) = (*"outQ", numQ*)

λ (*"sendQ2", σ, q, numQ, pasivo1, pasivo2*) = (*"outQ", numQ*)

λ (*"busy", σ, v • q*, numQ, pasivo1, pasivo2*) =

$$= \begin{cases} (\text{"out1"}, v)\ \&\&\ (pasivo1 = false) & \text{si } (pasivo1=true\ \&\&\ pasivo2 = false) \\ (\text{"out2"}, v)\ \&\&\ (pasivo2 = false) & \text{si } (pasivo2=true\ \&\&\ pasivo1 = false) \end{cases}$$

Si las dos cajas están libres, se utiliza una función de probabilidad para seleccionar el puerto por el que se envía la entidad.

ta (*fase, σ, q, numQ, pasivo1, pasivo2*) = *σ*

- <u>*cajaSimp*</u>:

$DEVS_\Delta = < X_M, S, Y_M, \delta_{ext}, \delta_{int}, \delta_{cont}, \lambda, ta >$

Δ : Parámetro del modelo. Tiempo de proceso. Se calcula cada vez mediante la función de probabilidad *uniforme* con parámetros 0.5 y 2.

$X_M = \{(p,v) | p \in InPorts, v \in X_p\}$

InPorts = { *"in"*}

$X_{in} : R^+$

$Y_M = \{(p,v) | p \in OutPorts, v \in Y_p\}$

OutPorts = {*"out"*, *"out1"*, *"out2"*, *"out3"*, *"estado"*}

$Y_{out} : Y_{out1} : Y_{out2} : Y_{out3} : R^+$

$Y_{estado} : \{0\}$

S = (*fase, σ*) = {*"activo", "pasivo"*} x R^+_0

δ_{ext} (*"pasivo"*, *σ*, *e*, (*"in"*, x)) = (*"activo"*, Δ)

δ_{int} (*"activo"*, *σ*) = (*"pasivo"*, ∞)

δ_{cont} (*s*, x) = δ_{ext} (δ_{int} (*s*), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación el externo.

λ (*"activo"*, σ) =
- (*"out"*, *v*), (*"out1"*, *etiqueta*), (*"estado"*, 0) si *etiqueta* identifica al *surtidor1*
- (*"out"*, *v*), (*"out2"*, *etiqueta*), (*"estado"*, 0) si *etiqueta* identifica al *surtidor2*
- (*"out"*, *v*), (*"out3"*, *etiqueta*), (*"estado"*, 0) si *etiqueta* identifica al *surtidor3*

Las entidades x que se reciben en el puerto de entrada están formadas por *v + etiqueta*.

La *etiqueta* es un parámetro de *surtidor*. Antes de enviarla, *surtidor* se la añadió a la entidad.

ta (*fase*, σ) = σ

- <u>*transdCaja*</u>:

DEVS$_\Delta$ = < X_M, S, Y_M, δ_{ext}, δ_{int}, δ_{cont}, λ, ta >

Δ : Parámetro del modelo. Tiempo que permanece activo.

$X_M = \{(p,v) | p \in InPorts, v \in X_p\}$

InPorts = { "*ariv*", "*solved*"}

X_{ariv}: X_{solved} : R^+

Y_M = { }

S = (*fase*, σ, L_a, L_s) = {*"activo"*, *"pasivo"*} x R^+_0 x V^+ x V^+

L_a *y* L_s almacenan una lista finita de valores, pertenecientes a R^+. Esta secuencia se representa con V^+.

δ_{ext} (*"activo"*, σ, L_a, L_s , *e*, (*"ariv"*, x)) = (*"activo"*, σ – *e*, L_a + x , L_s)

δ_{ext} (*"activo"*, σ, L_a, L_s, *e*, (*"solved"*, x)) = (*"activo"*, σ – *e*, L_a, L_s + *x*)

L_a + x indica que la entidad x se añade a la lista L_a

δ_{int} (*"activo", σ, L_a, L_s*) = (*"pasivo",* ∞, L_a, L_s)

δ_{cont} (*s*, x) = δ_{ext} (δ_{int} (*s*), 0, x)

En caso de que llegue un evento externo en el mismo instante en que está planificada una transición interna, se ejecuta primero el evento interno y a continuación el externo.

λ (*"activo", σ, L_a, L_s*) = { }

ta (*fase, σ, L_a, L_s*) = *σ*

4.3 Descripción con el Lenguaje de Definición DEVS

En el Apartado 2.4 se explicaron las reglas de especificación del Lenguaje de Definición de Zeigler. Según dichas reglas, el modelo gasolinera puede describirse de la siguiente forma:

```
begin description
overall model : gasolinera

atomic model Generador
      inports: stop;
      outports: out;
      state variables:
            status in {active, passive};
            contador in {integer};
      initial condition:
            status:= active;
            contador:= 1;
      internal transition:
            {status = active}
                => {status := active && contador:= contador + 1};
```

```
      external transition:
            {status = active} * stop => {status := passive};
      output function:
            {status = active} => out;
end Generador


atomic model Transd
      inports: ariv; solved;
      outports: out;
      state variables:
            status in {active, passive};
      initial condition:
            status := active;
      internal transition:
            {status = active} => {status := passive};
      external transition:
            { };    /* Ninguna entrada produce una transición */
      output function:
            {status = active} => out;
end Transd


atomic model Entrada
      inports: in; inQ1; inQ2; inQ3;
      outports: out1; out2; out3;
      state variables:
            status in {active, passive};
            q1 in {integer}; q2 in {integer}; q3 in {integer};
      initial condition:
            status := passive;
            q1 := 0; q2 := 0; q3 := 0;
```

```
    internal transition:

        {status = active} => {status := passive};

    external transition:

        {status = passive} * in => {status := active};

        {status = passive} * inQ1

            => {status:= passive && q1:= valor_en_inQ1};

        {status = passive} * inQ2

            => {status:= passive && q2:= valor_en_inQ2};

        {status = passive} * inQ3

            => {status:= passive && q3:= valor_en_inQ3};

            /* valor_en_inQ1 es el valor de tipo integer
            que se recibe en el puerto inQ1. Lo mismo con
            valor_en_inQ2 y valor_en_inQ3 */

    output function:

        {q1 < q2 && q1 < q3} => out1;

        {q2 < q3 && q2 < q1} => out2;

        {q3 < q1 && q3 < q2} => out3;

            /* Si las tres colas (q1, q2, q3) tienen el
            mismo tamaño o dos de ellas son iguales y las
            más cortas, una función aleatoria decide el
            puerto de salida */

end Entrada
```

```
atomic model Surtidor

    inports: in; inEnd;

    outports: out; outQ; transd;

    state variables:

        status in {passive, busy, wait, end, sendQ, sendQ2,

                    sendOutQ, sendOutQ2};

        q in {integer};
```

```
    internal transition:
        {status = sendQ} => {status := busy};
        {status = busy} => {status := wait};
        {status = sendQ2} => {status := wait};
        {status = end && q > 0} => {status := sendOutQ2};
        {status = end && q = 0} => {status := passive};
        {status = sendOutQ} => {status := sendQ};
        {status = sendOutQ2} => {status := busy};
    external transition:
        {status = passive} * in
                => {status := sendOutQ && q := q + 1};
        {status = busy} * in
                => {status := sendQ && q := q + 1};
        {status = wait} * in
                => {status := sendQ2 && q := q + 1};
        {status = wait} * inEnd
                => {status := end && q := q - 1};
    output function:
        {status = sendQ || status = sendQ2 || status = end}
                => outQ;
        {status = busy} => out;
        {status = sendOutQ || status = sendOutQ2} => transd;
end Surtidor


atomic model TransdSurt
    inports: ariv; solved1; solved2; solved3;
    outports: { };
    state variables:
        status in {active, passive};
    initial condition:
        status := active
```

```
    internal transition:
        {status = active} => {status := passive};
    external transition:
        { };  /* Ninguna entrada produce una transición */
    output function:
        { };
end TransdSurt


atomic model ColaCajas
    inports: in; estado1; estado2;
    outports: out1; out2; outQ;
    state variables:
        status in {passive, busy, wait, sendQ, sendQ2};
        q in {integer};
        pasivo1 in {true, false};
        pasivo2 in {true, false};
    initial condition:
        status := passive;
        q := 0;
        pasivo1 := true;  /*Indica que la caja 1 está libre*/
        pasivo2 := true;  /*Indica que la caja 2 está libre*/
    internal transition:
        {status = sendQ && (pasivo1 = true||pasivo2 = true)}
            => {status := busy && q := q - 1};
        {status = sendQ && (pasivo1=false && pasivo2=false)}
            => {status := wait};
        {status = sendQ2} => {status := passive};
        {status = busy && q>0 &&(pasivo1=true||pasivo2=true)}
            => {status := sendQ};
        {status= busy &&q>0 &&(pasivo1=false&&pasivo2=false)}
            => {status := wait};
```

```
        {status = busy && q = 0} => {status := passive};
    external transition:
        {status = passive} * in
              =>  {status := sendQ && q := q + 1};
        {status = passive} * estado1
              =>  {status := sendQ2 && pasivo1:= true};
        {status = passive} * estado2
              =>  {status := sendQ2 && pasivo2:= true};
        {status = wait} * in
              =>  {status := sendQ && q := q + 1};
        {status = wait} * estado1
              =>  {status := sendQ && pasivo1 := true};
        {status = wait} * estado2
              =>  {status := sendQ && pasivo2 := true};
        {status = busy} * in
              => {status := busy && q := q + 1};
        {status = busy} * estado1
              => {status := busy && pasivo1 := true};
        {status = busy} * estado2
              => {status := busy && pasivo2 := true};
    output function:
        {status = sendQ || status = sendQ2} => outQ;
        {status = busy && pasivo1 = true && pasivo2 = false}
              => out1 && {pasivo1 := false};
        {status = busy && pasivo1 = false && pasivo2 = true}
              => out2 && {pasivo2 := false};
        /* Cuando las dos cajas están libres una función
        aleatoria elige el puerto de salida */
end ColaCajas
```

```
atomic model CajaSimp
      inports: in;
      outports: out; out1; out2; out3; estado;
      state variables:
            status in {passive, busy};
      initial condition:
            status := passive;
      internal transition:
            {status = busy} => {status := passive};
      external transition:
            {status = passive} * in => {status := busy};
                  /* Las entidades que llegan al puerto de
                  entrada tienen una etiqueta que identifica al
                  surtidor que las procesó */
      output function:
            {status = busy} => {out, out1, estado};
                  /*si la entidad fue procesada en el surtidor1*/
            {status = busy} => {out, out2, estado};
                  /*si la entidad fue procesada en el surtidor2*/
            {status = busy} => {out, out3, estado};
                  /*si la entidad fue procesada en el surtidor3*/
end CajaSimp
```

```
atomic model TransdCaja
      inports: ariv; solved;
      outports: { };
      state variables:
            status in {active, passive};
      initial condition:
            status := active;
      internal transition:
            {status = active} => {status := passive};
```

```
        external transition:
              { };  /* Ninguna entrada produce una transición */
        output function:
              { };
end TransdCaja


coupled model Ef
        components:
              Generador g; Transd t;
        internal coupling:
              g.out -> t.ariv;
              t.out -> g.stop;
        external input coupling:
              Ef.in -> t.solved;
        external output coupling:
              g.out -> Ef.out;
end Ef


coupled model SetCajas
        components:
              ColaCajas colaCajas; CajaSimp caja1; CajaSimp caja2;
              TransdCaja tranCaja;
        internal coupling:
              colaCajas.out1 -> caja1.in;
              colaCajas.out2 -> caja2.in;
              colaCajas.out1 -> tranCaja.solved;
              colaCajas.out2 -> tranCaja.solved;
              caja1.estado -> colaCajas.estado1;
              caja2.estado -> colaCajas.estado2;
```

```
        external input coupling:
                SetCajas.in -> colaCajas.in;
                SetCajas.in -> tranCaja.ariv;
        external output coupling:
                colaCajas.outQ -> SetCajas.outQ;
                caja1.out -> SetCajas.out;
                caja2.out -> SetCajas.out;
                caja1.out1 -> SetCajas.outEnd1;
                caja1.out2 -> SetCajas.outEnd2;
                caja1.out3 -> SetCajas.outEnd3;
                caja2.out1 -> SetCajas.outEnd1;
                caja2.out2 -> SetCajas.outEnd2;
                caja2.out3 -> SetCajas.outEnd3;
end SetCajas


coupled model NetGasolinera
        components:
                Ef ef; SetCajas cajas; Entrada entrada;
                TransdSurt tranSurt; Surtidor surtidor1;
                Surtidor surtidor2; Surtidor surtidor3;
        internal coupling:
                ef.out -> entrada.in;
                entrada.out1 -> surtidor1.in;
                entrada.out2 -> surtidor2.in;
                entrada.out3 -> surtidor3.in;
                surtidor1.out -> cajas.in;
                surtidor1.outQ -> entrada.inQ1;
                surtidor2.out -> cajas.in;
                surtidor2.outQ -> entrada.inQ2;
                surtidor3.out -> cajas.in;
                surtidor3.outQ -> entrada.inQ3;
```

```
            cajas.outEnd1 -> surtidor1.inEnd;
            cajas.outEnd2 -> surtidor2.inEnd;
            cajas.outEnd3 -> surtidor3.inEnd;
            cajas.out -> ef.in;
            entrada.out1 -> tranSurt.ariv;
            entrada.out2 -> tranSurt.ariv;
            entrada.out3 -> tranSurt.ariv;
            surtidor1.transd -> tranSurt.solved1;
            surtidor2.transd -> tranSurt.solved2;
            surtidor3.transd -> tranSurt.solved3;
      external input coupling:
            NetGasolinera.in -> entrada.in;
      external output coupling:
            cajas.out -> NetGasolinera.out;
end NetGasolinera
```

4.4 Descripción y Simulación con DEVSJAVA

Los archivos con el código de las clases del modelo se pueden ver en el Anexo B, al final de esta Memoria. También están incluidos en el CD del Proyecto, dentro de la carpeta *gasolinera* que está en la carpeta *Gasolinera*.

Para comenzar la descripción del modelo y realizar el modelado con DEVSJAVA, se debe crear un proyecto nuevo en Eclipse, como se ha explicado en el Apartado 2.2.1. Al proyecto se le asigna el nombre *Gasolinera*. A continuación se crea un paquete en el proyecto recién creado, con el nombre *gasolinera*, que es el que va a contener las clases del modelo. Todos los modelos, tanto atómicos como acoplados, que se han descrito anteriormente

en el Apartado 4.2, se corresponderán con una clase, y por lo tanto, con uno de los archivos *.java* que hay que crear, y que se detallan a continuación.

Las clases del modelo que se corresponden con modelos atómicos heredan de la clase *ViewableAtomic*, y las que representan a modelos acoplados heredan de *ViewableDigraph*. *ViewableAtomic* y *ViewableDigraph* están en el paquete *SimView*, y así los modelos podrán ser visualizados con las aplicaciones de dicho paquete. Si no se quisieran aprovechar las utilidades gráficas del paquete *SimView*, las clases de los modelos atómicos heredarían directamente de *Atomic*, y las de los modelos acoplados de *Coupled Atomic* y *Coupled*. Ambas son clases del paquete *genDevs.modeling*. Estos dos paquetes pertenecen a la biblioteca de clases de DEVSJAVA, *coreDEVS.jar*

Los mensajes que se intercambian entre los modelos deben ser instancias de la clase *entity*, del paquete *GenCol* de *coreDEVS.jar*. En el caso de que sean cantidades numéricas, serán instancias de la clase *doubleEnt* del mismo paquete.

Para la creación de los archivos del modelo, vamos a utilizar como base, algunos de los archivos de la carpeta *SimpArc*, que se encuentra dentro del archivo *IllustratinCode.zip*, el cual se ha bajado de la página web de DEVSJAVA.

Generador.java: En la carpeta *SimpArc* se encuentra el archivo *genrRand.java*, que es muy similar al que se necesita para el modelo. Utiliza la función *rand.expon(double)*, del paquete *statistics* de *coreDEVS.jar*, para los tiempos entre creación de entidades.

El método constructor necesita como parámetros, además del nombre, dos números de tipo *double.* Uno lo utiliza como parámetro de la función *exponencial*, y el otro lo emplea como semilla al generar la función *rand*.

Dispone de un contador, que incrementa cada vez que crea una nueva entidad. De este modo puede asignar un número a cada entidad, que la identifica y diferencia de las demás. En el método *out()*se asigna esta identificación. A continuación se muestra el código correspondiente a dicho método:

```
public message  out( ) {
        return outputNameOnPort("coche" + count, "out");
}
```

Como puede verse, en este modelo a las entidades se las identifica con la cadena de caracteres siguiente: `"coche" + count`, donde el primer término, `"coche"`, es igual para todas y el segundo término, `count`, es el valor que tiene la variable contador, cuando se crea la entidad.

Transd.java: Para la creación de este archivo, se dispone de *trands.java* en la carpeta *SimpArc*.

El método constructor tiene dos parámetros, el nombre y uno de tipo *double*, que es el que señala el final de la simulación.

Tiene dos listas, las cuales son instancias de la clase *Function*, del paquete *GenCol* de *coreDEVS.jar*. En una de las listas se van almacenando los valores de las entidades que son recibidas por el puerto *ariv*, y el momento en

que han llegado. Cuando llega un mensaje al puerto *solved*, comprueba si el valor de la entidad que contiene está en la primera lista, y en caso de que sea así, lo guarda en la segunda junto con el tiempo que ha transcurrido entre los dos mensajes, y actualiza los valores estadísticos del modelo.

Cada vez que llega un mensaje a alguno de los puertos, se activa el método *show_state()*, el cual imprime en la consola el tiempo que falta para que termine la simulación, en minutos, el número de coches llegados, el de los terminados y el tiempo medio de permanencia. A continuación se muestra un ejemplo recogido durante la simulación:

```
sigma : 860.5888087825407 minutos.
Total coches llegados = 111 coches.
Total coches terminados = 108 coches.
Tiempo medio de permanencia = 4.880515890455998 minutos.
```

Al finalizar la simulación se obtiene:

```
sigma : 0.0 minutos.
Total coches llegados = 299 coches.
Total coches terminados = 299 coches.
Tiempo medio de permanencia = 4.978128636045093 minutos.
```

Ef.java: En la carpeta *SimpArc* está el archivo *ef.java*. Solamente contiene el método constructor, el cual, además del nombre, tiene dos parámetros de tipo *double,* que son, el intervalo entre llegadas y el tiempo de simulación. Estos parámetros los utiliza para crear las instancias de *generador* y *transd.* En este modelo, el intervalo de llegadas es 5 minutos, y el tiempo de simulación es 1440 minutos (24 horas). Además establece los enlaces entre los puertos de todos los modelos.

<u>*Entrada.java*</u>: Se puede partir de *proc.java*, de la carpeta *SimpArc*, ya que se trata de un proceso sin cola. Hay que tener en cuenta que el tiempo de proceso es cero.

El modelo tiene tres variables, *q1*, *q2* y *q3*, de tipo *integer*, en las que va guardando los valores del número de entidades que hay en cada uno de los tres *surtidores*. Cada vez que recibe un mensaje con un nuevo valor, actualiza la variable correspondiente.

En el método *out()* selecciona el *surtidor* al que debe enviar la entidad que acaba de recibir, teniendo en cuenta el valor de las variables *q1*, *q2* y *q3*. Envía la entidad al *surtidor* cuya variable es más pequeña que las otras dos. En los casos en los que haya dos iguales, y que sean las de menor valor, o que las tres sean iguales, se aplica la función de probabilidad *rand.iuniform(int)*, del paquete *statistics* de *coreDEVS.jar*, para decidir el surtidor al que se envía la entidad. Para ello, primero en el método *initalize()* se crean cuatro instancias de la clase *rand*, con semillas diferentes, con el fin de alcanzar una mayor aleatoriedad. Después, se utiliza una de las instancias *rand* en cada una de las cuatro alternativas que pueden presentarse. Cuando la igualdad es entre dos variables, se emplea la función *rand.iuniform(1)*, que genera un 0 ó un 1. Cuando es entre las tres, se usa *rand.iuniform(2)*, para obtener 0, 1 ó 2. Según sea el número obtenido, se deriva la entidad a un *surtidor* u otro.

Por ejemplo, si las tres variables son iguales, el código es el que se reproduce a continuación:

```
else if((q1==q2)&(q1==q3)){   //las tres colas son iguales

      n = r4.iuniform(2);     // genera un número
                              // aleatorio: 0,1 ó 2

      if (n % 3 == 0){
            con = makeContent("out3",job);
      }
      else if(n % 3 == 1) {
            con = makeContent("out1",job);
      }
      else  {
            con = makeContent("out2",job);
      }
}
```

Surtidor.java: El archivo de la carpeta *SimpArc* utilizado como referencia en esta ocasión es *procQ.java*, ya que es un proceso con una cola.

Para calcular el tiempo que dura el proceso, se utiliza la función de probabilidad *rand.uniform(2, 5)*, del paquete *statistics* de *coreDEVS.jar*. Cada uno de los *surtidores* crea la instancia de la clase *rand* con una semilla diferente, para intentar que los tiempos de proceso sean lo menos parecidos posible. Todos los valores de las semillas son números primos.

Posee una variable, que es una instancia de la clase *Queue*, la cual está en el paquete *GenCol* de *coreDEVS.jar*, con la que gestiona la cola.

No se hace una declaración de la lista de estados que contiene el modelo porque en DEVSJAVA no es necesario.

En el método *out()*, después de terminar el procesado de la entidad y antes de enviarla hacia las *cajas*, se le añade el nombre del *surtidor* al valor que la identifica, para que, cuando la caja termine de procesarla, reconozca el

surtidor al que debe enviar el mensaje de fin de proceso. El código es el siguiente:

```
if (phaseIs("busy")) {
      String stJob = job.toString() + name;
      entity newJob = new entity(stJob);
      m.add(makeContent("out",newJob));
}
```

donde `name` es el nombre del surtidor. Puede ser *surtidor1*, *surtidor2* o *surtidor3*.

Los métodos están construidos para que el modelo funcione como se detalló en el Apartado anterior. Por ejemplo, si el modelo se encuentra en el estado *busy* cuando se produce la llegada de una entidad, en el método *deltext(double e, message x)* se interrumpe este estado, para comunicar inmediatamente a *entrada* el nuevo número de entidades en el *surtidor*, por medio del estado *sendQ*. Para ello, se almacena el tiempo que queda de proceso en una variable llamada *resto*. A continuación, en el método *deltint()* se realiza una transición al estado *busy*, con el tiempo de proceso que restaba. Los dos fragmentos de código se muestran a continuación:

```
public void  deltext(double e, message   x){
. . . . . . . . . . . . . . .
      else if (phaseIs("busy")){
            for (int i=0; i< x.size();i++)
                  if (messageOnPort(x, "in", i))       {
                        entity jb = x.getValOnPort("in", i);
                        q.add(jb);
                        resto = sigma;
                        holdIn("sendQ", 0);
                  }
      }
. . . . . . . . . . . . . . .
```

```
public void  deltint( ) {
. . . . . . . . . . . . . . .
      if (phaseIs("sendQ")) {
            holdIn("busy",resto);
      }
. . . . . . . . . . . . . . .
```

TransdSurt.java: Para su creación se utiliza el archivo *transd.java*, de la carpeta *SimpArc*.

Su método constructor requiere dos parámetros, el nombre, y uno de tipo *double*, que señala el tiempo de duración del proceso de este modelo. Se le asigna un tiempo de 1500 minutos, mayor que el de la simulación, para dar tiempo a que terminen de ser procesadas las últimas entidades llegadas al sistema.

Como debe llevar un seguimiento de la utilización de las colas de los tres *surtidores*, este modelo posee cuatro listas, que son instancias de la clase *Function*, del paquete *GenCol* de *coreDEVS.jar*. En una de ellas lleva el registro de los valores de las entidades que recibe por el puerto *ariv*, y el momento en que han llegado. Las otras tres las utiliza para cada uno de los *surtidores*. Cuando llega un mensaje a uno de los puertos de entrada, *solved1, solved2* o *solved3*, comprueba si el valor está en la primera lista, y en caso de que sea así, lo guarda en la lista correspondiente al puerto por el que ha llegado el mensaje, junto con el tiempo que ha transcurrido entre los dos mensajes, y actualiza los valores estadísticos por medio de los métodos de los que dispone para este fin. Cuenta también con un método para truncar los decimales y dejarlos sólo en

dos, para que los datos en la consola se puedan visualizar con una mayor claridad.

Al finalizar el tiempo de duración del modelo se activa el método *show_state()*, por medio del cual, se imprimen en la consola los valores estadísticos de las colas de todos los *surtidores*. Al tener un tiempo de proceso mayor que el tiempo de simulación, los datos se imprimirán cuando *Transd.java* haya terminado de imprimir los de la última entidad que ha abandonado el sistema. A continuación se muestra la impresión en la consola al terminar la simulación.

```
Total coches que han pasado por el Surtidor 1 = 108 coches.
Total coches que no han esperado en la cola = 100.0 coches. (92.59%).
Total coches que han esperado más de tres minutos = 4.0 coches. (3.7%).
Tiempo máximo de espera en la cola del Surtidor 1 = 4.93 minutos.
Tiempo medio de espera total en la cola del Surtidor 1 = 0.18 minutos.
Tiempo medio de espera de los que han estado en la cola = 2.48 minutos.

Total coches que han pasado por el Surtidor 2 = 95 coches.
Total coches que no han esperado en la cola = 89.0 coches. (93.68%).
Total coches que han esperado más de tres minutos = 4.0 coches. (4.21%).
Tiempo máximo de espera en la cola del Surtidor 2 = 4.64 minutos.
Tiempo medio de espera total en la cola del Surtidor 2 = 0.21 minutos.
Tiempo medio de espera de los que han estado en la cola = 3.27 minutos.

Total coches que han pasado por el Surtidor 3 = 96 coches.
Total coches que no han esperado en la cola = 88.0 coches. (91.67%).
Total coches que han esperado más de tres minutos = 3.0 coches. (3.13%).
Tiempo máximo de espera en la cola del Surtidor 3 = 5.83 minutos.
Tiempo medio de espera total en la cola del Surtidor 3 = 0.25 minutos.
Tiempo medio de espera de los que han estado en la cola = 2.97 minutos.
```

ColaCajas.java: Este modelo está basado en un proceso con una cola, por lo tanto se puede utilizar el archivo *procQ.java*, de la carpeta *SimpArc*.

El proceso es de tiempo cero, ya que las entidades, al abandonar la cola, salen en dirección a la *caja* que esté libre.

Para gestionar la cola, utiliza una variable, que es una instancia de la clase *Queue*, la cual está en el paquete *GenCol* de *coreDEVS.jar*.

En las variables *pasivo1* y *pasivo2*, de tipo *boolean*, almacena el estado de las *cajas*.

En el caso de que las dos *cajas* se encuentren libres, utiliza la función de probabilidad *rand.iuniform(int)*, del paquete *statistics* de *coreDEVS.jar*, para decidir la *caja* a la que debe enviar la entidad. Primero, en el método *initalize()*, crea una instancia de la clase *rand*. Como la igualdad es entre dos variables emplea la función *rand.iuniform(1)*, que genera un 0 ó un 1. El código es similar al que se mostró en el caso del archivo *Entrada.java*.

Para efectuar una simulación en la cual el sistema funcione con una sola *caja*, solamente es necesario activar la siguiente línea, que pertenece al método *initialize()*.

```
//    pasivo2 = false; //activando esta línea se anula la caja 2
```

Aunque este modelo contiene varios estados, no se hace una declaración de la lista de estados porque en DEVSJAVA no es necesario. Los métodos están construidos de manera que el modelo sigue el funcionamiento que se detalló en el Apartado anterior, cuando se realizó su descripción.

CajaSimp.java: Es un proceso sin cola, por lo que el archivo de *SimpArc* que sirve de modelo es *proc.java*.

El tiempo de proceso no es fijo, sino que sigue la función de probabilidad *rand.uniform(0.5, 2)*, del paquete *statistics* de *coreDEVS.jar*. Las instancias de la clase *rand* de cada una de las *cajas*, se crea con una semilla diferente, con objeto de que el proceso en cada *caja* sea lo más independiente posible del otro.

Cuando termina de procesar a la entidad y antes de enviar ningún mensaje, primero separa el nombre del *surtidor*, que fue añadido por el propio *surtidor* cuando lo envió hacia las *cajas*, del valor de la entidad. De este modo puede identificar el *surtidor* al que tiene que enviar el mensaje de fin de proceso. Esta acción se realiza en el método *out()*, y parte del código se muestra a continuación.

```
String stJob = job.toString();

if(stJob.endsWith("surtidor1")) {
      entity job1 =
            new entity(stJob.substring(0,stJob.length()-9));
      entity surt = new entity("surtidor1");
      m.add(makeContent("out",job1));
      m.add(makeContent("out1",surt));
      m.add(makeContent("estado",pasivo));
}
```

Se puede ver como la entidad `job` es convertida en un *String* llamado `stJob`. Luego se comprueba si `stJob` termina en `"surtidor1"`, y en caso de que sea así, se crea una nueva entidad, suprimiéndole a `stJob` los últimos nueve caracteres, para que coincida con la que creó *generador*, y de este modo pueda ser reconocida por *transd*. Finalmente se crea la entidad `surt`, y se envían los mensajes.

TransdCaja.java: Se trata de un archivo muy similar a *TransdSurt.java*. La mayor diferencia consiste en que solamente contiene dos instancias de la clase *Function*, ya que únicamente hace el seguimiento de una cola. El tiempo de duración se fija en 1500 minutos, por las mismas razones que se hizo en *TransdSurt.java*. Al final de la simulación los datos estadísticos que se muestran en la consola son los siguientes:

```
Total clientes que han pasado por las Cajas = 299 clientes.
Total clientes que no han esperado en la cola = 294.0 clientes. (98.33%).
Total clientes que han esperado más de un minuto = 1.0 clientes. (0.33%).
Tiempo máximo de espera en la cola de Cajas = 1.18 minutos.
Tiempo medio de espera total en la cola de Cajas = 0.01 minutos.
Tiempo medio de espera de los que han estado en la cola = 0.43 minutos.
```

SetCajas.java: Este archivo solamente contiene el método constructor, que carece de parámetros. Crea una instancia de *ColaCajas*, dos de *CajaSimp* y otra de *TransdCaja*. Para poder visualizar las instancias en distintos colores se emplea el siguiente fragmento de código, en el que se establece que *ColaCajas* aparezca de color cyan, y las dos cajas de color verde.

```
Color color;
color = Color.cyan;
colaCajas.setBackgroundColor(color);
color = Color.green;
caja1.setBackgroundColor(color);
caja2.setBackgroundColor(color);
```

Tanto en *Ef.java* como en *NetGasolinera.java* se emplea también este código para cambiar el color de algunas instancias. Por defecto, las instancias se visualizan en *SimView* de color gris. También se crean las conexiones entre los puertos de todos los componentes del modelo acoplado.

NetGasolinera.java: Este archivo corresponde al modelo acoplado que acoge al sistema completo, por tanto en él se crea una instancia de cada uno de los otros dos modelos acoplados, e instancias de los modelos atómicos que todavía no pertenecen a ningún otro modelo acoplado. El código que corresponde a la creación de todas estas instancias es el siguiente:

```
ViewableDigraph ef = new Ef("ef",5,1440);// nombre, intervalo
                                          //    de generación
                                         // y tiempo de generación
ViewableDigraph cajas = new SetCajas();

ViewableAtomic entrada = new Entrada("entrada");
ViewableAtomic surtidor1 = new Surtidor("surtidor1");
ViewableAtomic surtidor2 = new Surtidor("surtidor2");
ViewableAtomic surtidor3 = new Surtidor("surtidor3");
ViewableAtomic tranSurt = new TransdSurt("transdSurt", 1500);
```

También crea cuatro instancias de la clase *CellGridPlot*, que pertenece al paquete *SimView* de *coreDEVS.jar.* Estas instancias sirven para visualizar una gráfica de cada una de las colas del modelo, tres de los *surtidores* y una de las *cajas*. Al puerto de entrada *timePlot* de cada una de estas instancias, se le conecta el puerto de salida *outQ* de un *surtidor* o el puerto de salida *outQ* de *ColaCajas*. Estos puertos son los encargados de enviar el número de personas que se encuentran en cada momento en el *surtidor* o en *SetCajas*. A continuación se muestra el código correspondiente a la gráfica de la cola del *surtidor1*:

```
CellGridPlot cochesSurtidor1 = new CellGridPlot("Coches en
Surtidor 1",10,10);
cochesSurtidor1.setCellGridViewLocation(500,0);
cochesSurtidor1.setSpaceSize(100,25);
cochesSurtidor1.setCellSize(5);
cochesSurtidor1.setTimeScale(1500);
add(cochesSurtidor1);
addCoupling(surtidor1,"outQ",cochesSurtidor1,"timePlot");
```

Finalmente crea las conexiones entre los puertos de todos los modelos que contiene.

El aspecto de Eclipse, después de crear todos los archivos del modelo, es el siguiente:

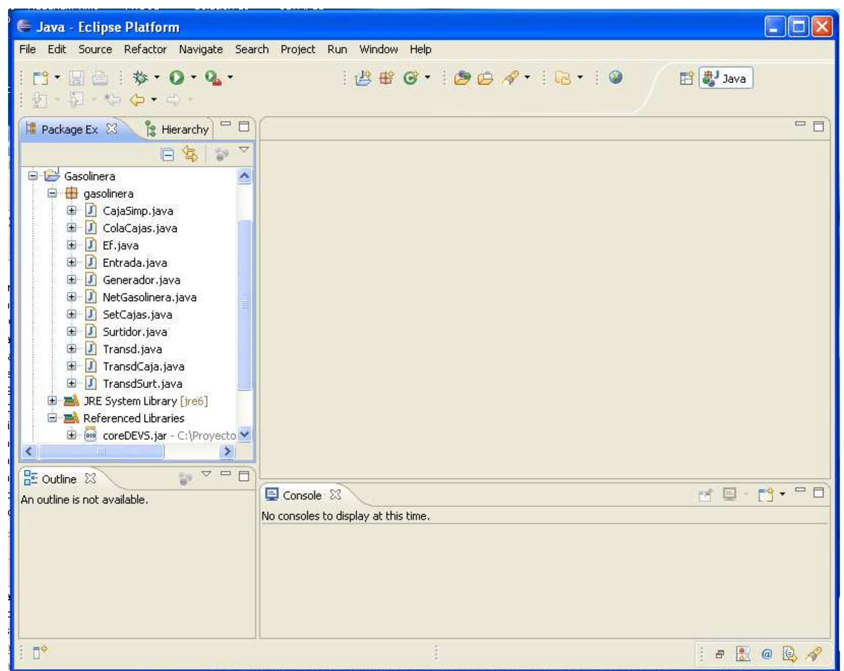


Figura 4.5: Aspecto del entorno Eclipse con todos los archivos del modelo *Gasolinera*.

Una vez terminados los archivos del modelo, se debe ejecutar *SimView*, como está explicado en los pasos 10 y 11 del Apartado 2.2.1. En la ventana de *SimView*, pulsar el botón *configure*. En el cuadro *Model package names*, escribir *gasolinera* y pulsar *OK*. En *Select a package*, seleccionar *gasolinera,* y en *Select a model*, seleccionar *NetGasolinera.* Aparecen cinco ventanas. La ventana principal y más grande, llamada *DEVSJAVA Simulation Viewer*, corresponde a la vista de la simulación, y en ella pueden observarse todos los modelos atómicos y acoplados del sistema. Las otras cuatro ventanas corresponden a las

entidades de *CellGridPlot*, que reflejan de forma gráfica, el valor del número de entidades que hay en los *surtidores* y en *SetCajas*, en cada momento.

En la Figura 4.6 se muestra el aspecto de la ventana *DEVSJAVA Simulation Viewer*, antes del comienzo de la simulación. Los botones de la parte inferior sirven para controlar la simulación. Con el deslizante *real time factor*, se aumenta o disminuye el tiempo de la simulación. El botón *step* hace que se ejecute un paso de la simulación, el botón *run* la lanza de modo continuo, pero se puede pausar pulsando *step*. El botón *restart* reinicia la simulación.

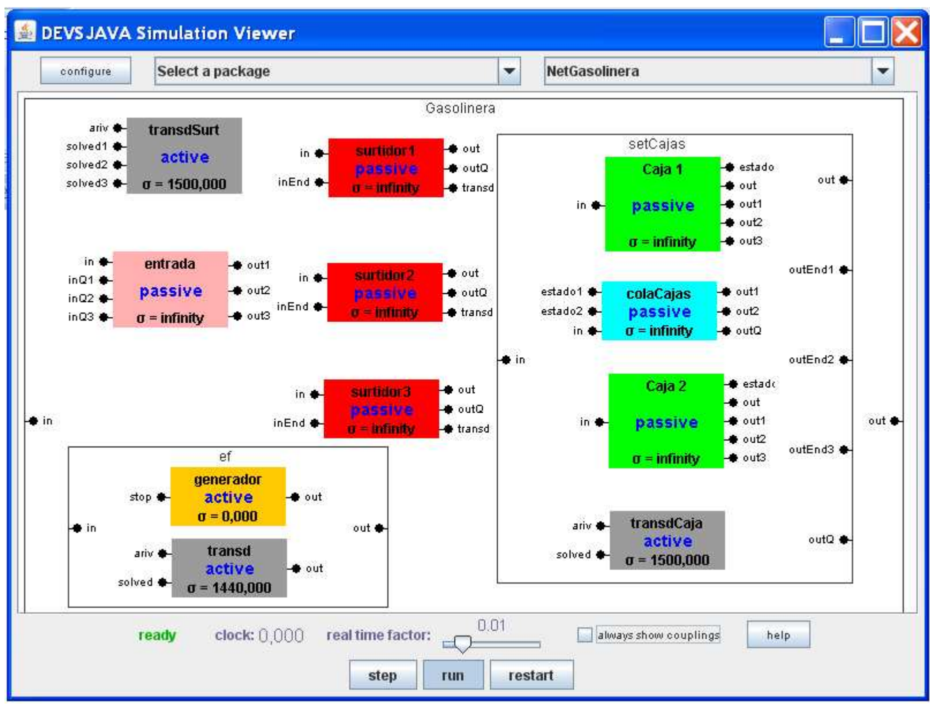


Figura 4.6: Ventana principal de *SimView* antes de comenzar la simulación.

Durante la simulación, y pulsando el botón *step*, se puede visualizar el paso de mensajes entre los modelos. Cada vez que se pulse *step*, se produce un paso de la simulación. En la Figura 4.7 se muestra un momento de la

simulación. Se observa que está activada la casilla *always show couplings*, lo que permite que estén visibles los enlaces entre los modelos.

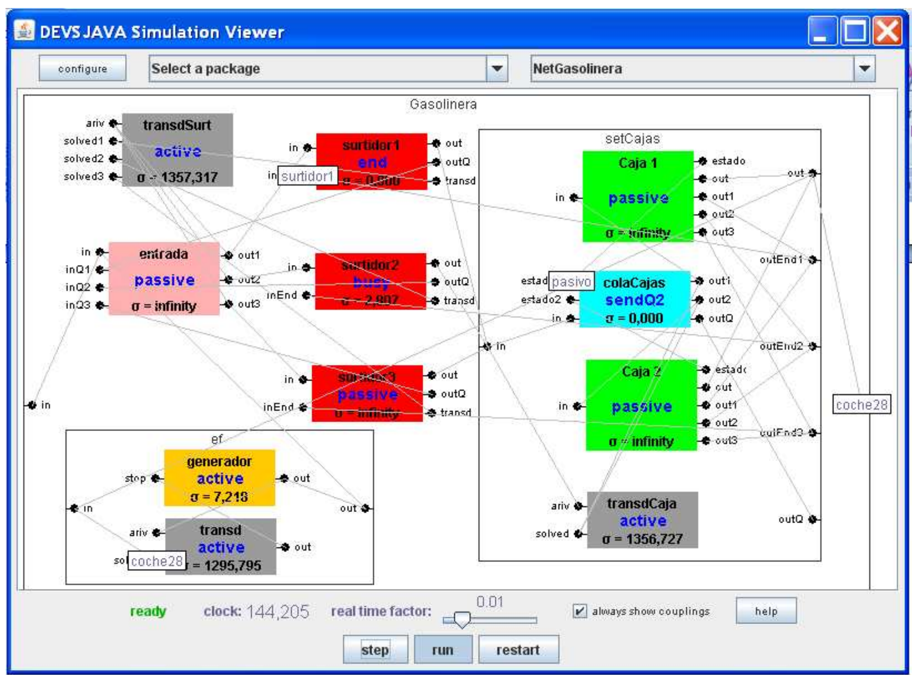


Figura 4.7: Ventana principal de *SimView* en pausa durante la simulación.

Si se sitúa el cursor del ratón sobre la representación gráfica de alguno de los modelos, se obtiene información de su estado, y otras variables. Esta información se ha configurado en el método *public String getTooltipText()*, que contienen todos los modelos atómicos. Como ejemplo, a continuación se muestra el código de dicho método en el archivo ColaCajas.java.

```
public String getTooltipText(){
      return
      super.getTooltipText()
      +"\n"+"Caja 1 pasivo: " + pasivo1
      +"\n"+"Caja 2 pasivo: " + pasivo2
      +"\n"+"tamaño de la cola: " + q.size()
      +"\n"+"elementos en la cola: " + q.toString();
}
```

Se puede ver en la Figura 4.8 el resultado de situar el cursor sobre *ColaCajas* durante una pausa de la simulación.

Se muestran por defecto, el valor actual del estado del modelo (*phase*), el valor del tiempo que falta hasta la próxima transición interna (*sigma*), el tiempo global de simulación en que ocurrió el último evento (*tL*), y el tiempo global en que está previsto el próximo evento (*tN*). Se entiende por tiempo global, el tiempo que ha transcurrido desde el comienzo de la simulación. En todo momento se cumple la función de avance del tiempo: $tN = tL + t_a(s)$.

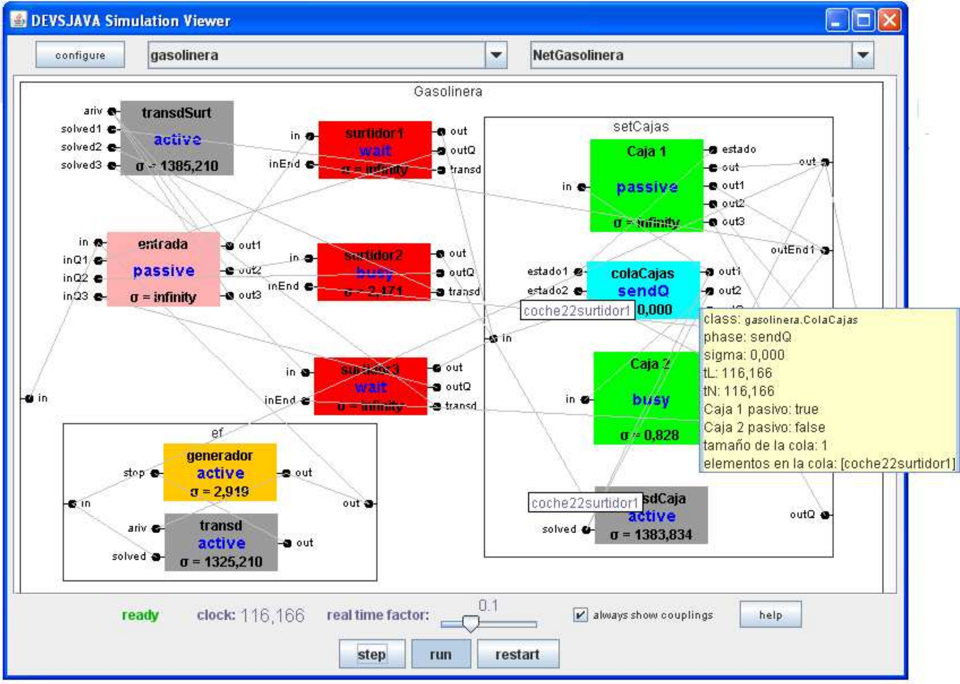


Figura 4.8: Resultado de situar el cursor sobre *ColaCajas* durante una pausa de la simulación.

Además de estos cuatro valores, se obtienen también los que se hayan configurado en el método *public String getTooltipText()*. En este caso son, el valor de las variables que indican el estado de las dos *cajas*, el valor del número de entidades que hay en la cola, en ese instante, y la lista de dichas entidades.

Las otras cuatro ventanas tienen todas el mismo aspecto. Durante la simulación van indicando el número de entidades que hay en cada instante, en el modelo al que representan. La Figura 4.9, muestra el resultado de la ventana *Personas en Caja*, y la Figura 4.10, de *Coches en Surtidor 1, 2 y 3*, durante una pausa de la simulación.

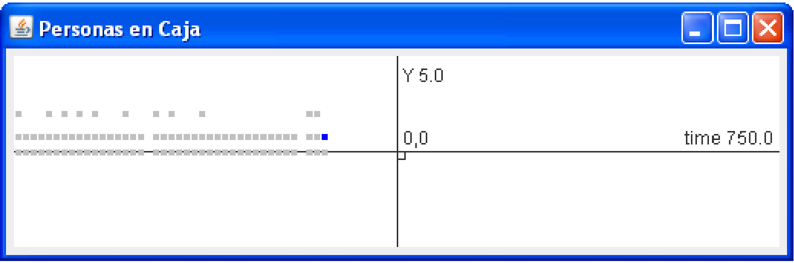


Figura 4.9: Aspecto de la ventana *Personas en Caja* durante una pausa de la simulación.

Figura 4.10: Aspecto de las ventanas *Coches en Surtidor 1, 2 y 3* durante una pausa de la simulación.

4.5 Descripción y Simulación con CD++

Los archivos con el código de las clases del modelo se pueden ver en el Anexo C, al final de esta Memoria. También están incluidos en el CD del Proyecto, dentro de la carpeta *gasolineraCD++ Gasolinera*.

El primer paso para comenzar la descripción del modelo y realizar el modelado con CD++, será crear un proyecto nuevo en Eclipse, como se explicó en el Apartado 2.3.1. Al proyecto se le asigna el nombre *gasolineraCD++*, y acogerá todos los archivos necesarios para la simulación del modelo.

Los modelos atómicos descritos en el Apartado 4.2 se corresponderán con las parejas de archivos con extensión *.h* y *.cpp.* Los archivos *.h* son archivos de cabecera, destinados a las declaraciones de funciones y variables, que después serán utilizadas en los archivos *.cpp*. Estos archivos se describirán a continuación. Los modelos acoplados descritos en ese mismo apartado se construirán como archivos de extensión *.ma*, que también se detallarán más adelante.

Para crear algunos de los archivos de modelos atómicos, se pueden utilizar, como referencia o punto de partida, los archivos de la carpeta *internal* de *CD++Builder*, que está dentro de Eclipse. Para localizarla, hay que situarse en el directorio donde se instaló Eclipse, y seguir la siguiente ruta: *eclipse\plugins\CD++Builder_1.1.0\internal*.

Los mensajes se intercambian en forma de instancias de tipo numérico. Pueden ser *integer*, *double* o *Value. Value* es un tipo que está definido en la carpeta *internal*, y posee métodos propios.

La información de los valores estadísticos se imprimen en un archivo llamado *salida.txt*. Este archivo no es necesario crearlo, porque, si no existiese, se creará automáticamente al realizar la simulación.

Cada vez que se cree un modelo atómico, debe registrarse en un archivo llamado *register.cpp*, que forma parte del proyecto. A continuación, se comenta la creación de los archivos necesarios para el modelo, comenzando por *register.cpp*.

register.cpp: En la carpeta *internal* existe un archivo con este mismo nombre, que puede servir de patrón. Cada vez que se crea un modelo atómico, hay que añadir unas líneas en este archivo. Por ejemplo, cuando se cree el modelo entrada habrá que añadir, al principio del archivo:

```
#include "entrada.h"
```

a continuación, dentro del método:

```
void MainSimulator::registerNewAtomics()
```

se añade la siguiente línea:

```
SingleModelAdm::Instance().
    registerAtomic( NewAtomicFunction<Entrada>() , "Entrada" ) ;
```

y se procede de la misma forma con todos los modelos atómicos.

generador.h y *generador.cpp:* En la carpeta *internal*, se encuentran los archivos *generat.h* y *generat.cpp*, que son muy parecidos a los que se necesitan para el modelo.

Para calcular el tiempo entre la generación de entidades, este modelo utiliza una instancia de la clase abstracta *Distribution.* Esta clase está definida, junto con las clases que heredan de ella, en el archivo *distri.h*, de la carpeta *internal*. La clase heredera de *Distribution*, que se vaya a utilizar, hay que describirla, junto con los parámetros necesarios para su creación, en el archivo *.ma*, del modelo acoplado al que pertenezca *generador*. El método constructor de *generador*, busca estos valores en dicho archivo *.ma*, y crea la instancia. Aquí se va a utilizar la clase *ExponentialDistribution* que corresponde a una función de probabilidad exponencial. El parámetro hay que indicarlo en segundos. Como el intervalo de generación requerido por el modelo es de cinco minutos, el parámetro será igual a 300.

El modelo dispone de tres variables, para llevar la cuenta del número de entidades generadas, y aplicarles un valor que las identifique. La variable *pid*, que acumula el valor total de las unidades creadas. La variable *initial*, que sirve para indicar el valor que se le asignará a la primera entidad creada, y la variable *increment*, que indica el valor en que hay que incrementar la variable *pid*, cada vez que se genere una nueva entidad. El método constructor se encarga de comprobar si los valores de *initial* e *increment* están definidos en el archivo *.ma*

correspondiente. Si no los encuentra, les asigna a ambos el valor 1. Aquí se utilizan estos valores.

En el método correspondiente a la función de salida, se le asigna un número a cada entidad creada. Este valor servirá para identificarla y diferenciarla de las demás. El código de dicho método se muestra a continuación:

```
Model &Generador::outputFunction( const InternalMessage &msg ) {
      sendOutput( msg.time(), out, pid ) ;
      pid += increment;
      return *this ;
}
```

transd.h y transd.cpp: Para crear estos archivos, se dispone de *transduc.h* y *transduc.cpp*, en la carpeta *internal*.

El método constructor comprueba si se ha definido el valor del tiempo que dura la simulación, en el archivo *.ma* del modelo acoplado correspondiente. Si no lo encuentra, utiliza el que está marcado por defecto, que en este modelo es de 24 horas.

Para almacenar los tiempos de llegada de las entidades, utiliza una variable de la clase *map*, en la que, cada vez que recibe una entidad, incluye el valor del tiempo de llegada, junto con el valor de la entidad. Cuando llega el mensaje desde las *cajas* de una entidad terminada, el modelo comprueba cuál fue su tiempo de llegada, y establece la diferencia. Esta diferencia será el tiempo de permanencia de la entidad en el sistema.

También cuenta con dos variables de tipo *list*, una para almacenar las entidades que llegan desde *generador*, y otra para almacenar las que llegan desde las *cajas*. De este modo mantiene actualizadas las variables acumuladoras, para poder ofrecer los datos estadísticos.

Cada vez que llega un mensaje a alguno de los puertos, se imprime, en el archivo *salida.txt*, el tiempo que falta para que termine la simulación, el número de coches llegados, el de los terminados y el tiempo medio de permanencia en minutos.

A continuación se muestra un ejemplo recogido durante la simulación:

```
sigma : 10:20:06:000
Total coches llegados = 171 coches.
Total coches terminados = 168 coches.
Tiempo medio de permanencia = 4.76091 minutos.
```

Al final de la simulación se obtiene:

```
sigma : 00:00:00:000
Total coches llegados = 283 coches.
Total coches terminados = 283 coches.
Tiempo medio de permanencia = 4.8977 minutos.
```

ef.ma: En CD++ existe un lenguaje específico para la creación de los modelos acoplados, como ya se comentó en el Apartado 2.3. El modelo completo está compuesto por un único archivo con extensión *.ma*. En este archivo, se añaden todos los demás modelos acoplados que existan. Para este modelo, el archivo general será *netGasolinera.ma*, y en él se incluirán *ef.ma* y el resto de modelos acoplados. Aquí se muestra a continuación el código completo de *ef.ma*:

```
[top]
components : gen@generador transd@transd

in : in
out : out

Link : in solved@transd
Link : out@transd stop@gen
Link : out@gen arrived@transd
Link : out@gen out


[gen]
distribution : exponential
mean : 300
```

Se observa la estructura del archivo, donde, en primer lugar, aparecen los modelos que lo componen, después los puertos que posee, y los enlaces. Al final, se especifican los valores que necesita *generador*, para crear la función de distribución de probabilidad.

entrada.h y entrada.cpp: Aunque se trata de un proceso sin cola, se puede partir de los archivos *queue.h* y *queue.cpp*, de la carpeta *internal*, haciendo los reajustes necesarios, ya que la estructura de los archivos es muy similar a la que se necesita. Conviene tener en cuenta, que el tiempo de proceso es cero.

Dispone de tres variables, *q1*, *q2* y *q3*, de tipo *double*, que utiliza para ir guardando los valores del número de entidades que hay en cada uno de los tres *surtidores.* Cada vez que recibe un mensaje con un nuevo valor, actualiza la variable correspondiente.

En el método correspondiente a la función de salida, selecciona el *surtidor* al que debe enviar la entidad que acaba de recibir, teniendo en cuenta

el valor de las variables *q1*, *q2* y *q3*. Selecciona aquel *surtidor* cuya variable tenga el valor más pequeño, y le envía la entidad. Cuando dos variables sean iguales y las de menor valor, o cuando las tres sean iguales, utiliza la función *rand()*, junto con el operador *resto,* para elegir el *surtidor* al que se debe enviar la entidad. La función *rand()* genera uniformemente un número aleatorio en el intervalo (0, 1). Por ejemplo, si las tres variables son iguales el código que se utiliza es el que se reproduce a continuación:

```
else if ((q1==q2)&&(q1==q3)) {
      int n ;
      n = rand() % 3;
      if(n == 0) {
            sendOutput( msg.time(), out3, valor ) ;
      }
      else if(n == 1){
             sendOutput( msg.time(), out1, valor );
      }
      else if(n == 2){
            sendOutput( msg.time(), out2, valor );
      }
}
```

surtidor.h y surtidor.cpp: Es un proceso con cola, por lo que se pueden utilizar los archivos *queue.h* y *queue.cpp*, de la carpeta *internal*.

Como el modelo tiene varios estados, es necesario declararlos en el archivo de cabecera y crear una variable, tal y como se muestra en el siguiente fragmento de código:

```
enum State {passive, busy, sendOutQ, sendOutQ2, sendQ,
            sendQ2, wait, end};

State estado;
```

Al realizar esta declaración ya no va a ser posible utilizar el método *Model &holdIn(const State & ,const Time &)* de la clase *Atomic,* que está

definido en el archivo *atomic.h*, de la carpeta *internal*. En su lugar, se deberá emplear el método *Model &nextChange(const Time &)* de la clase *Model* definido en *model.h*, junto con la variable *estado*, que se acaba de crear. Así, en vez de escribir:

```
holdIn(wait, Time::Inf);
```

se debe escribir:

```
estado = wait;
nextChange(Time::Inf);
```

El tiempo de duración de cada proceso se calcula mediante la función *float genunf(float high, float low)*, que está definida en el archivo *randlib.c*, de la carpeta *internal*. Esta función genera un número real, uniformemente distribuido entre *low* y *high*. Para el funcionamiento correcto de la función, es necesario importar también los archivos *com.c* y *linpack.c*, ya que se necesitan clases que están definidas en ellos. Como los parámetros han de ser asignados en segundos, aquí se utiliza *genunf(120, 300)* para seguir las especificaciones del modelo.

La cola se gestiona mediante una variable de tipo *list*.

En el método correspondiente a la función de salida, se suma una cantidad, que es única para cada *surtidor*, al valor que identifica a una entidad. Las *cajas*, antes de devolver la entidad, le restarán la misma cantidad que se ha sumado aquí. De este modo reconocerá el *surtidor* al que debe enviar el mensaje de fin de proceso. Las cantidades se eligen lo suficientemente grandes,

para que no interfieran con los valores de las entidades. El código es el siguiente:

```
if(estado == busy) {
      double num;
      if(nombre == "surtidor1")  num = 100000;
      if(nombre == "surtidor2")  num = 200000;
      if(nombre == "surtidor3")  num = 300000;
      Value val = valor + num;
      sendOutput( msg.time(), out, val ) ;
}
```

El modelo sigue el funcionamiento que se detalló en el Apartado 4.3, y los métodos están construidos para que se produzcan las transiciones entre los estados de la forma adecuada. Por ejemplo, en el método que corresponde a la función de transición externa, si, cuando se produce la llegada de una entidad, el modelo se encuentra en el estado *busy*, se interrumpe este estado, para comunicar inmediatamente a *entrada* el nuevo número de entidades en el *surtidor*, por medio del estado *sendQ*. El tiempo que queda de proceso se almacena en una variable llamada *resto.* Posteriormente, en el método de la función de transición interna, se efectúa la transición al estado *busy*, con el tiempo almacenado en *resto*. Ambos fragmentos de código son:

```
Model &Surtidor::externalFunction( const ExternalMessage &msg ){
. . . . . . . . . . . . . . . . . .

if (estado == busy) {
      if(msg.port() == in) {
            elements.push_back(msg.value());
            resto = nextChange();
            estado = sendQ;
            nextChange(Time::Zero);
      }
}
. . . . . . . . . . . . . . . . . .
```

```
Model &Surtidor::internalFunction( const InternalMessage & ){

if(estado == sendQ) {

      estado = busy;
      nextChange(Time(resto));
}

. . . . . . . . . . . . . . . . . .
```

<u>*transdSurt.h* y *transdSurt.cpp*</u>: Los archivos de la carpeta *internal* adecuados para este modelo son *transduc.h* y *transduc.cpp.*

El método constructor, primero busca el valor del tiempo de duración del modelo en el archivo *.ma*, en el que está declarado. Si no lo encuentra allí, utiliza el valor asignado por defecto. En este modelo se ha fijado un tiempo de 25 horas, mayor que el de la simulación, para que terminen de ser procesadas las últimas entidades llegadas al sistema.

Utiliza una variable de la clase *map*, en la que incluye el valor del tiempo de llegada de la entidad y su valor, cuando recibe un mensaje de uno de los *surtidores* en el puerto *ariv*. Dispone también de cuatro variables de tipo *list*. En una de ellas, almacena los valores de las entidades que llegan por el puerto *ariv*. Cuando recibe una entidad por uno de los puertos *solved1, solved2* o *solved3*, lo almacena en la variable tipo *list* correspondiente, y comprueba en la variable tipo *map* su tiempo de llegada. Así obtendrá el tiempo de permanencia de la entidad, en la cola del *surtidor* correspondiente. De este modo mantiene actualizadas las variables acumuladoras, para poder ofrecer los datos estadísticos.

Cuando finaliza el tiempo de duración del modelo, imprime en el archivo *salida.txt* los valores estadísticos de las colas de todos los *surtidores*. Al tener un tiempo de proceso mayor que el tiempo de simulación, los datos se imprimirán cuando hayan terminado de imprimirse los de la última entidad que ha abandonado el sistema. Los datos los muestra con dos decimales para conseguir una mayor claridad. A continuación se muestra la impresión en el archivo *salida.txt*, al terminar la simulación.

```
Total coches que han pasado por el Surtidor 1 = 95 coches.
Total coches que no han esperado en la cola = 88 coches. (92.63%).
Total coches que han esperado más de tres minutos = 3 coches. (3.16%).
Tiempo máximo de espera en la cola del Surtidor 1 = 4.02 minutos.
Tiempo medio de espera total en la cola del Surtidor 1 = 0.2 minutos.
Tiempo medio de espera de los que han estado en la cola = 2.7 minutos.

Total coches que han pasado por el Surtidor 2 = 95 coches.
Total coches que no han esperado en la cola = 89 coches. (93.68%).
Total coches que han esperado más de tres minutos = 4 coches. (4.21%).
Tiempo máximo de espera en la cola del Surtidor 2 = 5.87 minutos.
Tiempo medio de espera total en la cola del Surtidor 2 = 0.21 minutos.
Tiempo medio de espera de los que han estado en la cola = 3.32 minutos.

Total coches que han pasado por el Surtidor 3 = 93 coches.
Total coches que no han esperado en la cola = 85 coches. (91.4%).
Total coches que han esperado más de tres minutos = 2 coches. (2.15%).
Tiempo máximo de espera en la cola del Surtidor 3 = 4.38 minutos.
Tiempo medio de espera total en la cola del Surtidor 3 = 0.17 minutos.
Tiempo medio de espera de los que han estado en la cola = 1.94 minutos.
```

colaCajas.h y *colaCajas.cpp*: Como se trata de un proceso con una cola, los archivos de la carpeta *internal* que se pueden utilizar, son *queue.h* y *queue.cpp*.

El modelo tiene varios estados, por lo que hay que declararlos en el archivo de cabecera y crear una variable, del mismo modo que se ha explicado en los archivos del modelo *surtidor*. Tampoco aquí se puede utilizar el método *holdIn*, de la clase *Atomic*, debiendo ser sustituido como se indicó en *surtidor*.

El proceso es de tiempo cero, ya que las entidades, al abandonar la cola, salen en dirección a la *caja* que esté libre.

La cola se gestiona mediante una variable de tipo *list*.

El modelo cuenta con dos variables, *pasivo1* y *pasivo2*, de tipo *boolean*, en las cuales almacena el estado de las *cajas*. Si las dos *cajas* están libres, se hace uso de la función *rand(),* que genera uniformemente un número aleatorio en el intervalo (0, 1), junto con el operador *resto,* para elegir la *caja* a la que se debe enviar la entidad. El código es similar al que se mostró en el caso de los archivos del modelo *entrada*.

Para efectuar una simulación en la cual el sistema funcione con una sola *caja*, solamente es necesario activar la siguiente línea, que pertenece al método **`Model &ColaCajas::initFunction():`**

```
//    pasivo2 = false; //activando esta línea se anula la caja 2.
```

El modelo sigue el funcionamiento que se detalló en el Apartado 4.3, y los métodos están construidos para que se produzcan las transiciones entre los estados de la forma adecuada.

cajaSimple.h y *cajaSimple.cpp*: Es un proceso sin cola, pero los archivos de los que se puede partir son *queue.h* y *queue.cpp*, de la carpeta *internal*, haciendo los reajustes necesarios, ya que la estructura de los archivos es muy similar a la que se necesita.

Para calcular el tiempo de duración de cada proceso, se utiliza la función *float genunf(float high, float low)*, que está definida en el archivo *randlib.c*, de la carpeta *internal*. Por medio de esta función, se obtiene un número real, uniformemente distribuido entre *low* y *high*. Es necesario importar también los archivos *com.c* y *linpack.c*, para que la función trabaje correctamente, ya que utiliza clases que están definidas en ellos. Los parámetros de la función deben estar en segundos, por lo que aquí se utiliza *genunf(30, 120)*, para seguir las especificaciones del modelo.

En el método de la función de salida, se selecciona el *surtidor* al que debe enviar el mensaje de fin de proceso. Para este proceso, utiliza el siguiente código:

```
Model &CajaSimple::outputFunction( const InternalMessage &msg ){

	if(state() == active) {

		if((valor - 300000) > 0) {
			Value val = valor - 300000;
			Value surt = 3;
			sendOutput(msg.time(), out, val);
			sendOutput(msg.time(), out3, surt);
			sendOutput(msg.time(), estado, pasivo);
		}
		else if((valor - 200000) > 0) {
			Value val = valor - 200000;
			Value surt = 2;
			sendOutput(msg.time(), out, val);
			sendOutput(msg.time(), out2, surt);
			sendOutput(msg.time(), estado, pasivo);
		}
		else if((valor - 100000) > 0) {
			Value val = valor - 100000;
			Value surt = 1;
			sendOutput(msg.time(), out, val);
			sendOutput(msg.time(), out1, surt);
			sendOutput(msg.time(), estado, pasivo);
		}
	}

	return *this ;
}
```

transdCaja.h y *transdCaja.cpp*: Estos archivos son muy similares a *transdSurt.h* y *transdSurt.cpp*. La diferencia principal consiste en que en este modelo se hace el seguimiento de una única cola, y, por tanto, solamente se necesitan dos variables tipo *list*. Al término de la simulación, se obtienen en el archivo *salida.txt* los siguientes datos estadísticos:

```
Total clientes que han pasado por las Cajas = 283 clientes.
Total clientes que no han esperado en la cola = 278 clientes. (98.23%).
Total clientes que han esperado más de un minuto = 0 clientes. (0%).
Tiempo máximo de espera en la cola de Cajas = 0.95 minutos.
Tiempo medio de espera total en la cola de Cajas = 0.01 minutos.
Tiempo medio de espera de los que han estado en la cola = 0.48 minutos.
```

setCajas.ma: Este archivo corresponde al modelo acoplado *setCajas*. Crea una instancia de *colaCajas*, dos de *cajaSimple* y otra de *transdCaja*, según se puede ver en el siguiente código:

```
components : colaCajas@colaCajas caja1@cajaSimple
             caja2@cajaSimple tranCaja@transdCaja
```

Después, declara los puertos que tiene y, a continuación, establece los enlaces entre los puertos de todos los modelos.

netGasolinera.ma: Este es el archivo que corresponde al modelo acoplado que recoge a todos los demás modelos del sistema. La declaración de componentes es:

```
components : entrada@entrada surtidor1@surtidor
             surtidor2@surtidor surtidor3@surtidor
             tranSurt@transdSurt ef setCajas
```

Se puede ver cómo crea las instancias de los modelos atómicos y declara los modelos acoplados que contiene. Seguidamente declara los puertos que posee:

```
in : in
out : out
```

Después crea las conexiones entre los puertos de todos los modelos que contiene, incluido él mismo. Finalmente, añade el código de los modelos acoplados.

Con todos estos archivos ya es posible hacer la compilación del modelo. Para ello, hay que pulsar el botón *Build* de la barra de Eclipse, como se explicó en el paso 5, del Apartado 2.3.1. En la ventana *Verbose Mode?*, pulsar *Yes* y, si todo está correcto, se obtendrá en pantalla el siguiente mensaje:

```
Simu was created, ready to start simulations...
```

Para realizar la simulación del modelo, se pulsa el botón *Simu* de la barra de Eclipse, como se explicó en el paso 6, del apartado 2.3.1. Aparece la ventana *Simulate Project*, en la cual hay que cargar una serie de datos. Lo más práctico es crear un archivo con extensión *.bat*, que evite tener que rellenar los datos cada vez que se ejecute la simulación. Antes es conveniente crear en el proyecto dos archivos, llamados *netGasolineraLOG.log* y *netGasolineraOUT.out*. En la ventana *Simulate Project*, se rellenan los datos, como se muestra en la Figura 4.11.

Pulsando ahora el botón *Save as .bat*, se guarda el archivo con el nombre *netGasolinera.bat* en la carpeta del proyecto. Cada vez que sea necesario ejecutar una simulación, se pulsa *Load .bat*, se selecciona *netGasolinera.bat*, y aparecen los datos tal y como fueron grabados.

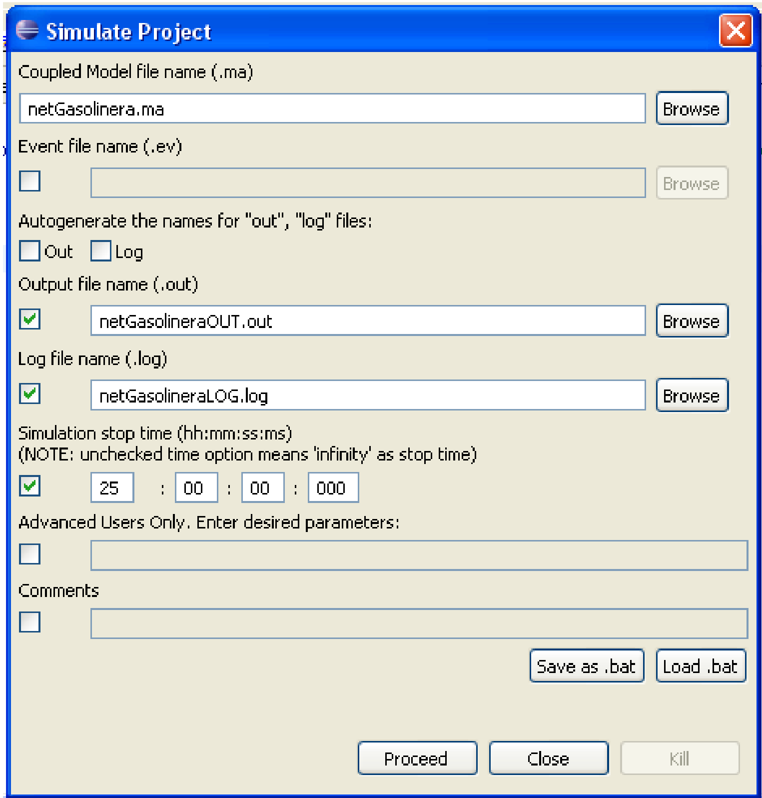


Figura 4.11: Aspecto de la ventana *Simulate Project* con los datos necesarios para la simulación.

El aspecto del proyecto en Eclipse después de crear todos los archivos del modelo es el que se ve en la Figura 4.12.

Después de realizar la simulación, se obtiene un listado en el archivo *netGasolineraLOG.log*, con los datos del intercambio de mensajes que ha habido. También se recogen, en el archivo *netGasolineraOUT.out*, los datos de las entidades que han salido del sistema. En el archivo *salida.txt* se imprimen los datos estadísticos.

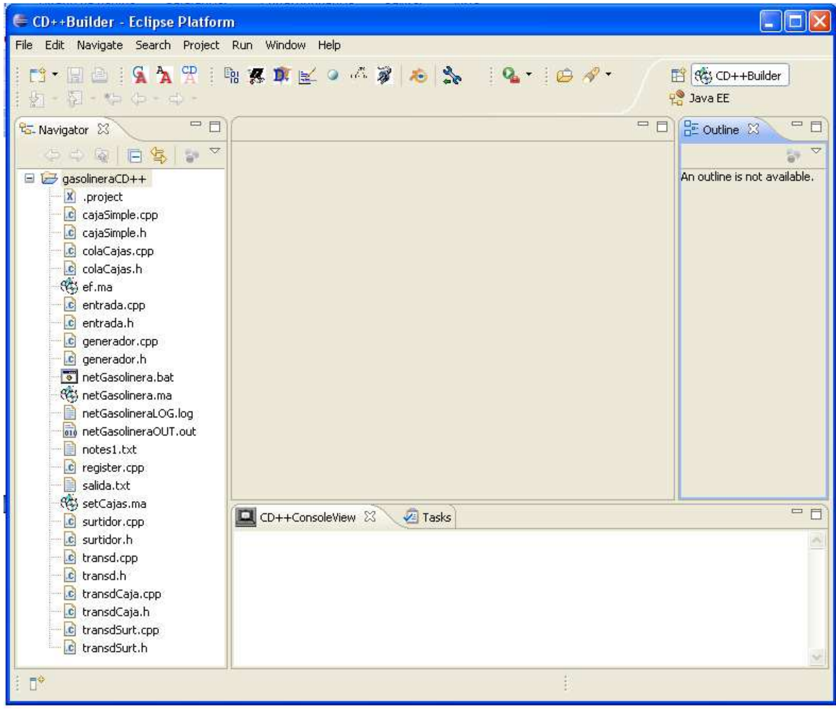


Figura 4.12: Aspecto de Eclipse después de crear todos los archivos del modelo *gasolineraCD++*.

No es posible tener una visualización del modelo, ni una animación con paso de mensajes, ya que, para ello, es necesario disponer de un archivo con extensión *.gam* para los modelos atómicos, y un archivo con extensión *.gcm* para los modelos acoplados. Para generar estos archivos, hay que crear el modelo con la aplicación *CD++ Modeler*. Un modelo de este nivel es bastante complicado de crear de esta manera. Además, no se ha conseguido que *CD++ Modeler* funcione correctamente en la creación gráfica de modelos. Por ejemplo, no crea transiciones internas. La aplicación que se puede bajar de la página web de CD++, tampoco funciona de un modo correcto.

Sin embargo, sí que es posible obtener gráficas, con los datos generados en todos los puertos de salida de los modelos atómicos. Para verlas, hay que pulsar el botón *Animate Atomic Model* de la barra de Eclipse, y aparece la

ventana *atomic animate*. Pulsar el botón del cuadro *Log File*, para buscar, seleccionar y cargar el archivo *netGasolineraLOG.log,* y luego pulsar OK.

Aparece la ventana *Atomic Animate*, que es la que se muestra en la Figura 4.13. Arriba, a la izquierda, es posible seleccionar el modelo que se quiere observar. Debajo, están los puertos del modelo, y a la derecha, las gráficas de estos puertos. Es posible elegir las gráficas que se desean ver, seleccionando el puerto en la casilla outq.

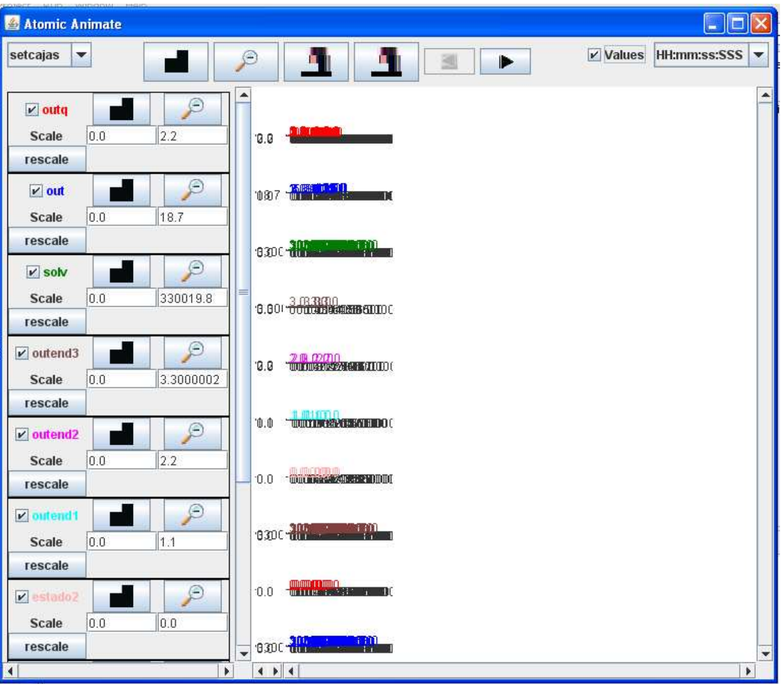


Figura 4.13: Aspecto de la ventana *Atomic Animate* que contiene las gráficas del modelo.

Con los botones de cada puerto, se alarga o se comprime el eje Y de la gráfica. Con los botones iguales a éstos, pero que están en la línea de arriba, se consigue el mismo efecto en el eje X de todas las gráficas. Con los botones se recorre la gráfica hacia atrás o hacia delante, viendo un

intervalo de ella en cada pulsación. La anchura del intervalo se fija utilizando los botones .

Los valores de las gráficas se pueden visualizar marcando la casilla *Values*, y también es posible seleccionar el formato en el que aparece el tiempo.

Los puertos *outQ* de los modelos atómicos *surtidor1*, *surtidor2*, *surtidor3*, muestran las gráficas del número de personas que hay en cada momento de la simulación en el *surtidor* correspondiente. El puerto *outQ* de *colaCajas* muestra la del número de personas en *setCajas*. Esta última gráfica es la que aparece en la Figura 4.14.

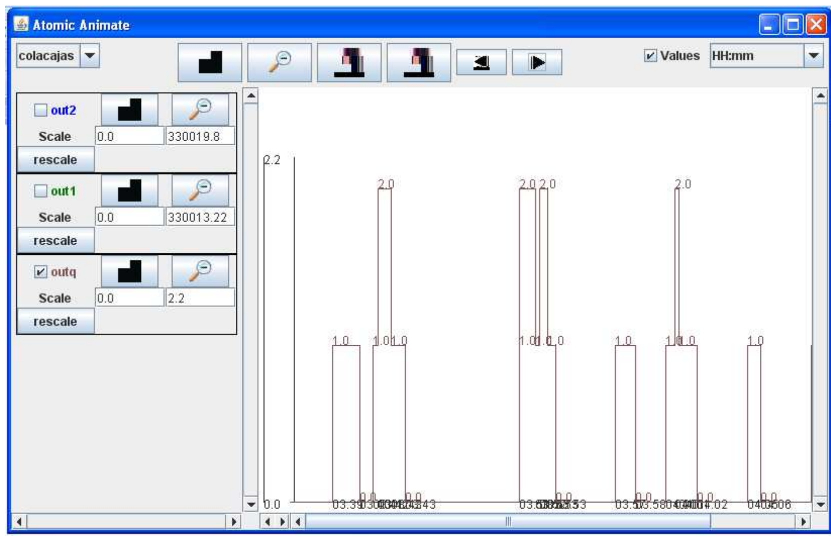


Figura 4.14: Gráfica correspondiente al puerto *outQ* del modelo *colaCajas*.

Conviene señalar que si los datos estadísticos del modelo se muestran en la consola, se ralentiza muchísimo la simulación. Por ese motivo es preferible utilizar el archivo *salida.txt*.

Hay un hecho a destacar en relación con el modelo atómico *colaCajas*. Este modelo recibe en su puerto de entrada *in* las entidades que llegan de los tres *surtidores*. Puede ocurrir que lleguen simultáneamente dos o más entidades a dicho puerto, y en ese caso se perderán algunas de ellas. Para evitar que esto ocurra, el modelo debería construirse como un modelo atómico para simulación paralela. A pesar de que el *Manual de Usuario* [Wainer05], en su página 30 explica la forma de hacerlo, en la carpeta *CD++Builder_1.1.0* no aparecen ninguna de las clases ni archivos necesarios, como *MessageBag(), ParallelMainSimulator*, etc. Tampoco aparecen ejemplos en la página web de la herramienta en los que se haya implementado, y puedan servir de referencia. Con los tiempos de proceso que se han definido para el modelo, da la casualidad de que no se pierde ninguna entidad, pero, con otros tiempos de proceso más pequeños en los *surtidores*, siempre se pierden entidades. Una posible solución sería crear tres puertos de entrada en lugar de uno, de manera que se reciban en cada uno de ellos las entidades de cada uno de los *surtidores*. Esto mejora pero no resuelve completamente el problema, porque para tiempos muy pequeños todavía se siguen perdiendo entidades. Otra solución consistiría en implementar las clases y los métodos que faltan, pero eso está fuera de los objetivos de este Proyecto. Este problema invalida al modelo completo para ser utilizado en ningún tipo de estudio de análisis o mejora del sistema, ya que carece de fiabilidad.

4.6 Conclusiones

El sistema logístico que se ha elegido para continuar el estudio de las herramientas DEVS, simula el comportamiento de los clientes de una gasolinera.

El modelo contiene varios modelos atómicos y acoplados, que intercambian mensajes entre sí.

Su descripción mediante el Lenguaje de Definición DEVS detalla la composición de los modelos y su estructura.

El modelado con las dos herramientas DEVS, no es excesivamente complicado en ninguno de los dos casos. Pero, con DEVSJAVA se tiene una visualización de la simulación y del paso de mensajes entre los modelos, lo cual facilita mucho el trabajo. El modelo que se consigue con CD++ no garantiza una fiabilidad total, debido a que el modelo *colaCajas* no se ha podido construir como un modelo atómico para simulación paralela. En dicho modelo puede ocurrir que se produzca una llegada simultánea de dos o más entidades a uno de sus puertos, y en ese caso se perderían algunas de ellas. Sin embargo, para los parámetros concretos del modelo que se ha definido aquí, no se producen pérdidas de entidades y el modelo funciona.

Los resultados obtenidos en el modelado y la simulación con DEVSJAVA y CD++ son muy similares.

5

Conclusiones y Trabajos futuros

5.1 Introducción

Después de haber efectuado una descripción de las herramientas DEVS, y haber realizado la simulación de dos modelos con ellas, ya podemos extraer las conclusiones del estudio comparativo.

5.2 Conclusiones

El Lenguaje de Especificación DEVS de Bernard Zeigler se ha mostrado muy útil para la definición de los modelos. Es posible establecer una detallada lista de los modelos que lo componen, y la estructura que poseen. Por medio de las reglas BNF que tiene establecidas, es posible la descripción de los modelos DEVS de un modo directo, pudiendo especificar los puertos, las variables y los métodos que contienen.

Tanto DEVSJAVA como CD++ han demostrado ser unas herramientas muy completas y fáciles de manejar, para el modelado y la simulación de modelos DEVS. Sin embargo, conviene destacar las diferencias que se han ido encontrando durante la realización de los trabajos para el estudio comparativo. Una diferencia evidente es que utilizan lenguajes de programación diferentes, pero ello no tiene una importancia muy significativa, ya que tanto Java como C++ son dos lenguajes muy conocidos, y ampliamente utilizados hoy en día en la programación en general. Existen algunas otras diferencias, que se comentan seguidamente.

La instalación de DEVSJAVA resulta prácticamente inmediata al trabajar con el entorno Eclipse, porque tiene integradas todas las herramientas necesarias para la programación con Java. Solamente hay que obtener la librería de clases *Core DEVSJAVA.jar* de la página web, y añadírsela a los proyectos que se vayan creando.

CD++ requiere la instalación del entorno *Cygwin,* que hace un poco más engorroso el proceso. Sin embargo, al instalarse como un plugin de Eclipse, trabaja con su propia perspectiva y no hay que preocuparse de añadir ninguna librería al crear los proyectos.

En el modelado y la simulación del modelo del autómata celular, se han podido comprobar las facilidades que ofrece CD++ para trabajar con este tipo de modelos. La implementación es muy sencilla, gracias al lenguaje que tiene diseñado para ello, y el modelo está contenido en un único archivo. La

aplicación con la que se ejecuta la visualización de la simulación es muy completa, y permite varias operaciones, como detenerla, controlar las iteraciones, crear una paleta de colores, ver los valores y algunas más.

En este mismo modelo, DEVSJAVA requiere la escritura de más líneas de código, aunque no son muchas. La visualización es también perfecta, pero más lenta. La herramienta con la que se realiza no ofrece tantas posibilidades, por ejemplo, no es posible detenerla y los datos que se obtienen son muy limitados. Es posible añadir colores, pero el proceso es más lento que en el caso de CD++. El espacio celular utilizado ha tenido que reducirse a 20 x 20 células, porque con un espacio celular de 40 x 40 células, como el que se ha configurado en el modelo de CD++, no termina la simulación, apareciendo un error de falta de memoria.

El mayor inconveniente en el modelo del sistema logístico de la gasolinera, ha sido no poder disponer con CD++ de una visualización del modelo, ni del paso de mensajes, lo que ha dificultado bastante la implementación. También sorprende la falta de las clases necesarias en la construcción de modelos atómicos para simulación paralela. Los datos estadísticos no se pueden obtener en la consola con CD++ durante la simulación, porque se ralentiza excesivamente. Por otra parte CD++ ofrece todas las gráficas de los puertos de salida de los modelos atómicos, e incluso la de algunos puertos de los modelos acoplados, sin necesidad de escribir ni una sola línea de código.

En comparación con CD++, el modelado del sistema logístico de la gasolinera con DEVSJAVA ha sido mucho más agradable, aunque haya habido que crear las instancias para obtener las gráficas del modelo. Se dispone de visualización de la simulación, la cual se puede efectuar paso a paso, viendo el paso de mensajes entre los modelos, y los datos aparecen en la consola durante la simulación.

A pesar de estas diferencias ambas herramientas han demostrado ser muy útiles para el modelado y la simulación de modelos DEVS.

5.3 Trabajos futuros

Un trabajo futuro podría ser la elaboración de un manual, que permita conocer la herramienta y ayude a empezar a trabajar con DEVSJAVA. En el caso de CD++ bastaría con actualizar el que ya dispone.

Los primeros pasos con ambas herramientas resultarían más fáciles, para las personas que no estuvieran familiarizadas con ellas.

Anexo A

Código DEVSJAVA del Autómata Celular

A.1 CelularCell.java

```
package celular;

import java.awt.Color;
import twoDCellSpace.*;
import GenCol.*;
import genDevs.modeling.*;
import genDevs.plots.DrawCellEntity;


public class CelularCell  extends TwoDimCell {

  protected double nivel;     //nivel de radioactividad de la célula

  protected double dif;       //coeficiente de difusión

  protected double adv;       //coeficiente de advección


  /**
   * Método Constructor:
   * @param nivel
   * @param xcoord      /* coordenadas de la célula en el espacio
   * @param ycoord         celular */
   */

  public CelularCell(double nivel, int xcoord, int ycoord) {
        super(new Pair(new Integer(xcoord), new Integer(ycoord)));

        this.nivel = nivel;

        this.dif = 0.03;        /* Valores de los coeficientes
        this.adv = 0.08;           para el modelo */
  }
```

```
  /**
   * Método Inicial:
   */

  public void initialize() {
    super.initialize();
    if(nivel == 0)
      passivate();
    else
      holdIn("active",0);     /* La célula con nivel = 1 pasa al estado
                                 active */
  }

  /**
   * Método de la Función de Transición Externa:
   */

  public void deltext(double e, message x) {

    Continue(e);          /* Este método reduce sigma en e,
                             es decir: sigma = sigma - e  */

      /* variables para almacenar los valores
         recibidos en los puertos de entrada: */
    doubleEnt      norte = new doubleEnt(0),
                   sur = new doubleEnt(0),
                   este = new doubleEnt(0),
                   oeste = new doubleEnt(0);

    for (int i = 0; i < x.getLength(); i++) {  /* Por si llega más de
                                                  un mensaje en el
                                                  mismo instante */

        if (messageOnPort(x, "inN", i))
            norte = (doubleEnt) x.getValOnPort("inN", i);

        if (messageOnPort(x, "inE", i))
            este = (doubleEnt) x.getValOnPort("inE", i);

         if (messageOnPort(x, "inS", i))
             sur = (doubleEnt) x.getValOnPort("inS", i);

         if (messageOnPort(x, "inW", i))
             oeste = (doubleEnt) x.getValOnPort("inW", i);
      }

/* Regla para la propagación del nivel de radioactividad entre las
   células: */

    nivel = nivel + dif*(norte.getv() + sur.getv() + este.getv() +
                   oeste.getv() - 4*nivel) + adv*(norte.getv() - nivel);

      holdIn("active", 0);

      /* Si la célula recibe un valor en alguno de sus puertos de
         entrada,actualiza su nivel y pasa al estado active */

  }
```

```
/**
 * Método de la Función de Transición Interna:
 */

public void deltint() {
      passivate();    // La célula pasa al estado passive
}


/**
 * Método de la Función de Salida:
 */

public message out() {

  message m = new message();

  if(nivel > 1.0E-5) {      /* Para nivel > 1.0E-6 sale un error de
                               memoria */
    doubleEnt nivelEnt = new doubleEnt(nivel);

    /* Envía por cada uno de los cuatro puertos de salida,
       el nivel a las células vecinas: */
    m.add(makeContent("outN",nivelEnt));
    m.add(makeContent("outE",nivelEnt));
    m.add(makeContent("outS",nivelEnt));
    m.add(makeContent("outW",nivelEnt));

    /* Clasificación por rangos, del valor de nivel
       que se envía por el puerto "out" a CellGridPlot.
       Según el rango, la representación será del color
       especificado: */
  if(nivel > 0.5)
    m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.black,Color.black)));
  else if(nivel > 0.1)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.darkGray,Color.darkGray)));
  else if(nivel > 0.01)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.red,Color.red)));
  else if(nivel > 0.0075)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.orange,Color.orange)));
  else if(nivel > 0.005)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.yellow,Color.yellow)));
  else if(nivel > 0.0025)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.green,Color.green)));
  else if(nivel > 0.001)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.cyan,Color.cyan)));
  else if(nivel > 5.0E-4)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.blue,Color.blue)));
  else if(nivel > 5.0E-5)
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
          x_pos, y_pos, Color.magenta,Color.magenta)));
```

```
    else if(nivel > 1.0E-6)
        m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
            x_pos, y_pos, Color.pink,Color.pink)));
    }
    else
      m.add(makeContent("out",new DrawCellEntity("drawCellToScale",
            x_pos, y_pos, Color.white,Color.white)));

    return m;
  }


 /**
  * Método para establecer el nivel de la célula:
  */

  public void setNivel(double niv) {
          this.nivel = niv;
  }


  /**
   * Método para establecer las coordenadas de la célula:
   */

  public void setcelularPosXY(double xPos, double yPos) {
          this.x_pos = xPos;
          this.y_pos = yPos;
        }

} // Fin de CelularCell
```

A.2 CelularSpace.java

```
package celular;

import twoDCellSpace.*;
import java.awt.*;
import genDevs.modeling.*;
import genDevs.plots.*;
import genDevs.simulation.coordinator;
import simView.*;

public class CelularSpace extends TwoDimCellSpace {

        /**
         * Método Constructor
         * @param xDim  //Dimensiones del espacio celular
         * @param yDim
         */
        public CelularSpace( int xDim, int yDim) {

                super("Celular", xDim, yDim);
```

```
        double nivelInicial = 1;  /* nivel de inicio en la
                                     célula afectada */
        // Creación de las celdas:
        for (int i = 0; i < xDimCellspace; i++){
              for (int j = 0; j < yDimCellspace; j++){
                //Crea las celdas con nivel = 0:
                CelularCell celc = new CelularCell(0, i, j);

              //Establece el nivel inicial en la celda central:
                if((i == xDimCellspace/2) &&
                                          (j == yDimCellspace/2))
                      celc.setNivel(nivelInicial);

                /* Sitúa las células de modo que el espacio
                   celular se visualice como un sistema de ejes
                   de coordenadas: */
                celc.setcelularPosXY(((double)i*10-
                            xDimCellspace*5), ((double)j*10 -
                            yDimCellspace*5));
                celc.setTwoDimSpaceSize(xDim, yDim);

                // Añade la célula al espacio celular:
                addCell(celc);
              }
        }

          /* Por medio del siguiente método se realizan todos
             los enlaces entre los puertos de cada célula, con
             los de sus vecinas: */
      doNeighborToNeighborCoupling();

      //Creación de CellGridPlot:
      CellGridPlot t = new CellGridPlot("celular", 1.0,"",
                                              400, "", 400);
      t.setCellGridViewLocation(500,75);
      add(t);

      /* Conecta el puerto de salida "out" de cada célula,
         con el puerto de entrada drawCellToScale de
         CellGridPlot: */
      componentIterator it1 = components.cIterator();
      while(it1.hasNext()) {
        devs d1 = (devs)it1.nextComponent();
        addCoupling(d1,"out",t,"drawCellToScale");
      }
}


/**
   * Automatically generated by the SimView program.
   * Do not edit this manually, as such changes will get
   * overwritten.
   */
  public void layoutForSimView()
  {
      preferredSize = new Dimension(164, 198);
  }
```

```
    /**
     * Método main:
     */

    public static void main (String[ ] args){
        //Crea el espacio celular:
        ViewableDigraph d = new CelularSpace(20,20);

        //Crea el coordinador DEVS:
        coordinator r = new coordinator (d);
        r.initialize();

        //Imprime el tiempo inicial:
        double initTime = System.currentTimeMillis();
        System.out.println("Initial Time: " + initTime);

        //Establece el tiempo de duración de la simulación:
        r.simulate(676);

        //Imprime el tiempo final:
        double termTime = System.currentTimeMillis();
        System.out.println("Termination Time: " + termTime);

        //Imprime el tiempo de ejecución:
        double eTime = termTime-initTime;
        System.out.println("Execution Time: " + eTime);
    }

}      //Fin de CelularSpace
```

Anexo B

Código DEVSJAVA del Sistema Logístico

B.1 Generador.java

```
package gasolinera;

import simView.*;
import genDevs.modeling.*;
import statistics.*;

public class Generador extends ViewableAtomic {
      protected double interArrivalTime; /* Parámetro para la función
                                           exponencial */
      protected int count; // Variable contador
      protected rand r;    //Variable para la función de probabilidad
      protected long seed; //Semilla para generar la función rand


      /**
       * Método Constructor:
       */

      public Generador(String name,double InterArrivalTime,long seed){
            super(name);
            this.seed = seed;
            addInport("stop");      // Puerto de entrada
            addOutport("out");      // Puerto de salida
            interArrivalTime = InterArrivalTime;
      }

      /**
       * Método inicial:
       */

      public void initialize(){
            r = new rand(seed);  /* Creación de la función rand, con la
                                    que se generará la función de
                                    probabilidad exponencial */
```

```
            holdIn("active",0); /* Al iniciar el modelo en el estado
                                  active con ta=0, se genera una entidad
                                  en el momento inicial */
            count = 1;
            super.initialize();
      }

      /**
       * Método de la Función de Transición Externa:
       */

      public void  deltext(double e,message x) {
            if (phaseIs("active")&& somethingOnPort(x,"stop"))
                  passivate();
            /* Si llega algún mensaje al puerto "stop",
               el modelo pasa al estado passive */
      }


      /**
       * Método de la Función de Transición Interna:
       */

      public void  deltint( ) {
            if(phaseIs("active")){
                  count = count +1;
                  holdIn("active",r.expon(interArrivalTime));
            /* Cada vez que se produce una transición interna,
               el contador se incrementa, y el modelo pasa
               al estado active con una transición planificada
               en el tiempo que fije la función exponencial */
            }
      }

      /**
       * Método de la Función de Salida:
       */

      public message  out( ) {
            return outputNameOnPort("coche" + count, "out");
            /* Envía por el puerto "out", un mensaje con el nombre
               "coche" seguido del valor de la variable contador */
      }
}   // Fin de Generador
```

B.2 Transd.java

```
package gasolinera;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;
```

```
public class Transd extends  ViewableAtomic{
      protected Function  arrived, solved; /*Funciones para el control
                                 de los tiempos de las entidades */
      protected double  clock,total_ta, /* Variables auxiliares para la
                             generación de los datos estadísticos */
                        observation_time; /* Parámetro del modelo, que
                             indica el tiempo que dura la simulación */

      /**
       * Método Constructor:
        * @param name
        * @param Observation_time
       */

      public Transd(String  name,double Observation_time){
            super(name);
            addInport("ariv");      // Puertos de entrada
            addInport("solved");
            addOutport("out");      // Puerto de salida
            arrived = new Function();
            solved = new Function();
            observation_time = Observation_time;
      }

      /**
       * Método inicial:
       */

      public void initialize(){
            phase = "active";
            sigma = observation_time;
            clock = 0;
            total_ta = 0;
      }

      /**
       * Método de la Función de Transición Externa:
       */
      public void  deltext(double e,message  x){
            clock = clock + e; /* Actualiza la variable clock, que
                                  indica el tiempo actual */
            Continue(e);    /* Este método reduce sigma en e,
                               es decir: sigma = sigma - e */
            entity  val;
            for(int i=0; i< x.size();i++){   /* Por si llega más de un
                                       mensaje en el mismo instante */
                  if(messageOnPort(x,"ariv",i)) {
                        val = x.getValOnPort("ariv",i);
                        arrived.put(val.getName(),newdoubleEnt(clock));
                        /* Añade a la función arrived el nombre de la
                 entidad que se acaba de recibir, y el tiempo actual */
                  }
                  if(messageOnPort(x,"solved",i)){
                        val = x.getValOnPort("solved",i);
                        /* Comprueba si la entidad recien llegada está
                           en la función arrived */
                        if(arrived.containsKey(val.getName())){
                              /* Si está, toma el tiempo de llegada que
                                 figura en arrived */
```

```
                        entity  ent=
                            (entity)arrived.assoc(val.getName());
                        doubleEnt  num = (doubleEnt)ent;
                        double arrival_time = num.getv();
                        /* y se lo resta al tiempo actual para
                           obtener el tiempo que la entidad ha
                           permanecido en el sistema */
                        double turn_around_time = clock -
                                                arrival_time;
                        /* Actualiza la variable total_ta, que
                           almacena la suma del tiempo de todas
                           las entidades */
                        total_ta = total_ta + turn_around_time;
                        /* Añade la entidad a la función solved,
                           junto con el tiempo actual */
                        solved.put(val, new doubleEnt(clock));
                  }
            }
      }
      show_state(); /* Al invocar a este método aqui, se consigue
                       que se ejecute cada vez que una entidad
                       entra o sale del sistema */
}

/**
 * Método de la Función de Transición Interna:
 */

public void  deltint(){
      /* La transición interna ocurre cuando termina el tiempo
         marcado por el parámetro observation_time, el modelo
         pasa al estado passive */
      passivate();
}

/**
 * Método de la Función de Salida:
 */

public  message out( ){
      message  m = new message();
      //Envía un mensaje por el puerto "out", con el valor 1
      content  con = makeContent("out",new doubleEnt(1));
      m.add(con);
      return m;
}

/**
 * Método para calcular el valor del tiempo de media que han
 * estado las entidades en el sistema:
 */

public double compute_TA(){
      double avg_ta_time = 0;
      if(!solved.isEmpty())
            avg_ta_time = ((double)total_ta)/solved.size();
      return avg_ta_time;
}
```

```
/**
 * Método para calcular el número medio de entidades procesadas
 * por unidad de tiempo:
 */

public double compute_Thru(){
      double thruput = 0;
      if(clock > 0)
            thruput = solved.size()/(double)clock;
      return thruput;
}

/**
 * Método para imprimir los datos estadísticos:
 */

public void  show_state(){
      if(sigma == INFINITY)  sigma = 0;   /* Evita que sigma haya
                              pasado a valer INFINITY debido al método
                              passivate() de la función de transición
                              interna */
      //Imprime los datos estadísticos
      System.out.println("sigma : " +  sigma  +  " minutos.");
      if (arrived != null && solved != null){
            System.out.println("Total coches llegados = " +
                              arrived.size() + " coches.");
            System.out.println("Total coches terminados = " +
                              solved.size() + " coches.");
            System.out.println("Tiempo medio de permanencia = "
                              +  compute_TA() + " minutos.");
            System.out.println();
      }
}

/**
 * Método getTooltipText() que muestra datos del modelo
 * al situar el cursor del ratón sobre él en la visualización
 */

public String getTooltipText(){
      String  s = "";
      if (arrived != null && solved != null){
            s = "\n"+"coches llegados :"   +  arrived.size()
            +"\n"+"coches terminados :"  +  solved.size()
            +"\n" + "AVG TA = "   +  compute_TA()
            +"\n" + "THRUPUT = "   +  compute_Thru();
      }
      return super.getTooltipText()+s;
}
} //Fin de Transd
```

B.3 Ef.java

```
package gasolinera;

import simView.*;
import java.awt.*;
public class Ef extends ViewableDigraph{

      /**
       * Método Constructor:
       * @param nm
       * @param int_arr_t    // Parámetro para la creación de Generador
       * @param observe_t    // Parámetro para la creación de Transd
       */

      public Ef(String nm,double int_arr_t,double observe_t){
            super(nm);
            addInport("in");        // Puerto de entrada
            addOutport("out");      // Puerto de salida

            // Creación de Generador y Transd
            ViewableAtomic g = new Generador("generador",int_arr_t,11);
            ViewableAtomic t = new Transd("transd",observe_t);
            // Se añaden al modelo
            add(g);
            add(t);

            initialize();
            /* Generador se visualizará con el color orange,
               Transd se deja con el color por defecto, que es gris */
            Color color;
            color = Color.orange;
            g.setBackgroundColor(color);

            // Se realizan los enlaces entre los puertos
            addCoupling(g,"out",t,"ariv");
            addCoupling(this,"in",t,"solved");
            addCoupling(t,"out",g,"stop");
            addCoupling(g,"out",this,"out");
      }


    /**
     * Automatically generated by the SimView program.
     * Do not edit this manually, as such changes will get
     * overwritten.
     */
    public void layoutForSimView()
    {
      preferredSize = new Dimension(278, 140);
      ((ViewableComponent)withName("transd")).setPreferredLocation
                                    (new Point(-5, 80));

      ((ViewableComponent)withName("generador")).setPreferredLocation
                                    (new Point(5, 18));
    }
} // Fin de Ef
```

B.4 Entrada.java

```
package gasolinera;

import simView.*;
import statistics.rand;
import genDevs.modeling.*;
import GenCol.*;

public class Entrada extends  ViewableAtomic {
    protected entity job; /* Variable para almacenar temporalmente las
                             entidades */

    protected rand r1, r2, r3, r4; /* Variables para las funciones de
                                      probabilidad */

    protected double q1,q2,q3;      /* Variables para los valores de
                                       las colas de los surtidores */

   /**
    * Método Constructor:
    * @param name
    */

    public Entrada(String  name) {
      super(name);
      addInport("in");         // Puertos de entrada
      addInport("inQ1");
      addInport("inQ2");
      addInport("inQ3");
      addOutport("out1");      // Puertos de salida
      addOutport("out2");
      addOutport("out3");
    }

    /**
     * Método inicial:
     */

    public void initialize(){
      phase = "passive"; // Se inicia el modelo en el estado passive
      sigma = INFINITY;

      job = new entity("job");
      q1 = q2 = q3 = 0;

      /* Se crean las funciones de probabilidad,
      cada una con una semilla diferente */
      r1 = new rand(11);
      r2 = new rand(131);
      r3 = new rand(337);
      r4 = new rand(991);

      super.initialize();

    }
```

```
/**
 * Método de la Función de Transición Externa:
 */

public void  deltext(double e,message x) {
  Continue(e);       /* Este método reduce sigma en e,
                        es decir: sigma = sigma - e  */
  if (phaseIs("passive")) {
        for (int i=0; i< x.getLength();i++) /* Por si llega más de
                                               un mensaje en el mismo
                                         instante */
              if (messageOnPort(x,"in",i)) {
                    /* Si se recibe una entidad en el puerto "in",
                       el modelo pasa al estado active */
                    job = x.getValOnPort("in",i);
                    holdIn("active",0);
              }
        for (int i=0; i< x.getLength();i++)
              if (messageOnPort(x,"inQ1",i)) {
                    /* Si se recibe un mensaje en el puerto "inQ1",
                        se actualiza el valor de q1 */
               q1 = ((doubleEnt)(x.getValOnPort("inQ1",i))).getv();
              }
        for (int i=0; i< x.getLength();i++)
              if (messageOnPort(x,"inQ2",i)) {
               q2 = ((doubleEnt)(x.getValOnPort("inQ2",i))).getv();
              }
        for (int i=0; i< x.getLength();i++)
              if (messageOnPort(x,"inQ3",i)) {
               q3 = ((doubleEnt)(x.getValOnPort("inQ3",i))).getv();
              }
  }
}

/**
 * Método de la Función de Transición Interna:
 */

public void  deltint( ) {
  passivate();
}

/**
 * Método de la Función de Salida:
 */

public message out( ) {
  message   m = new message();
  content   con;
  int n;
  if((q1<q2)&(q1<q3)) {   /* q1 es la cola más corta, y la entidad
                             se envía por el puerto "out1" */
        con = makeContent("out1",job);
  }
  else if((q2<q1)&(q2<q3)) {    /* q2 es la cola más corta, y la
                            entidad se envía por el puerto "out2" */
        con = makeContent("out2",job);
  }
```

```
else if((q3<q1)&(q3<q2)) {    // q3 es la cola más corta, y la
                        entidad se envía por el puerto "out3" */
      con = makeContent("out3",job);
}

else if((q1==q2)&(q1<q3)) {   /* q1 y q2 son iguales y las más
                              cortas */
      n = r1.iuniform(1);    // genera un número aleatorio: 0 ó 1
      if (n % 2 == 0){
            con = makeContent("out2",job);
      }
      else  {
            con = makeContent("out1",job);
      }
}
else if((q1==q3)&(q1<q2)) {   /* q1 y q3 son iguales y las más
                              cortas */
      n = r2.iuniform(1);    // genera un número aleatorio: 0 ó 1
      if (n % 2 == 0){
            con = makeContent("out3",job);
      }
      else  {
            con = makeContent("out1",job);
      }
}
else if((q2==q3)&(q2<q1)) {   /* q2 y q3 son iguales y las más
                              cortas */
      n = r3.iuniform(1);    /* genera un número aleatorio: 0 ó 1
      if (n % 2 == 0){
            con = makeContent("out3",job);
      }
      else  {
            con = makeContent("out2",job);
      }
}

else if((q1==q2)&(q1==q3)){   // Las tres colas son iguales
      n = r4.iuniform(2); // genera un número aleatorio: 0, 1 ó 2
      if (n % 3 == 0){
            con = makeContent("out3",job);
      }
      else if(n % 3 == 1) {
            con = makeContent("out1",job);
      }
      else  {
            con = makeContent("out2",job);
      }
}

else {
      con = null;
      System.out.println("ERROR en entrada con las colas de los
                              surtidores");
}
m.add(con);
return m;
}
```

```
    /**
     * Método getTooltipText() que muestra datos del modelo
     * al situar el cursor del ratón sobre él en la visualización
     */

    public String getTooltipText(){
         return
         super.getTooltipText()
         +"\n"+"q1: " + q1
         +"\n"+"q2: " + q2
         +"\n"+"q3: " + q3  ;
        }
} // Fin de Entrada
```

B.5 Surtidor.java

```
package gasolinera;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;
import statistics.*;

public class Surtidor extends ViewableAtomic{
      protected entity job;   /* Variable para almacenar temporalmente
                                 las entidades */
      protected Queue q;       // Variable para gestionar la cola
      protected double tcola;  /* Variable para almacenar el tamaño de
                                  la cola */
      protected double resto; /* Variable auxiliar para almacenar el
                                 tiempo restante, cuando se interrumpe
                                 el proceso */
      protected rand r;      // Variable para la función de probabilidad
      protected long seed;    // Semilla para generar la función rand

      /**
       * Método constructor:
       * @param name
       */

      public Surtidor(String name) {
            super(name);
            addInport("in");        // Puertos de entrada
            addInport("inEnd");
            addOutport("out");      // Puertos de salida
            addOutport("outQ");
            addOutport("transd");

            /* Se le asigna un valor distinto a cada una de las
               Semillas para generar las funciones de probabilidad en
               cada surtidor. Todos son números primos */
            if(name.equals("surtidor1")) {
                  seed = 11;
            }
```

```
        else if(name.equals("surtidor2")) {
              seed = 141;
        }
        else if(name.equals("surtidor3")) {
              seed = 2039;
        }
        // Se crea la función de probabilidad
        r = new rand(seed);
}

/**
 * Método inicial:
 */

public void initialize(){
        phase = "passive";      // Se inicia el modelo en el estado
        sigma = INFINITY;       // passive
        job = new entity("job");
        q = new Queue();
        super.initialize();
}

/**
 * Método para calcular el tamaño de la cola:
 * @return
 */

public int tamCola() {
        return q.size();
}


/**
 * Método de la Función de Transición Externa:
 */

public void  deltext(double e, message   x){
        Continue(e);      /* Este método reduce sigma en e,
                            es decir: sigma = sigma - e  */
        if (phaseIs("passive")){
              for (int i=0; i< x.size(); i++)  /* Por si llega más
                                                  de un mensaje en
                                                el mismo instante */
                    if (messageOnPort(x, "in", i)){
                          /* Si llega una entidad al puerto "in",
                             la mete en la cola */
                          q.add(x.getValOnPort("in", i));
                    }
              /* Coge a la primera de la cola. De este modo se
                 asegura de que job es la primera entidad en la
                 cola */
              job = (entity)q.first();
              /* Calcula el tiempo de proceso, y lo guarda en la
                 variable resto */
              resto = r.uniform(2,5);
              /* Planifica una transición interna para este mismo
              instante, para que se active la función de salida */
              holdIn("sendOutQ", 0);
        }
```

```
        else if (phaseIs("busy")){
              for (int i=0; i< x.size();i++)
                    if (messageOnPort(x, "in", i))       {
                          /* Si llega una entidad al puerto "in",
                             la mete en la cola */
                          entity jb = x.getValOnPort("in", i);
                          q.add(jb);
                          /* Almacena en la variable resto el
                             tiempo que falta para terminar con la
                             entidad que está siendo procesada */
                          resto = sigma;
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la función de salida */
                          holdIn("sendQ", 0);
                    }
        }
        else if (phaseIs("wait"))      {
              for (int i=0; i< x.size();i++)
                    if (messageOnPort(x, "inEnd", i)) {
                          /* Elimina a la primera de la cola,
                            que es la última que se ha procesado */
                          q.remove();
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la función de salida */
                          holdIn("end", 0);
                    }
              for (int i=0; i< x.size();i++)
                    if (messageOnPort(x, "in", i))       {
                          /* Si llega una entidad al puerto "in",
                            la mete en la cola */
                          entity jb = x.getValOnPort("in", i);
                          q.add(jb);
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la función de salida */
                          holdIn("sendQ2",0);
                    }
        }
        else if(phaseIs("end")) {  //Es difícil que ocurra, pero...
              for (int i=0; i< x.size();i++)
                    if (messageOnPort(x, "in", i))       {
                          /* Si llega una entidad al puerto "in",
                             la mete en la cola */
                          entity jb = x.getValOnPort("in", i);
                          q.add(jb);
                    }
        }
  }

  /**
   * Método de la Función de Transición Interna:
   */
  public void  deltint( ) {
        if (phaseIs("sendQ")) {
        // Pasa al estado busy, con el tiempo de la variable resto
              holdIn("busy",resto);
        }
```

```
        else if(phaseIs("busy")) {
              holdIn("wait",INFINITY);
        }
        else if(phaseIs("sendQ2")) {
              holdIn("wait",INFINITY);
        }
        else if(phaseIs("end")) {
              if(!q.isEmpty()){        /* Si la cola no está vacía,
                          coge a la primera entidad de la cola */
                    job = (entity)q.first();
                    /* Calcula el tiempo de proceso, y lo guarda en
                       la variable resto */
                    resto = r.uniform(2,5);
                    /* Planifica una transición interna para este
                       mismo instante, para que se active la
                       función de salida */
                    holdIn("sendOutQ2", 0);
              }
              else passivate();
        }
        else if(phaseIs("sendOutQ")) {
              /* Planifica una transición interna para este mismo
               instante, para que se active la función de salida */
              holdIn("sendQ", 0);
        }
        else if(phaseIs("sendOutQ2")) {
              /* Pasa al estado busy, con el tiempo de la variable
                 resto */
              holdIn("busy",resto);
        }
}

/**
 * Método de la Función de Salida:
 */

public message out() {
        message   m = new message();
        if ((phaseIs("sendQ")) || (phaseIs("sendQ2")) ||
                                         (phaseIs("end"))) {
              /* Envía por "outQ" hacia entrada, el tamaño de la
                 cola  */
              tcola = tamCola();
              doubleEnt dEnt = new doubleEnt(tcola);
              m.add(makeContent("outQ",dEnt));
        }
        if (phaseIs("busy")) {
              /* Envía por "out" hacia colaCajas, la entidad que
                 acaba de procesar,añadiéndole el nombre del
                 surtidor */
              String stJob = job.toString() + name;
              entity newJob = new entity(stJob);
              m.add(makeContent("out",newJob));
        }
        if((phaseIs("sendOutQ")) || (phaseIs("sendOutQ2"))) {
              /* Envía por "transd" hacia transdSurt, el valor de
                 la entidad */
              m.add(makeContent("transd",job));
        }
```

```
            return m;
        }

        /**
         * Método getTooltipText() que muestra datos del modelo
         * al situar el cursor del ratón sobre él en la visualización
         */

        public String getTooltipText(){
            return
            super.getTooltipText()
            +"\n"+"tamaño de la cola: " + q.size()
            +"\n"+"elementos en la cola: " + q.toString();
        }
} // Fin de Surtidor
```

B.6 TransdSurt.java

```
package gasolinera;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;

public class TransdSurt extends  ViewableAtomic {
        protected Function  arrived, solved1, solved2, solved3;
       /* Funciones para el control de los tiempos de las entidades */
        protected double observation_time; /* Parámetro del modelo, que
                                             indica el tiempo que dura
                                             la simulación */
        /* Variables auxiliares para la generación
          de los datos estadísticos: */
        protected double clock;
        protected double total_ta_1, total_ta_2, total_ta_3;
        protected double tmax_1, tmax_2, tmax_3;
        protected double noCola_1, noCola_2, noCola_3;
        protected double espera_1, espera_2, espera_3;

        /**
         * Método Constructor:
         * @param name
         * @param Observation_time
         */

        public TransdSurt(String  name,double Observation_time){
              super(name);
              addInport("ariv");     // Puertos de entrada
              addInport("solved1");
              addInport("solved2");
              addInport("solved3");
              arrived = new Function();
              solved1 = new Function();
              solved2= new Function();
              solved3 = new Function();
```

```
        observation_time = Observation_time;
}

/**
 * Método inicial:
 */

public void initialize() {
        phase = "active";
        sigma = observation_time;
        clock = 0;
        total_ta_1 = total_ta_2 = total_ta_3 = 0;
        tmax_1 = tmax_2 = tmax_3 = 0;
        noCola_1 = noCola_2 = noCola_3 = 0;
        espera_1 = espera_2 = espera_3 = 0;
        arrived = new Function();
        solved1 = new Function();
        solved2 = new Function();
        solved3 = new Function();
}

/**
 * Método de la Función de Transición Externa:
 */

public void  deltext(double e,message  x){
        clock = clock + e; /* Actualiza la variable clock, que
                              indica el tiempo actual */
        Continue(e);      /* Este método reduce sigma en e,
                             es decir: sigma = sigma - e */
        entity  val;
        for(int i=0; i< x.size();i++){     /* Por si llega más de
                                             un mensaje en el mismo
                                             instante */
                if(messageOnPort(x,"ariv",i)){
                        val = x.getValOnPort("ariv",i);
                        arrived.put(val.getName(),
                                               newdoubleEnt(clock));
                        /* Añade a la función arrived el nombre de la
                           entidad que se acaba de recibir, y el
                           tiempo actual */
                }
                if(messageOnPort(x,"solved1",i)){
                        val = x.getValOnPort("solved1",i);
                       /* Comprueba si la entidad recien llegada está
                          en la función arrived */
                        if(arrived.containsKey(val.getName())){
                             /* Si está, toma el tiempo de llegada que
                                figura en arrived */
                              entity  ent = (entity)arrived.
                                               assoc(val.getName());
                              doubleEnt  num = (doubleEnt)ent;
                              double arrival_time = num.getv();
                              /* y se lo resta al tiempo actual para
                                 obtener el tiempo que la entidad ha
                               permanecido en la cola del surtidor1 */
                              double turn_around_time = clock -
                                                          arrival_time;
```

```
                        /* Actualiza la variable total_ta_1, que
                           almacena la suma del tiempo de todas
                           las entidades que han pasado por la
                           cola del surtidor1 */
                        total_ta_1 = total_ta_1 +
                                             turn_around_time;
                       /* Añade la entidad a la función solved1,
                          junto con el tiempo actual */
                        solved1.put(val, new doubleEnt(clock));
                        // Actualiza las variables auxiliares
                        if(turn_around_time > tmax_1) {
                             /* Variable que almacena el tiempo
                             máximo de espera en la cola del
                             surtidor1 */
                              tmax_1 = turn_around_time;
                        }
                        if(turn_around_time == 0) {
                             /* Variable que almacena el número
                             de entidades que no han esperado en
                             la cola del surtidor1 */
                              noCola_1 = noCola_1 + 1;
                        }
                        if(turn_around_time > 3) {
                             /* Variable que almacena el número
                             de entidades que han esperado en la
                             cola del surtidor1 más de tres
                             minutos */
                              espera_1 = espera_1 + 1;
                        }
                  }
            }
            if(messageOnPort(x,"solved2",i)){//Similar a solved1
                  val = x.getValOnPort("solved2",i);
                  if(arrived.containsKey(val.getName())){
                        entity  ent = (entity)arrived.
                                         assoc(val.getName());
                        doubleEnt  num = (doubleEnt)ent;
                        double arrival_time = num.getv();
                        double turn_around_time = clock -
                                             arrival_time;
                        total_ta_2 = total_ta_2 +
                                             turn_around_time;
                        solved2.put(val, new doubleEnt(clock));
                        if(turn_around_time > tmax_2) {
                              tmax_2 = turn_around_time;
                        }
                        if(turn_around_time == 0) {
                              noCola_2 = noCola_2 + 1;
                        }
                        if(turn_around_time > 3) {
                              espera_2 = espera_2 + 1;
                        }
                  }
            }
            if(messageOnPort(x,"solved3",i)){//Similar a solved1
                  val = x.getValOnPort("solved3",i);
                  if(arrived.containsKey(val.getName())){
                        entity  ent = (entity)arrived.
                                         assoc(val.getName());
```

```
                        doubleEnt  num = (doubleEnt)ent;
                        double arrival_time = num.getv();
                        double turn_around_time = clock -
                                                arrival_time;
                       total_ta_3 = total_ta_3 +
                                                turn_around_time;
                        solved3.put(val, new doubleEnt(clock));
                        if(turn_around_time > tmax_3) {
                              tmax_3 = turn_around_time;
                        }
                        if(turn_around_time == 0) {
                              noCola_3 = noCola_3 + 1;
                        }
                        if(turn_around_time > 3) {
                              espera_3 = espera_3 + 1;
                        }
                  }
            }
      }
}

/**
 * Método de la Función de Transición Interna:
 */

public void  deltint(){
      /* La transición interna ocurre cuando termina el tiempo
        marcado por el parámetro observation_time, el modelo
        pasa al estado passive */
      passivate();
      /* Al final de la simulación, se mostrarán los datos
         estadísticos de los tres surtidores */
      show_state_1();
      show_state_2();
      show_state_3();
}

/**
 * Método para calcular el valor del tiempo de media que han
 * estado en la cola, todas las entidades que han pasado por el
 * surtidor1:
 */

public double compute_TA_1(){
      double avg_ta_time = 0;
      if(!solved1.isEmpty())
            avg_ta_time = ((double)total_ta_1)/solved1.size();
      return avg_ta_time;
}

/**
 * Método para calcular el valor del tiempo de media que han
 * estado en la cola, sólo las entidades que han esperado en la
 * cola del surtidor1:
 */

public double compute_TA_1_1(){
       double avg_ta_time = 0;
```

```
        if(!solved1.isEmpty())
              avg_ta_time = ((double)total_ta_1)/(solved1.size()
                                                  - noCola_1);
        return avg_ta_time;
}

 /**
  * Método para calcular el valor del tiempo de media que han
  * estado en la cola, todas las entidades que han pasado por el
  * surtidor2:
  */
       public double compute_TA_2(){
       double avg_ta_time = 0;
       if(!solved2.isEmpty())
             avg_ta_time = ((double)total_ta_2)/solved2.size();
       return avg_ta_time;
 }

 /**
  * Método para calcular el valor del tiempo de media que han
  * estado en la cola, sólo las entidades que han esperado en la
  * cola del surtidor2:
  */

 public double compute_TA_2_2(){
        double avg_ta_time = 0;
        if(!solved2.isEmpty())
              avg_ta_time = ((double)total_ta_2)/(solved2.size()
                                                  - noCola_2);
        return avg_ta_time;
}

 /**
  * Método para calcular el valor del tiempo de media que han
  * estado en la cola, todas las entidades que han pasado por el
  * surtidor3:
  */

 public double compute_TA_3(){
       double avg_ta_time = 0;
       if(!solved3.isEmpty())
             avg_ta_time = ((double)total_ta_3)/solved3.size();
       return avg_ta_time;
 }

 /**
  * Método para calcular el valor del tiempo de media que han
  * estado en la cola, sólo las entidades que han esperado en la
  * cola del surtidor3:
  */

 public double compute_TA_3_3(){
        double avg_ta_time = 0;
        if(!solved3.isEmpty())
              avg_ta_time = ((double)total_ta_3)/(solved3.size()
                                                  - noCola_3);
        return avg_ta_time;
}
```

```
/**
 * Método para truncar los decimales de un número:
 * @param numero
 * @param decimales
 * @return
 */

public double redondear(double numero, int decimales) {
      return Math.round(numero*Math.pow(10, decimales)) /
                                    Math.pow(10, decimales);
}
/**
 * Método para imprimir los datos estadísticos
 * del surtidor1:
 */

public void  show_state_1() {
      if (arrived != null && solved1 != null) {
            System.out.println("Total coches que han pasado por
            el Surtidor 1 = " + solved1.size() + " coches.");

            System.out.println("Total coches que no han esperado
            en la cola = " + noCola_1 + " coches. (" +
            redondear((noCola_1/solved1.size())*100, 2) + "%).");

            System.out.println("Total coches que han esperado
            más de tres minutos = " + espera_1 + " coches. (" +
            redondear((espera_1/solved1.size())*100, 2) + "%).");

            System.out.println("Tiempo máximo de espera en la
            cola del Surtidor 1 = "  + redondear(tmax_1, 2) + "
            minutos.");

            System.out.println("Tiempo medio de espera total en
            la cola del Surtidor 1 = "  +
            redondear(compute_TA_1(), 2) + " minutos.");

            System.out.println("Tiempo medio de espera de los
            que han estado en la cola = "  +
            redondear(compute_TA_1_1(), 2) + " minutos.");

            System.out.println();
      }
}

/**
 * Método para imprimir los datos estadísticos
 * del surtidor2:
 */

public void  show_state_2() {
      if (arrived != null && solved2 != null) {
            System.out.println("Total coches que han pasado por
            el Surtidor 2 = " + solved2.size() + " coches.");

            System.out.println("Total coches que no han esperado
            en la cola = " + noCola_2 + " coches. (" +
            redondear((noCola_2/solved2.size())*100, 2) + "%).");
```

```
                System.out.println("Total coches que han esperado
                más de tres minutos = " + espera_2 + " coches. (" +
                redondear((espera_2/solved2.size())*100, 2) + "%).");

                System.out.println("Tiempo máximo de espera en la
                cola del Surtidor 2 = "  + redondear(tmax_2, 2) + "
                minutos.");

                System.out.println("Tiempo medio de espera total en
                la cola del Surtidor 2 = "  +
                redondear(compute_TA_2(), 2) + " minutos.");
                System.out.println("Tiempo medio de espera de los
                que han estado en la cola = "  +
                redondear(compute_TA_2_2(), 2) + " minutos.");

                System.out.println();
        }
}

/**
 * Método para imprimir los datos estadísticos
 * del surtidor3:
 */

public void  show_state_3() {
        if (arrived != null && solved3 != null) {
                System.out.println("Total coches que han pasado por
                el Surtidor 3 = " + solved3.size() + " coches.");

                System.out.println("Total coches que no han esperado
                en la cola = " + noCola_3 + " coches. (" +
                redondear((noCola_3/solved3.size())*100, 2) + "%).");

                System.out.println("Total coches que han esperado
                más de tres minutos = " + espera_3 + " coches. (" +
                redondear((espera_3/solved3.size())*100, 2) + "%).");

                System.out.println("Tiempo máximo de espera en la
                cola del Surtidor 3 = "  + redondear(tmax_3, 2) + "
                minutos.");

                System.out.println("Tiempo medio de espera total en
                la cola del Surtidor 3 = "  +
                redondear(compute_TA_3(), 2) + " minutos.");

                System.out.println("Tiempo medio de espera de los
                que han estado en la cola = "  +
                redondear(compute_TA_3_3(), 2) + " minutos.");

                System.out.println();
        }
}

 /**
  * Método getTooltipText() que muestra datos del modelo
  * al situar el cursor del ratón sobre él en la
  * visualización
  */
```

```
        public String getTooltipText(){
              String  s = "";
              if (arrived != null && solved1 != null && solved2 != null
                                                  && solved2 != null){
                    s = "\n"+"coches por surtidor1 :"  +
                                                  solved1.size()
                    + "\n"+"coches por surtidor2 :"  +  solved2.size()
                    + "\n"+"coches por surtidor3 :"  +  solved3.size()
                    +"\n" + "AVG TA cola Surtidor 1 = "  +
                                                  compute_TA_1()

                    +"\n" + "AVG TA cola Surtidor 2 = "  +
                                                  compute_TA_2()
                    +"\n" + "AVG TA cola Surtidor 3 = "  +
                                                  compute_TA_3();
              }
              return super.getTooltipText()+s;
        }
} // Fin de TransdSurt
```

B.7 ColaCajas.java

```
package gasolinera;

import simView.*;

import genDevs.modeling.*;
import GenCol.*;
import statistics.*;

public class ColaCajas extends ViewableAtomic {
      protected entity job;   /* Variable para almacenar temporalmente
                                    las entidades */
      protected Queue q;      // Variable para gestionar la cola
      protected double tcola; /* Variable para almacenar el tamaño de
                                    la cola */
      protected rand r;       /* Variable para la función de
                                    probabilidad */
    protected boolean pasivo1,pasivo2; /* Variables para guardar el
                                          estado de las cajas */

      /**
       * Método Constructor:
       * @param name
       */

      public ColaCajas(String  name) {
            super(name);
            addInport("in");         // Puertos de entrada
            addInport("estado1");
            addInport("estado2");
            addOutport("out1");      // Puertos de salida
            addOutport("out2");
            addOutport("outQ");
```

```
        // Se crea la función de probabilidad
        r = new rand(1);
}

/**
 * Método inicial:
 */

public void initialize(){
        phase = "passive";      // Se inicia el modelo en el estado
        sigma = INFINITY;       // passive
        job = new entity("job");
        q = new Queue();
        pasivo1 = true;
        pasivo2 = true;
//      pasivo2 = false; // activando esta línea se anula la caja2
        super.initialize();
}

/**
 * Método para calcular el tamaño de la cola:
 * @return
 */

public int tamCola() {
        return q.size();
}

/**
 * Método de la Función de Transición Externa:
 */

public void  deltext(double e, message   x){
        Continue(e);      /* Este método reduce sigma en e,
                             es decir: sigma = sigma - e  */
        if (phaseIs("passive")){
              for (int i=0; i< x.size(); i++) /* Por si llega más
                                                  de un mensaje en
                                                el mismo instante */
                    if (messageOnPort(x, "in", i)){
                          /* Si llega una entidad al puerto "in",
                             la mete en la cola */
                          q.add(x.getValOnPort("in", i));
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la función de salida */
                          holdIn("sendQ", 0);
                    }
                    else if (messageOnPort(x, "estado1", i)) {
                          /* Si llega una entidad al puerto
                             "estado1" actualiza la variable
                             pasivo1 */
                          pasivo1=true;
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la  función de salida */
                          holdIn("sendQ2",0);
                    }
```

```
                    else if (messageOnPort(x, "estado2", i)) {
                          pasivo2=true;
                          holdIn("sendQ2",0);
                    }
      }

      else if  (phaseIs("wait")) {
         for (int i=0; i< x.size();i++)
                    if (messageOnPort(x, "in", i)) {
                          /* Si llega una entidad al puerto "in",
                             la mete en la cola */
                          q.add(x.getValOnPort("in", i));
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la función de salida */
                          holdIn("sendQ", 0);
                    }
                    else if (messageOnPort(x, "estado1", i)) {
                          /* Si llega una entidad al puerto
                             "estado1" actualiza la variable
                             pasivo1 */
                          pasivo1=true;
                          /* Planifica una transición interna para
                             este mismo instante, para que se
                             active la función de salida */
                          holdIn("sendQ",0);
                    }
                    else if (messageOnPort(x, "estado2", i)) {
                          pasivo2=true;
                          holdIn("sendQ",0);
                    }
      }

      else if(phaseIs("busy")) {
            for (int i=0; i< x.size();i++)
                    if (messageOnPort(x, "in", i))      {
                          /* Si llega una entidad al puerto "in",
                             la mete en la cola */
                       entity jb = x.getValOnPort("in", i);
                       q.add(jb);
                    }
                    else if (messageOnPort(x, "estado1", i)) {
                          /* Si llega una entidad al puerto
                             "estado1" actualiza la variable
                             pasivo1 */
                          pasivo1=true;
                    }
                    else if (messageOnPort(x, "estado2", i)) {
                          pasivo2=true;
                    }
      }
}

/**
 * Método de la Función de Transición Interna:
 */

public void  deltint( ){
```

```
        if (phaseIs("sendQ")) {
            if((pasivo1==true)||(pasivo2==true)) {
                /* Si hay una caja libre, coge la primera
                   entidad de la cola */
                job = (entity)q.first();
                // La elimina de la cola
                q.remove();
                /* Planifica una transición interna para
                   este mismo instante, para que se active la
                   función de salida */
                holdIn("busy", 0);
            }

            else {
            // Si no hay ninguna caja libre, pasa al estado wait
                holdIn("wait",INFINITY);
            }
        }
        else if(phaseIs("sendQ2")) {
            passivate();
        }
        else if(phaseIs("busy")) {
            if(!q.isEmpty()) {
                if((pasivo1==true)||(pasivo2==true)) {
                    /* Si la cola no está vacía y hay una
                       caja libre, provoca una transición
                       interna en este instante, para que se
                       active la función de salida */
                    holdIn("sendQ",0);
                }
                else {
                    /* Si no hay ninguna caja libre pasa al
                       estado wait */
                    holdIn("wait",INFINITY);
                }
            }

            // Si la cola está vacía pasa al estado passivate
            else passivate();
        }
}

/**
 * Método de la Función de Salida:
 */

public message    out() {
        message   m = new message();
        if((phaseIs("sendQ")) || (phaseIs("sendQ2"))) {
            tcola = tamCola();
            /* Si hay alguna caja ocupada, aumenta el tamaño de
               la cola, porque el valor que envía es el del
               número de entidades en el conjunto de SetCajas */
            if(pasivo1==false) tcola=tcola+1;
            if(pasivo2==false) tcola=tcola+1;
            doubleEnt dEnt = new doubleEnt(tcola);
            m.add(makeContent("outQ",dEnt));
        }
```

```
        if (phaseIs("busy")) {
            if((pasivo1==true)&&(pasivo2==false)) {
                /* Si sólo está libre la caja1, le envía la
                   entidad y actualiza la variable pasivo1 */
                m.add(makeContent("out1",job));
                pasivo1 = false;
            }
            else if((pasivo1==false)&&(pasivo2==true)) {
                m.add(makeContent("out2",job));
                pasivo2 = false;
            }
            else if((pasivo1==true)&&(pasivo2==true)) {
                /* Si las dos cajas están libres,
                   genera un número aleatorio: 0 ó 1 */
                double n = r.iuniform(1);
                if (n % 2 == 0){
                    m.add(makeContent("out2",job));
                    pasivo2 = false;
                }
                else  {
                    m.add(makeContent("out1",job));
                    pasivo1 = false;
                }
            }
        }
        return m;
    }

    /**
     * Método getTooltipText() que muestra datos del modelo
     * al situar el cursor del ratón sobre él en la visualización
     */

    public String getTooltipText(){
       return
       super.getTooltipText()
       +"\n"+"Caja 1 pasivo: " + pasivo1
       +"\n"+"Caja 2 pasivo: " + pasivo2
       +"\n"+"tamaño de la cola: " + q.size()
        +"\n"+"elementos en la cola: " + q.toString();
    }

} // Fin de ColaCajas
```

B.8 CajaSimp.java

```
package gasolinera;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;
import statistics.*;
```

```
public class CajaSimp extends ViewableAtomic {
      protected entity job;   /* Variable para almacenar temporalmente
                                 las entidades */
      protected rand r;      // Variable para la función de probabilidad
      protected long seed;    // Semilla para generar la función rand

      /**
       * Método constructor:
       * @param name
       */

      public CajaSimp(String  name) {
            super(name);
            addInport("in");         // Puerto de entrada
            addOutport("out");       // Puertos de salida
            addOutport("out1");
            addOutport("out2");
            addOutport("out3");
            addOutport("estado");
            /* Se le asigna un valor distinto a cada una de las
            semillas para generar las funciones de probabilidad en cada
            caja.Todos son números primos */
            if(name.equals("Caja 1")) {
                  seed = 457;
            }
            else if(name.equals("Caja 2")) {
                  seed = 883;
            }
            // Se crea la función de probabilidad
            r = new rand(seed);
      }

      /**
       * Método inicial:
       */

      public void initialize(){
            phase = "passive";       // Se inicia el modelo en el estado
            sigma = INFINITY;        // passive
            job = new entity("job");
            super.initialize();
      }

      /**
     * Método de la Función de Transición Externa:
     */

      public void  deltext(double e, message x){
            Continue(e);       /* Este método reduce sigma en e,
                                  es decir: sigma = sigma - e  */
            if (phaseIs("passive")) {
             for (int i=0; i< x.getLength();i++) /* Por si llega más de
                                                    un mensaje en el
                                                    mismo instante */
              if (messageOnPort(x,"in",i)) {
                  /* Si llega una entidad al puerto "in",
                     empieza a procesarla */
                  job = x.getValOnPort("in",i);
                  holdIn("busy",r.uniform(0.5,2));
```

```
            }
        }
    }

    /**
     * Método de la Función de Transición Interna:
     */

    public void  deltint( ){
          passivate();
    }

  /**
   * Método de la Función de Salida:
   */
          public message    out() {
          message    m = new message();
          entity pasivo = new entity("pasivo");
          if (phaseIs("busy")) {
                String stJob = job.toString();
                /* Comprueba cuál es el nombre del surtidor
                   que está añadido a la entidad */
                if(stJob.endsWith("surtidor1")) { // Si es surtidor1
                      // Lo elimina de la entidad
                      entity job1 = new entity
                            (stJob.substring(0,stJob.length()-9));
                      // Crea otra entidad con el nombre de surtidor1
                      entity surt = new entity("surtidor1");
                      /* Envía por "out" hacia Trands y fuera del
                      sistema, la entidad que acaba de procesar */
                      m.add(makeContent("out",job1));
                      /* Envia por "out1" hacia surtidor1,
                         la entidad con el nombre de surtidor1 */
                      m.add(makeContent("out1",surt));
                      /* Envía por "estado" hacia ColaCajas,
                         un mensaje de que queda libre la caja */
                      m.add(makeContent("estado",pasivo));
                }
                if(stJob.endsWith("surtidor2")) {
                      entity job1 = new entity
                            (stJob.substring(0,stJob.length()-9));
                      entity surt = new entity("surtidor2");
                      m.add(makeContent("out",job1));
                      m.add(makeContent("out2",surt));
                      m.add(makeContent("estado",pasivo));
                }
                if(stJob.endsWith("surtidor3")) {
                      entity job1 = new entity
                            (stJob.substring(0,stJob.length()-9));
                      entity surt = new entity("surtidor3");
                      m.add(makeContent("out",job1));
                      m.add(makeContent("out3",surt));
                      m.add(makeContent("estado",pasivo));
                }
          }
          return m;
    }
```

```
        /**
         * Método getTooltipText() que muestra datos del modelo
         * al situar el cursor del ratón sobre él en la visualización
         */

        public String getTooltipText(){
           return
           super.getTooltipText();
        }

} // Fin de CajaSimp
```

B.9 TransdDCaja.java

```
package gasolinera;

import genDevs.modeling.message;
import GenCol.Function;
import GenCol.doubleEnt;
import GenCol.entity;
import simView.ViewableAtomic;

public class TransdCaja extends  ViewableAtomic {
            protected Function  arrived, solved;      /* Funciones para
                        el control de los tiempos de las entidades */
            protected double observation_time; /* Parámetro del
                modelo, que indica el tiempo que dura la simulación */
            /* Variables auxiliares para la generación
               de los datos estadísticos: */
            protected double clock,total_ta, tmax;
            protected double noCola_c, espera_c;

        /**
         * Método Constructor:
         * @param name
         * @param Observation_time
         */

        public TransdCaja(String  name,double Observation_time){
              super(name);
              addInport("ariv");      // Puertos de entrada
              addInport("solved");
              arrived = new Function();
              solved = new Function();
              observation_time = Observation_time;
        }

        /**
          * Método inicial:
          */

        public void initialize(){
              phase = "active";
```

```
      sigma = observation_time;
      clock = 0;
      total_ta = 0;
      tmax = 0;
      noCola_c = 0;
      espera_c = 0;
      arrived = new Function();
      solved = new Function();
}

/**
  * Método de la Función de Transición Externa:
  */

public void  deltext(double e,message  x){
      clock = clock + e;      /* Actualiza la variable clock, que
                              indica el tiempo actual */
      Continue(e);      /* Este método reduce sigma en e,
                        es decir: sigma = sigma - e */
      entity  val;
      for(int i=0; i< x.size();i++){  /* Por si llega más de un
                              mensaje en el mismo instante */
            if(messageOnPort(x,"ariv",i)){
             val = x.getValOnPort("ariv",i);
             arrived.put(val.getName(),new doubleEnt(clock));
             /* Añade a la función arrived el nombre de la
      entidad que se acaba de recibir, y el tiempo actual */
          }
          if(messageOnPort(x,"solved",i)){
             val = x.getValOnPort("solved",i);
             /* Comprueba si la entidad recien llegada está en la
            función arrived */
             if(arrived.containsKey(val.getName())){
             /* Si está, toma el tiempo de llegada que figura en
                  arrived */
               entity  ent = (entity)arrived.
                                          assoc(val.getName());
               doubleEnt  num = (doubleEnt)ent;
               double arrival_time = num.getv();
               /* y se lo resta al tiempo actual para obtener el
                  tiempo que la entidad ha permanecido en la cola
                  de las cajas */
               double turn_around_time = clock - arrival_time;

               /* Actualiza la variable total_ta, que almacena
                  la suma del tiempo de todas las entidades
                  que han pasado por la cola de las cajas */
               total_ta = total_ta + turn_around_time;

               /* Añade la entidad a la función solved, junto con
                  el tiempo actual */
               solved.put(val, new doubleEnt(clock));

               // Actualiza las variables auxiliares
               if(turn_around_time > tmax) {
                  /* Variable que almacena el tiempo máximo de
                  espera en la cola de las cajas */
                   tmax = turn_around_time;
               }
```

```
                if(turn_around_time == 0) {
                   /* Variable que almacena el número de entidades
                   que no han esperado en la cola de cajas */
                    noCola_c = noCola_c + 1;
                }
                if(turn_around_time > 1) {
                   /* Variable que almacena el número de entidades
                   que han esperado en la cola de cajas más de un
                   minuto */
                    espera_c = espera_c + 1;
                }
              }
           }
       }
}

/**
 * Método de la Función de Transición Interna:
 */

public void  deltint(){
      /* La transición interna ocurre cuando termina el tiempo
         marcado por el parámetro observation_time, el modelo
         pasa al estado passive */
       passivate();


       /* Al final de la simulación, se mostrarán los datos
          estadísticos de la cola de las cajas */
       show_state();
}

 /**
  * Método para calcular el valor del tiempo de media que han
  * estado en la cola, todas las entidades que han pasado por
  * las cajas:
  */

public double compute_TA(){
        double avg_ta_time = 0;
        if(!solved.isEmpty())
          avg_ta_time = ((double)total_ta)/solved.size();
        return avg_ta_time;
}

/**
  * Método para calcular el valor del tiempo de media que han
  * estado en la cola, sólo las entidades que han esperado en la
  * cola de las cajas:
  */

public double compute_TA_2(){
        double avg_ta_time = 0;
        if(!solved.isEmpty())
          avg_ta_time = ((double)total_ta)/(solved.size() -
                                                   noCola_c);
        return avg_ta_time;
}
```

```
/**
  * Método para truncar los decimales de un número:
  * @param numero
  * @param decimales
  * @return
  */

public double redondear(double numero, int decimales) {
       return Math.round(numero*Math.pow(10, decimales))/
                                    Math.pow(10, decimales);
 }

/**
  * Método para imprimir los datos estadísticos
  * de las cajas:
  */

public void  show_state(){
      if (arrived != null && solved != null) {
            System.out.println("Total clientes que han pasado por
                  las Cajas = " + solved.size() + " clientes.");

            System.out.println("Total clientes que no han
            esperado en la cola = " + noCola_c + " clientes. ( "
            + redondear((noCola_c/solved.size())*100, 2)+ "%).");
            System.out.println("Total clientes que han esperado
            más de un minuto = " + espera_c + " clientes. ( " +
            redondear((espera_c/solved.size())*100, 2) + "%).");

            System.out.println("Tiempo máximo de espera en la
            cola de Cajas = "  + redondear(tmax, 2) + "
            minutos.");

            System.out.println("Tiempo medio de espera total en
            la cola de Cajas = "  +  redondear(compute_TA(), 2) +
            " minutos.");

            System.out.println("Tiempo medio de espera de los que
            han estado en la cola = "  +
            redondear(compute_TA_2(), 2) + " minutos.");
            System.out.println();
      }
}

/**
 * Metodo getTooltipText() que muestra datos del modelo
 * al situar el cursor del ratón sobre él en la visualización
 */

public String getTooltipText() {
      String  s = "";
        if (arrived != null && solved != null){
             s = "\n"+"coches llegados :" + arrived.size()
             +"\n"+"coches terminados :" + solved.size()
             +"\n" + "AVG TA = "  +  compute_TA();
        }
      return super.getTooltipText()+s;
}
} // Fin de TransdCaja
```

B.10 SetCajas.java

```
package gasolinera;

import java.awt.*;
import simView.*;

public class SetCajas extends ViewableDigraph {

      /**
       * Método Constructor:
       */

      public SetCajas(){
          super("setCajas");
          addInport("in");    //Puerto de entrada
          addOutport("out");  //Puertos de salida
          addOutport("outEnd1");
          addOutport("outEnd2");
          addOutport("outEnd3");
          addOutport("outQ");

            // Creación de los componentes del modelo
          ViewableAtomic colaCajas = new ColaCajas("colaCajas");
          ViewableAtomic caja1 = new CajaSimp("Caja 1");
          ViewableAtomic caja2 = new CajaSimp("Caja 2");
          ViewableAtomic tranCaja = new TransdCaja("transdCaja",1500);

          add(colaCajas);
          add(caja1);
          add(caja2);
          add(tranCaja);

          initialize();

          /* Los componentes se visualizarán con los
             colores que se indican. TransdCaja se deja con
             el color por defecto, que es el gris  */
          Color color;
          color = Color.cyan;
          colaCajas.setBackgroundColor(color);
          color = Color.green;
          caja1.setBackgroundColor(color);
          caja2.setBackgroundColor(color);

          // Se realizan los enlaces entre los puertos
          addCoupling(this,"in",colaCajas,"in");
          addCoupling(colaCajas,"outQ",this,"outQ");
          addCoupling(colaCajas,"out1",caja1,"in");
          addCoupling(colaCajas,"out2",caja2,"in");

          addCoupling(caja1,"out",this,"out");
          addCoupling(caja2,"out",this,"out");
          addCoupling(caja1,"out1",this,"outEnd1");
          addCoupling(caja1,"out2",this,"outEnd2");
          addCoupling(caja1,"out3",this,"outEnd3");
          addCoupling(caja2,"out1",this,"outEnd1");
          addCoupling(caja2,"out2",this,"outEnd2");
```

```
        addCoupling(caja2,"out3",this,"outEnd3");
        addCoupling(caja1,"estado",colaCajas,"estado1");
        addCoupling(caja2,"estado",colaCajas,"estado2");

        addCoupling(this,"in",tranCaja,"ariv");
        addCoupling(colaCajas,"out1",tranCaja,"solved");
        addCoupling(colaCajas,"out2",tranCaja,"solved");
    }

    /**
     * Automatically generated by the SimView program.
     * Do not edit this manually, as such changes will get
     * overwritten.
     */
    public void layoutForSimView()
    {
        preferredSize = new Dimension(309, 390);
        ((ViewableComponent)withName("Caja 2")).
                            setPreferredLocation(new Point(23, 208));

        ((ViewableComponent)withName("transdCaja")).
                            setPreferredLocation(new Point(3, 327));
        ((ViewableComponent)withName("Caja 1")).
                            setPreferredLocation(new Point(20, 20));
        ((ViewableComponent)withName("colaCajas")).
                            setPreferredLocation(new Point(-11,128));
    }
} // Fin de SetCajas
```

B.11 NetGasolinera.java

```
package gasolinera;

import java.awt.*;
import simView.*;
import genDevs.plots.CellGridPlot;

public class NetGasolinera extends ViewableDigraph {

      /**
       * Método Constructor:
       */
public NetGasolinera(){
    super("Gasolinera");
    addInport("in");    //Puerto de entrada
    addOutport("out");  //Puertos de salida

    // Creación de los componentes del modelo
    ViewableDigraph ef = new Ef("ef",5,1440); /* Nombre, intervalo de
                            generación de las entidades, y tiempo de
                            duración de la simulación */
    ViewableDigraph cajas = new SetCajas();

    ViewableAtomic entrada = new Entrada("entrada");
```

```
ViewableAtomic surtidor1 = new Surtidor("surtidor1");
ViewableAtomic surtidor2 = new Surtidor("surtidor2");
ViewableAtomic surtidor3 = new Surtidor("surtidor3");
ViewableAtomic tranSurt = new TransdSurt("transdSurt", 1500);

add(ef);
add(entrada);
add(surtidor1);
add(surtidor2);
add(surtidor3);
add(cajas);
add(tranSurt);

initialize();

/* Los componentes se visualizarán con los
colores que se indican. TransdSurt se deja con
el color por defecto, que es el gris  */
Color color;
color = Color.pink;
entrada.setBackgroundColor(color);
color = Color.red;
surtidor1.setBackgroundColor(color);
surtidor2.setBackgroundColor(color);
surtidor3.setBackgroundColor(color);

/* Creación de las instancias de CellGridPlot,
   para la visualización de gráficas con los datos
   de la ocupación de las colas */
CellGridPlot cochesSurtidor1 = new CellGridPlot("Coches en
                                          Surtidor 1",10,10);
cochesSurtidor1.setCellGridViewLocation(500,0);
cochesSurtidor1.setSpaceSize(100,25);
cochesSurtidor1.setCellSize(5);
cochesSurtidor1.setTimeScale(1500);
add(cochesSurtidor1);
addCoupling(surtidor1,"outQ",cochesSurtidor1,"timePlot");

CellGridPlot cochesSurtidor2 = new CellGridPlot("Coches en
                                          Surtidor 2",10,10);
cochesSurtidor2.setCellGridViewLocation(500,175);
cochesSurtidor2.setSpaceSize(100,25);
cochesSurtidor2.setCellSize(5);
cochesSurtidor2.setTimeScale(1500);
add(cochesSurtidor2);
addCoupling(surtidor2,"outQ",cochesSurtidor2,"timePlot");

CellGridPlot cochesSurtidor3 = new CellGridPlot("Coches en
                                          Surtidor 3",10,10);
cochesSurtidor3.setCellGridViewLocation(500,350);
cochesSurtidor3.setSpaceSize(100,25);
cochesSurtidor3.setCellSize(5);
cochesSurtidor3.setTimeScale(1500);
add(cochesSurtidor3);
addCoupling(surtidor3,"outQ",cochesSurtidor3,"timePlot");

CellGridPlot personasCaja = new CellGridPlot("Personas en
                                          Caja",10,10);
personasCaja.setCellGridViewLocation(500,530);
```

```
    personasCaja.setSpaceSize(100,25);
    personasCaja.setCellSize(5);
    personasCaja.setTimeScale(1500);
    add(personasCaja);
    addCoupling(cajas,"outQ",personasCaja,"timePlot");

    // Conexión de los enlaces entre los puertos
    addCoupling(entrada,"out1",tranSurt,"ariv");
    addCoupling(entrada,"out2",tranSurt,"ariv");
    addCoupling(entrada,"out3",tranSurt,"ariv");
    addCoupling(surtidor1,"transd",tranSurt,"solved1");
    addCoupling(surtidor2,"transd",tranSurt,"solved2");
    addCoupling(surtidor3,"transd",tranSurt,"solved3");

    addCoupling(ef,"out",entrada,"in");
    addCoupling(this,"in",entrada,"in");

     addCoupling(entrada,"out1",surtidor1,"in");
     addCoupling(entrada,"out2",surtidor2,"in");
     addCoupling(entrada,"out3",surtidor3,"in");

     addCoupling(surtidor1,"out",cajas,"in");
     addCoupling(surtidor1,"outQ",entrada,"inQ1");
     addCoupling(surtidor2,"out",cajas,"in");
     addCoupling(surtidor2,"outQ",entrada,"inQ2");
     addCoupling(surtidor3,"out",cajas,"in");
     addCoupling(surtidor3,"outQ",entrada,"inQ3");

     addCoupling(cajas,"outEnd1",surtidor1,"inEnd");
     addCoupling(cajas,"outEnd2",surtidor2,"inEnd");
     addCoupling(cajas,"outEnd3",surtidor3,"inEnd");
     addCoupling(cajas,"out",this,"out");
     addCoupling(cajas,"out",ef,"in");
}
    /**
     * Automatically generated by the SimView program.
     * Do not edit this manually, as such changes will get
overwritten.
     */
    public void layoutForSimView()
    {
        preferredSize = new Dimension(763, 559);
        ((ViewableComponent)withName("setCajas")).
                              setPreferredLocation(new Point(373, 31));
        ((ViewableComponent)withName("transdSurt")).
                              setPreferredLocation(new Point(-12, 17));
        ((ViewableComponent)withName("ef")).
                              setPreferredLocation(new Point(1, 302));
        ((ViewableComponent)withName("surtidor3")).
                              setPreferredLocation(new Point(167,244));
        ((ViewableComponent)withName("surtidor2")).
                              setPreferredLocation(new Point(170,143));
        ((ViewableComponent)withName("surtidor1")).
                              setPreferredLocation(new Point(171, 35));
        ((ViewableComponent)withName("entrada")).
                              setPreferredLocation(new Point(-11,133));
    }

} // Fin de NetGasolinera
```

Anexo C

Código CD++ del Sistema Logístico

C.1 register.cpp

```
#include "modeladm.h"
#include "mainsimu.h"

#include "entrada.h"          // class Entrada
#include "surtidor.h"         // class Surtidor
#include "generador.h"        // class Generador
#include "transd.h"           // class Transductor
#include "transdSurt.h"       // class TransdSurt
#include "colaCajas.h"        // class ColaCajas
#include "cajaSimple.h"       // class CajaSimple
#include "transdCaja.h"       // class TransdCaja

/* Todos los modelos atómicos que se necesita crear, hay que
   registrarlos en este método. Primero se incluye el archivo de
   cabecera */
void MainSimulator::registerNewAtomics() {
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<Entrada>() , "Entrada" ) ;
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<Surtidor>() , "Surtidor" ) ;
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<TransdSurt>(), "TransdSurt");
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<Generador>(), "Generador");
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<Transductor>() , "Transd" ) ;
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<ColaCajas>() , "ColaCajas" );
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<CajaSimple>(), "CajaSimple");
      SingleModelAdm::Instance().registerAtomic(
                        NewAtomicFunction<TransdCaja>(), "TransdCaja");

} // Fin de register.cpp
```

C.2 generador.h

```
#ifndef __GENERADOR_H
#define __GENERADOR_H

#include "atomic.h"      // class Atomic
#include "except.h"      // class InvalidMessageException

class Distribution ;    //Declaración de la clase Ditribution

class Generador : public Atomic
{
public:
      // Default Constructor
      Generador( const string &name = "Generador" );

      virtual string className() const
            {return "Generador";}

protected:
      //Métodos de control
      Model &initFunction() ;
      Model &externalFunction( const ExternalMessage & );
      Model &internalFunction( const InternalMessage & );
      Model &outputFunction( const InternalMessage & );

private:
      //Variables para el contador
      int pid;
      int initial, increment;

      const Port &stop;       //Puerto de entrada
      Port &out ;             //Puerto de salida

      //Variable de la clase Distribution
      Distribution *dist ;
      Distribution &distribution()
                  {return *dist;}

};    // class Generador

#endif   //__GENERADOR_H
```

C.3 generador.cpp

```
#include "generador.h"        // base class
#include "message.h"          // class InternalMessage
#include "mainsimu.h"         // class Simulator
#include "distri.h"           // class Distribution
#include "strutil.h"          // str2Value( ... )
```

```
/*******************************************************************
* Function Name: Generador
* Description: constructor
********************************************************************/

Generador::Generador( const string &name )
: Atomic( name )
, out( addOutputPort( "out" ) )     //Puerto de salida
, stop( addInputPort( "stop" ) )    //Puerto de entrada
{

      try
      {     /* Busca el tipo de función de distribución, que está
            declarado en el archivo del modelo acoplado
            correspondiente, y el parámetro de la función */
            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 ) ) ;
            }

            /* Busca si se ha declarado el valor de initial en el
            archivo del modelo acoplado correspondiente */
            if( MainSimulator::Instance().
                        existsParameter(description(), "initial" ) )
                  initial = str2Int( MainSimulator::Instance().
                        getParameter( description(), "initial" ) );
            /* Si no lo encuentra, le asigna el valor 1 */
            else
                  initial = 1;

            /* Busca si se ha declarado el valor de increment en el
            archivo del modelo acoplado correspondiente */
            if( MainSimulator::Instance().
                        existsParameter(description(), "increment" ) )
                  increment = str2Int( MainSimulator::Instance().
                        getParameter( description(), "increment" ) );

            /* Si no lo encuentra, le asigna el valor 1 */
            else
                  increment = 1;

      } catch( InvalidDistribution &e )
      {
            e.addText( "The model " + description() + " has
                                          distribution problems!" ) ;
            e.print(cerr);
            MTHROW( e ) ;
      } catch( MException &e )
      {
            MTHROW( e ) ;
      }
}
```

```
/*****************************************************************
* Function Name: initFunction
*****************************************************************/

Model &Generador::initFunction()
{
	pid = initial;    /*La variable que almacena el contador
	                   se inicializa con el valor de initial.
	                   En este modelo es 1 */
	holdIn( active, Time::Zero ) ;/* Al iniciar el modelo en el
	                               estado active con ta=0, se genera
	                               una entidad en el momento inicial*/
	return *this ;
}

/*****************************************************************
* Function Name: externalFunction
*****************************************************************/
Model &Generador::externalFunction( const ExternalMessage &msg )
{
	/*Si llega algún mensaje al puerto "stop",
	 el modelo pasa al estado passive */
	passivate();

	return *this ;
}

/*****************************************************************
* Function Name: internalFunction
*****************************************************************/
Model &Generador::internalFunction( const InternalMessage & )
{
	holdIn(active, Time(0,0,fabs( distribution().get() ),0));
	/* Cada vez que se produce una transición interna,
	el modelo pasa al estado active,
	con una transición planificada en el tiempo que
	fije la función exponencial */

	return *this ;
}

/*****************************************************************
* Function Name: outputFunction
*****************************************************************/
Model &Generador::outputFunction( const InternalMessage &msg )
{
	sendOutput( msg.time(), out, pid ) ;

	pid += increment;
	/* Envía por el puerto "out", un mensaje con el valor de la
	variable pid, y la incrementa según el valor de increment. En
	este modelo es 1 */
	return *this ;
}

// Fin de generador.cpp
```

C.4 transd.h

```
#ifndef __TRANSDUCTOR_H
#define __TRANSDUCTOR_H

#include <fstream.h>    // class fstream
#include <list>         // class list
#include <map.h>        // class map
#include "atomic.h"     // class Atomic

class Transductor: public Atomic
{
public:
	//Default Constructor
	Transductor( const string &name = "Transductor" ) ;

	virtual string className() const ;

protected:
	//Métodos de control
	Model &initFunction();
	Model &externalFunction( const ExternalMessage & );
	Model &internalFunction( const InternalMessage & );
	Model &outputFunction( const InternalMessage & );

private:
	typedef map< int, Time, less<int> > JobsList ;
	typedef list<Value> ElementsList;

	const Port &arrived ;	//Puertos de entrada
	const Port &solved ;
	Port &out ;		//Puerto de salida
	//Variable para almacenar el tiempo de duración
	Time observ_time ;
	//Variables auxiliares para los datos estadísticos
	Time parcial_ta, total_ta;
	/*Variable para almacenar las entidades que llegan
	  y sus tiempos de llegada */
	JobsList unsolved ;
	/*Variables para almacenar las entidades que llegan
	  por cada uno de los dos puertos de entrada */
	ElementsList arriv, solv;
	//Variable para el archivo salida.txt
	ofstream archivo;

	const Time &observation_time() const ;

};	// class Transductor

// ** inline ** //

inline
//Devuelve el nombre de la clase
string Transductor::className() const
{
	return "Transductor" ;
}
```

```
inline
//Devuelve el valor de la variable observ_time
const Time &Transductor::observation_time() const
{
	return observ_time ;
}

#endif   //__TRANSDUCTOR_H
```

C.5 transd.cpp

```
#include "transd.h"     // base class
#include "message.h"    // class ExternalMessage
#include "mainsimu.h"   // class MainSimulator

/*******************************************************************
* Function Name: Transductor
* Description: Constructor
********************************************************************/
Transductor::Transductor( const string &name )
: Atomic( name )
, arrived( addInputPort( "arrived" ) )   //Puertos de entrada
, solved( addInputPort( "solved" ) )
, out( addOutputPort( "out"))            //Puerto de salida
{
	/* Valor por defecto. Será el que utilice este modelo */
	observ_time = "24:0:0:0" ;

	/* Si hay un valor asignado en el archivo del modelo compuesto
	   correspondiente, se lo asigna */
	if( MainSimulator::Instance().existsParameter( description(),
	                                    "observation_time" ) )
		observ_time = MainSimulator::Instance().
		            getParameter(description(),"observation_time");
}
/*******************************************************************
* Function Name: initFunction
********************************************************************/
Model &Transductor::initFunction()
{
	//Inicia las variables de tipo map y tipo list
	unsolved.erase( unsolved.begin(), unsolved.end() ) ;
	arriv.erase(arriv.begin(), arriv.end());
	solv.erase(solv.begin(), solv.end());

	/* Abre el archivo salida.txt, en el que
	imprimirá los datos estadísticos */
	archivo.open("salida.txt");

	/* Inicia el modelo en el estado active con el tiempo
	   que almacena la variable observ_time */
	holdIn( active, observation_time() ) ;
	return *this ;
```

```
}
/*******************************************************************
* Function Name: externalFunction
*******************************************************************/
Model &Transductor::externalFunction( const ExternalMessage &msg )
{
	if( msg.port() == arrived )
	{
		if( unsolved.find( msg.value() ) != unsolved.end() )
		{
			MException e( string("Unresolved Work Id: ") +
			                  msg.value() + " is duplicated." );
			e.addLocation( MEXCEPTION_LOCATION() );
			throw e;
			/*Si encuentra un valor en unsolved igual al de la
			entidad que acaba de llegar por el puerto arrived,
			lanza un mensaje de error*/
		}

		/* Introduce la entidad y el tiempo de su llegada
		   en la variable unsolved */
		unsolved[ msg.value() ] = msg.time() ;

		/* Introduce la entidad en la lista arriv */
		arriv.push_back(msg.value());
	}

	if( msg.port() == solved )
	{
		JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

		if( cursor == unsolved.end() )
		{
			MException e( string("Resolved Work Id: ") +
			                  msg.value() + " Not Found!" );
			e.addLocation( MEXCEPTION_LOCATION() );
			throw e;
			/* Si no encuentra el valor de la entidad que acaba
			de llegar por el puerto solved,lanza un mensaje de
			error */
		}

		/* Actualiza las variables auxiliares */
		parcial_ta =  msg.time() - unsolved[msg.value()];
		total_ta = total_ta + parcial_ta;

		/* Introduce la entidad en la lista solv */
		solv.push_back(msg.value());

		/* Borra la entidad de la variable unsolved */
		unsolved.erase( cursor ) ;

	}

	/* Cada vez que llega una entidad por alguno de los dos puertos
	de entrada, imprime en el archivo salida.txt los datos
	estadísticos */
	if((observ_time - msg.time()) > Time::Zero) {
		archivo << "sigma : " << observ_time - msg.time() << endl;
```

```
        }
        else {
              archivo << "sigma : " << Time::Zero << endl;
        }
        archivo << "Total coches llegados = " << arriv.size() <<
                                                "coches." << endl;
        archivo << "Total coches terminados = " << solv.size() <<
                                                "coches." << endl;
        if(!(solv.size() == 0)) {
              archivo << "Tiempo medio de permanencia = " <<
              (double(int(total_ta.asMsecs()/1000))/60) / solv.size() <<
              " minutos." << endl;
        }
        archivo << endl;

        return *this ;
}

/*******************************************************************
* Function Name: internalFunction
*******************************************************************/
Model &Transductor::internalFunction( const InternalMessage &)
{       /* La transición interna ocurre cuando termina el tiempo
           marcado por la variable observ_time, el modelo pasa al estado
        passive */
        passivate();
        return *this ;
}

/*******************************************************************
* Function Name: outputFunction
*******************************************************************/
Model &Transductor::outputFunction( const InternalMessage &msg )
{       //Envía un mensaje por el puerto "out", con el valor 1
        sendOutput( msg.time(), out, 1);

        return *this ;
}
```

C.6 ef.ma

```
[top]
%Creación de los componentes. Se crea la instancia gen
%de la clase generador, y la instancia transd de la clase transd.
components : gen@generador transd@transd

in : in             % Puerto de entrada
out : out           % Puerto de salida

% Conexión de los enlaces entre los puertos
Link : in solved@transd
Link : out@transd stop@gen
Link : out@gen arrived@transd
Link : out@gen out
```

```
% Parámetros de la función de distribución
% de la instancia gen de generador.
% Es una función exponencial, de parámetro 300(segundos),
% que son los 5 minutos que debe tener el modelo.
[gen]
distribution : exponential
mean : 300
```

C.7 entrada.h

```
#ifndef __ENTRADA_H
#define __ENTRADA_H

#include "atomic.h"     // class Atomic

class Entrada : public Atomic
{
public:
      //Default constructor
      Entrada( const string &name = "Entrada" );

      virtual string className() const ;
protected:
      //Métodos de control
      Model &initFunction();
      Model &externalFunction( const ExternalMessage & );
      Model &internalFunction( const InternalMessage & );
      Model &outputFunction( const InternalMessage & );

private:
      const Port &in;               //Puertos de entrada
      const Port &inQ1;
      const Port &inQ2;
      const Port &inQ3;
      Port &out1;                   //Puertos de salida
      Port &out2;
      Port &out3 ;
      double q1, q2, q3;      /* Variables para los valores de las
                              colas de los surtidores */
      Value valor;            /* Variable para almacenar temporalmente
                              el valor de las entidades */

};    // class Entrada

// ** inline ** //
inline
//Devuelve el nombre de la clase
string Entrada::className() const
{
      return "Entrada" ;
}
#endif   //__ENTRADA_H
```

C.8 entrada.cpp

```
#include "entrada.h"    // class Entrada
#include "message.h"    // class ExternalMessage, InternalMessage
#include "mainsimu.h"   // MainSimulator::Instance().getParameter(...)

/*******************************************************************
* Function Name: Entrada
********************************************************************/
Entrada::Entrada( const string &name )
: Atomic( name )
, in(addInputPort("in"))           //Puertos de entrada
, inQ1(addInputPort("inQ1"))
, inQ2(addInputPort("inQ2"))
, inQ3(addInputPort("inQ3"))
, out1( addOutputPort("out1"))     //Puertos de salida
, out2( addOutputPort("out2"))
, out3( addOutputPort("out3"))
{
}

/*******************************************************************
* Function Name: initFunction
********************************************************************/
Model &Entrada::initFunction()
{
   q1 = q2 = q3 = 0;
   return *this ;
}

/*******************************************************************
* Function Name: externalFunction
********************************************************************/
Model &Entrada::externalFunction( const ExternalMessage &msg )
{
      if( msg.port() == in ){
      /* Si se recibe una entidad en el puerto "in",
      el modelo pasa al estado active */
      valor = msg.value();
      holdIn(active,0);
   }
   if (msg.port() == inQ1) {
      /* Si se recibe un mensaje en el puerto "inQ1",
      se actualiza el valor de q1 */
            q1 = (double)msg.value();
      }
   if (msg.port() == inQ2) {
               q2 = (double)msg.value();
      }
   if (msg.port() == inQ3) {
               q3 = (double)msg.value();
      }

      return *this;
}
```

```
/*******************************************************************
* Function Name: internalFunction
*******************************************************************/
Model &Entrada::internalFunction( const InternalMessage & )
{
    passivate();
      return *this ;
}

/*******************************************************************
* Function Name: outputFunction
*******************************************************************/
Model &Entrada::outputFunction( const InternalMessage &msg )
{
   if ((q1<q2)&&(q1<q3)) {
      /* q1 es la cola más corta, y la entidad se envía
      por el puerto "out1" */
      sendOutput( msg.time(), out1, valor ) ;
   }
   else if((q2<q1)&&(q2<q3)) {
      /* q2 es la cola más corta, y la entidad se envía
      por el puerto "out2" */
         sendOutput( msg.time(), out2, valor ) ;
   }
   else if ((q3<q1)&&(q3<q2)) {
      /* q3 es la cola más corta, y la entidad se envía
      por el puerto "out3" */
         sendOutput( msg.time(), out3, valor ) ;
   }
   else if ((q1==q2)&&(q1<q3)) {
         // q1 y q2 son iguales y las más cortas
         int n ;
         /* genera un número aleatorio con la función rand(),
            y le aplica la operación módulo 2
            para elegir el puerto de salida */
         n = rand() % 2;
         if(n == 0) {
               sendOutput( msg.time(), out2, valor ) ;
         }
         else {
               sendOutput( msg.time(), out1, valor );
         }
   }
   else if ((q1==q3)&&(q1<q2)) {
         // q1 y q3 son iguales y las más cortas
         int n ;
       /* genera un número aleatorio con la función rand(),
          y le aplica la operación módulo 2
          para elegir el puerto de salida */
         n = rand() % 2;
         if(n == 0) {
               sendOutput( msg.time(), out3, valor ) ;
         }
         else {
               sendOutput( msg.time(), out1, valor );
         }
   }
   else if ((q2==q3)&&(q2<q1)) {
         // q2 y q3 son iguales y las más cortas
```

```
          int n ;
        /* genera un número aleatorio con la función rand(),
           y le aplica la operación módulo 2
           para elegir el puerto de salida */
          n = rand() % 2;
          if(n == 0) {
                sendOutput( msg.time(), out3, valor ) ;
          }
          else {
                sendOutput( msg.time(), out2, valor );
          }
    }
    else if ((q1==q2)&&(q1==q3)) {
          // Las tres colas son iguales
          int n ;
       /* genera un número aleatorio con la función rand(),
          y le aplica la operación módulo 3
          para elegir el puerto de salida */
          n = rand() % 3;
          if(n == 0) {
                sendOutput( msg.time(), out3, valor ) ;
          }
          else if(n == 1){
                sendOutput( msg.time(), out1, valor );
          }
          else if(n == 2){
                sendOutput( msg.time(), out2, valor );
        }
    }
    else {
          cout << "ERROR en 'entrada' con las colas de los surtidores."
                                                   << endl;
}
    return *this ;
}
```

C.9 surtidor.h

```
#ifndef __SURTIDOR_H
#define __SURTIDOR_H

#include <list>          // class list
#include "atomic.h"      // class Atomic
#include "randlib.h"     // class randlib

class Surtidor : public Atomic
{
public:
      //Default constructor
      Surtidor( const string &name );

      virtual string className() const ;
```

```
protected:

	// Tipo enumerado que contiene los estados del modelo
	enum State  {passive, busy, sendOutQ, sendOutQ2, sendQ, sendQ2,
	             wait, end};

	/* Variable de tipo enum para almacenar temporalmente el estado
	del modelo */
	State estado;

	// Métodos de control
	Model &initFunction();
	Model &externalFunction( const ExternalMessage & );
	Model &internalFunction( const InternalMessage & );
	Model &outputFunction( const InternalMessage & );

private:
	const Port &in;		// Puertos de entrada
	const Port &inEnd;
	Port &out;		// Puertos de salida
	Port &outQ;
	Port &transd;

	/* Variable para almacenar temporalmente el valor de las
	entidades */
	Value valor;

	// Variable para almacenar el tamaño de la cola
	double tcola;

	/* Variable auxiliar para almacenar el tiempo restante,
	cuando se interrumpe el proceso */
	Time resto;

	// Variable para almacenar el nombre del surtidor
	string nombre;

	typedef list<Value> ElementList ;

	// Variable para gestionar la cola
	ElementList elements ;

};	// class Surtidor

// ** inline ** //

inline
string Surtidor::className() const
{
	return "Surtidor" ;
}


#endif   //__SURTIDOR_H
```

C.10 surtidor.cpp

```
#include "surtidor.h"   // class Surtidor
#include "message.h"    // class ExternalMessage, InternalMessage
#include "mainsimu.h"   // MainSimulator::Instance().getParameter(...)
#include "randlib.c"    // class genunf
#include "com.c"        // class ignlgi
#include "linpack.c"    // class spofa

/*******************************************************************
* Function Name: Surtidor
* Description:
*******************************************************************/
Surtidor::Surtidor( const string &name )
: Atomic( name )
, in(addInputPort("in"))            // Puertos de entrada
, inEnd(addInputPort("inEnd"))
, out( addOutputPort("out"))        // Puertos de salida
, outQ( addOutputPort("outQ"))
, transd( addOutputPort("transd"))
{
      nombre = name;    // Nombre del surtidor
}

/*******************************************************************
* Function Name: initFunction
*******************************************************************/
Model &Surtidor::initFunction()
{
      //Inicia la variable para gestionar la cola
      elements.erase( elements.begin(), elements.end() ) ;

      estado = passive; // Se inicia el modelo en el estado passive
      nextChange(Time::Inf); // con tiempo infinito

      return *this ;
}

/*******************************************************************
* Function Name: externalFunction
*******************************************************************/
Model &Surtidor::externalFunction( const ExternalMessage &msg )
{
      if(estado == passive) {
            if( msg.port() == in ){
                  /* Si llega una entidad al puerto "in", la mete en la
                  cola */
                  elements.push_back(msg.value());
            }

            /* Coge a la primera de la cola. De este modo se asegura de
            que es la primera entidad en la cola */
            valor = elements.front();

            /* Calcula el tiempo de proceso, y lo guarda en la variable
            resto */
            nextChange(Time(0,0,genunf(120,300),0));
            resto = nextChange();
```

```
        /* Planifica una transición interna para este mismo
        instante, para que se active la función de salida */
        estado = sendOutQ;
        nextChange(Time::Zero);

}
if (estado == busy) {
        if(msg.port() == in) {
                /* Si llega una entidad al puerto "in", la mete en la
                cola */
                elements.push_back(msg.value());

                /* Almacena en la variable resto el tiempo que falta
                para terminar con la entidad que está siendo
                procesada */
                resto = nextChange();

                /* Planifica una transición interna para este mismo
                instante, para que se active la función de salida */
                estado = sendQ;
                nextChange(Time::Zero);
        }
}
 if(estado == wait) {
        if(msg.port() == inEnd) {
                /* Elimina a la primera de la cola,
                  que es la última que se ha procesado */
                elements.pop_front();

                /* Planifica una transición interna para este mismo
                instante, para que se active la función de salida */
                estado = end;
                nextChange(Time::Zero);
        }
        else if(msg.port() == in) {
                /* Si llega una entidad al puerto "in", la mete en la
                cola */
                elements.push_back(msg.value());

                /* Planifica una transición interna para este mismo
                instante, para que se active la función de salida */
                estado = sendQ2;
                nextChange(Time::Zero);
        }
}
else if(estado == end) {
        if(msg.port() == in) {
                /* Si llega una entidad al puerto "in", la mete en la
                cola */
                elements.push_back(msg.value());
        }
}

return *this;

}
```

```
/*******************************************************************
* Function Name: internalFunction
********************************************************************/
Model &Surtidor::internalFunction( const InternalMessage & )
{

    if(estado == sendQ) {
      // Pasa al estado busy, con el tiempo de la variable resto
      estado = busy;
      nextChange(Time(resto));
    }
    else if(estado == busy) {
      // Pasa al estado wait con tiempo infinito
      estado = wait;
      nextChange(Time::Inf);

    }
    else if(estado == sendQ2) {
      // Pasa al estado wait con tiempo infinito
      estado = wait;
      nextChange(Time::Inf);
    }
    else if(estado == end) {
      if(!(elements.size() == 0)) {
            /* Si la cola no está vacía,
            coge a la primera entidad de la cola */
            valor = elements.front();

            /* Calcula el tiempo de proceso, y lo guarda en la variable
            resto */
            nextChange(Time(0,0,genunf(120,300),0));
            resto = nextChange();

            /* Planifica una transición interna para este mismo
            instante, para que se active la función de salida */
            estado = sendOutQ2;
            nextChange(Time::Zero);
      }
      else // passivate();
      {     estado = passive;
                  nextChange(Time::Inf);
      }
    }
    else if(estado == sendOutQ) {
      /* Planifica una transición interna para este mismo instante,
      para que se active la función de salida */
      estado = sendQ;
      nextChange(Time::Zero);
    }
    else if(estado == sendOutQ2) {
      // Pasa al estado busy, con el tiempo de la variable resto
      estado = busy;
      nextChange(Time(resto));
    }

      return *this ;

}
```

```
/*******************************************************************
* Function Name: outputFunction
********************************************************************/
Model &Surtidor::outputFunction( const InternalMessage &msg )
{
      if((estado == sendQ) || (estado == sendQ2) || (estado == end)) {
            // Envía por "outQ" hacia entrada, el tamaño de la cola
            tcola = elements.size();
            sendOutput( msg.time(), outQ, tcola ) ;
      }
      if(estado == busy) {
            /* Envía por "out" hacia colaCajas, el valor de la entidad
               que acaba de procesar, sumándole antes una cantidad muy
               grande, distinta según sea el nombre del surtidor */
            double num;
            if(nombre == "surtidor1")  num = 100000;
            if(nombre == "surtidor2")  num = 200000;
            if(nombre == "surtidor3")  num = 300000;
            Value val = valor + num;
            sendOutput( msg.time(), out, val ) ;
      }
      if((estado == sendOutQ) || (estado == sendOutQ2)) {
            /* Envía por "transd" hacia transdSurt, el valor de la
            entidad */
            sendOutput(msg.time(), transd, valor);
      }
      return *this ;
}
```

C.11 transdSurt.h

```
#ifndef __TRANSDSURT_H
#define __TRANSDSURT_H


#include <fstream.h>    // class fstream
#include <list>         // class list
#include <map.h>        // class map
#include "atomic.h"     // class Atomic

class TransdSurt: public Atomic
{
public:
      //Default constructor
      TransdSurt( const string &name = "TransdSurt" ) ;

      virtual string className() const ;

protected:
      //Métodos de control
      Model &initFunction();
      Model &externalFunction( const ExternalMessage & );
      Model &internalFunction( const InternalMessage & );
      Model &outputFunction( const InternalMessage & );
```

```
private:
	typedef map< int, Time, less<int> > JobsList ;
	typedef list<Value> ElementsList;

	const Port &ariv ;	//Puertos de entrada
	const Port &solved1 ;
	const Port &solved2 ;
	const Port &solved3 ;

	//Variable para almacenar el tiempo de duración
	Time observ_time ;

	/*Variable para almacenar las entidades que llegan
	y sus tiempos de llegada */
	JobsList unsolved ;

	/*Variables para almacenar las entidades que llegan
	por cada uno de los cuatro puertos de entrada */
	ElementsList arrived, solv1, solv2, solv3;

	//Variables auxiliares para los datos estadísticos
	Time parcial_ta, total_ta_1, total_ta_2, total_ta_3;
	Time tmax_1, tmax_2, tmax_3;
	double noCola_1, noCola_2, noCola_3;
	double espera_1, espera_2, espera_3;

	//Variable para el archivo salida.txt
	ofstream archivo;

	const Time &observation_time() const ;

};	// class TransdSurt

// ** inline ** //
inline
string TransdSurt::className() const
{
	return "TransdSurt" ;
}

inline
const Time &TransdSurt::observation_time() const
{
	return observ_time ;
}

#endif	//__TRANSDSURT_H
```

C.12 transdSurt.cpp

```
#include "transdSurt.h"		// base class
#include "message.h"		// class ExternalMessage
#include "mainsimu.h"		// class MainSimulator
```

```
/*******************************************************************
* Function Name: TransdSurt
* Description: Constructor
*******************************************************************/
TransdSurt::TransdSurt( const string &name )
: Atomic( name )
, ariv( addInputPort( "ariv" ) )          // Puertos de entrada
, solved1( addInputPort( "solved1" ) )
, solved2( addInputPort( "solved2" ) )
, solved3( addInputPort( "solved3" ) )
{
      /* Valor por defecto. Será el que utilice este modelo */
      observ_time = "25:0:0:0" ;

      /* Si hay un valor asignado en el archivo del modelo compuesto
         correspondiente, se lo asigna*/
      if( MainSimulator::Instance().
                  existsParameter(description(),"observation_time" ) )
            observ_time = MainSimulator::Instance().
                  getParameter(description(), "observation_time" ) ;
}

/*******************************************************************
* Function Name: initFunction
*******************************************************************/
Model &TransdSurt::initFunction()
{
      // Inicia las variables de tipo map y tipo list
      unsolved.erase( unsolved.begin(), unsolved.end() ) ;

      arrived.erase(arrived.begin(), arrived.end());
      solv1.erase(solv1.begin(), solv1.end());
      solv2.erase(solv2.begin(), solv2.end());
      solv3.erase(solv3.begin(), solv3.end());

      // Inicializa las variables auxiliares
      total_ta_1 = total_ta_2 = total_ta_3 = Time::Zero;
      tmax_1 = tmax_2 = tmax_3 = Time::Zero;
      noCola_1 = noCola_2 = noCola_3 = 0;
      espera_1 = espera_2 = espera_3 = 0;

      /* Abre el archivo salida.txt, y  sitúa  el cursor
         para escribir al final del archivo */
      archivo.open("salida.txt", ios::app);

      /* Inicia el modelo en el estado active con el tiempo
         que almacena la variable observ_time */
      holdIn( active, observation_time() ) ;
      return *this ;
}

/*******************************************************************
* Function Name: externalFunction
*******************************************************************/

Model &TransdSurt::externalFunction( const ExternalMessage &msg )
{
```

```
if( msg.port() == ariv )
{
        if( unsolved.find( msg.value() ) != unsolved.end() )
        {
                MException e( string("Unresolved Work Id: ") +
                            msg.value() + " is duplicated." );
                e.addLocation( MEXCEPTION_LOCATION() );
                throw e;
                /*Si encuentra un valor en unsolved igual al de la
                entidad que acaba de llegar por el puerto arrived,
                lanza un mensaje de error*/
        }
        /* Introduce la entidad y el tiempo de su llegada
        en la variable unsolved */
        unsolved[ msg.value() ] = msg.time() ;

        /* Introduce la entidad en la lista arrived */
        arrived.push_back(msg.value());
}

if( msg.port() == solved1 )
{
        JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

        if( cursor == unsolved.end() )
        {
                MException e( string("Resolved Work Id: ") +
                                  msg.value() + " Not Found!" );
                e.addLocation( MEXCEPTION_LOCATION() );
                throw e;
                /* Si no encuentra el valor de la entidad que acaba
                de llegar por el puerto solved1,lanza un mensaje de
                error */
        }

        /* Actualiza las variables auxiliares */
        parcial_ta =  msg.time() - unsolved[msg.value()];
        total_ta_1 = total_ta_1 + parcial_ta;

        if(parcial_ta > tmax_1) {
                /* En esta variable se almacena el tiempo máximo de
                espera en la cola del surtidor1 */
                tmax_1 = parcial_ta;
        }
        if(parcial_ta == Time::Zero) {
                /* En esta variable se contabilizan las entidades que
                no han esperado en la cola del surtidor1 */
                noCola_1 = noCola_1 + 1;
        }
        if(parcial_ta.asMsecs() > 180000) {
                /* En esta variable se contabilizan las entidades que
                han esperado en la cola del surtidor1 más de tres
                minutos (180000 milisegundos) */
                espera_1 = espera_1 + 1;
        }

        /* Introduce la entidad en la lista solv1 */
        solv1.push_back(msg.value());
```

```
		/* Borra la entidad de la variable unsolved */
		unsolved.erase( cursor ) ;
	}

	if( msg.port() == solved2 ) // Similar a solved1
	{
		JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

		if( cursor == unsolved.end() )
		{
			MException e( string("Resolved Work Id: ") +
						msg.value() + " Not Found!" );
			e.addLocation( MEXCEPTION_LOCATION() );
			throw e;
		}

		parcial_ta =  msg.time() - unsolved[msg.value()];
		total_ta_2 = total_ta_2 + parcial_ta;
		solv2.push_back(msg.value());

		unsolved.erase( cursor ) ;

		if(parcial_ta > tmax_2) {
			tmax_2 = parcial_ta;
		}
		if(parcial_ta == Time::Zero) {
			noCola_2 = noCola_2 + 1;
		}
		if(parcial_ta.asMsecs() > 180000) {
			espera_2 = espera_2 + 1;
		}
	}

	if( msg.port() == solved3 )  // Similar a solved1
	{
		JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

		if( cursor == unsolved.end() )
		{
			MException e( string("Resolved Work Id: ") +
						msg.value() + " Not Found!" );
			e.addLocation( MEXCEPTION_LOCATION() );
			throw e;
		}

		parcial_ta =  msg.time() - unsolved[msg.value()];
		total_ta_3 = total_ta_3 + parcial_ta;
		solv3.push_back(msg.value());

		unsolved.erase( cursor ) ;

		if(parcial_ta > tmax_3) {
			tmax_3 = parcial_ta;
		}
		if(parcial_ta == Time::Zero) {
			noCola_3 = noCola_3 + 1;
		}
		if(parcial_ta.asMsecs() > 180000) {
			espera_3 = espera_3 + 1;
```

```
		}
	}

	return *this ;
}

/*******************************************************************
* Function Name: internalFunction
********************************************************************/
Model &TransdSurt::internalFunction( const InternalMessage &)
{
	/* La transición interna ocurre cuando termina el tiempo
	   marcado por la variable observ_time, el modelo pasa al estado
	passive */
	passivate();

	/* Al final de la simulación, imprime en el archivo salida.txt
	   los datos estadísticos de las colas de los tres surtidores */

	/* Para redondear a dos decimales se utiliza la expresión:
	   numero = double(int(numero*100.0+.5))/100.0; */

	archivo << "Total coches que han pasado por el Surtidor 1 = " <<
		solv1.size() << " coches." << endl;
	archivo << "Total coches que no han esperado en la cola = " <<
		noCola_1 << " coches. (" <<
		double(int(((noCola_1*100)/solv1.size())*100.0 + .5))/100.0
		<< "%)." <<endl;
	archivo << "Total coches que han esperado más de tres minutos =
		" << espera_1 << " coches. (" <<
		double(int(((espera_1*100)/solv1.size())*100.0 + .5))/100.0
		<< "%)." <<endl;
	archivo << "Tiempo máximo de espera en la cola del Surtidor 1 =
		" <<
		double(int((double(int(tmax_1.asMsecs()/1000))/60)*100.0 +
		.5))/100.0 << " minutos." << endl;
	if(!(solv1.size() == 0)) {
		archivo << "Tiempo medio de espera total en la cola del
		Surtidor 1 = " <<
		double(int(((double(int(total_ta_1.asMsecs()/1000))/60)/
		solv1.size())*100.0 + .5))/100.0 << " minutos." << endl;
		if(!(solv1.size() - noCola_1) == 0) {
			archivo << "Tiempo medio de espera de los que han
			estado en la cola = " <<
			double(int(((double(int(total_ta_1.asMsecs()/1000))/
			60)/(solv1.size() - noCola_1))*100.0 + .5))/100.0 <<
			" minutos." << endl;
		}
	}
	archivo << endl;

	archivo << "Total coches que han pasado por el Surtidor 2 = " <<
		solv2.size() << " coches." << endl;
	archivo << "Total coches que no han esperado en la cola = " <<
		noCola_2 << " coches. (" <<
		double(int(((noCola_2*100)/solv2.size())*100.0 + .5))/100.0
		<< "%)." <<endl;
```

```
	archivo << "Total coches que han esperado más de tres minutos = 
		" << espera_2 << " coches. (" << 
		double(int(((espera_2*100)/solv2.size())*100.0 + .5))/100.0 
		<< "%)." <<endl;
	archivo << "Tiempo máximo de espera en la cola del Surtidor 2 = 
		" << 
		double(int((double(int(tmax_2.asMsecs()/1000))/60)*100.0 + 
		.5))/ 100.0 << " minutos." << endl;
	if(!(solv2.size() == 0)) {
		archivo << "Tiempo medio de espera total en la cola del 
		Surtidor 2 = " << 
		double(int(((double(int(total_ta_2.asMsecs()/1000))/60)/
		solv2.size())*100.0 + .5))/100.0 << " minutos." << endl;
		if(!(solv2.size() - noCola_2) == 0) {
			archivo << "Tiempo medio de espera de los que han 
			estado en la cola = " << 
			double(int(((double(int(total_ta_2.asMsecs()/1000))/
			60)/(solv2.size() - noCola_2))*100.0 + .5))/100.0 << 
			" minutos." << endl;
		}
	}
	archivo << endl;

	archivo << "Total coches que han pasado por el Surtidor 3 = " << 
		solv3.size() << " coches." << endl;
	archivo << "Total coches que no han esperado en la cola = " << 
		noCola_3 << " coches. (" << 
		double(int(((noCola_3*100)/solv3.size())*100.0 + .5))/100.0 
		<< "%)." <<endl;
	archivo << "Total coches que han esperado más de tres minutos = 
		" << espera_3 << " coches. (" << 
		double(int(((espera_3*100)/solv3.size())*100.0 + .5))/100.0 
		<< "%)." <<endl;
	archivo << "Tiempo máximo de espera en la cola del Surtidor 3 = 
		" << 
		double(int((double(int(tmax_3.asMsecs()/1000))/60)*100.0 + 
		.5))/100.0 << " minutos." << endl;
	if(!(solv3.size() == 0)) {
		archivo << "Tiempo medio de espera total en la cola del 
		Surtidor 3 = " << 
		double(int(((double(int(total_ta_3.asMsecs()/1000))/60)/
		solv3.size())*100.0 + .5))/100.0 << " minutos." << endl;
		if(!(solv3.size() - noCola_3) == 0) {
			archivo << "Tiempo medio de espera de los que han 
			estado en la cola = " << 
			double(int(((double(int(total_ta_3.asMsecs()/1000))/
			60)/(solv3.size() - noCola_3))*100.0 + .5))/100.0 << 
			" minutos." << endl;
		}
	}
	archivo << endl;

	return *this ;
}
/*******************************************************************
* Function Name: outputFunction
*******************************************************************/
Model &TransdSurt::outputFunction( const InternalMessage &msg )
{ } /* En este modelo no se produce ninguna salida*/
```

C.13 colaCajas.h

```
#ifndef __COLACAJAS_H
#define __COLACAJAS_H

#include <list>          // class list
#include "atomic.h"      // class Atomic

class ColaCajas : public Atomic
{
public:
      //Default constructor
      ColaCajas( const string &name );

      virtual string className() const ;
protected:

      // Tipo enumerado que contiene los estados del modelo
      enum State   {passive, busy, sendQ, sendQ2, wait};
      /* Variable de tipo enum para almacenar temporalmente el estado
      del modelo */
      State estado;

      // Métodos de control
      Model &initFunction();
      Model &externalFunction( const ExternalMessage & );
      Model &internalFunction( const InternalMessage & );
      Model &outputFunction( const InternalMessage & );

private:
      const Port &in;               // Puertos de entrada
      const Port &estado1;
      const Port &estado2;
      Port &out1;                   // Puertos de salida
      Port &out2;
      Port &outQ;

      /* Variable para almacenar temporalmente el valor de las
      entidades */
      Value valor;

      // Variable para almacenar el tamaño de la cola
      double tcola;

      // Variables para almacenar el estado de las cajas
      bool pasivo1, pasivo2;

      /* Variable auxiliar para almacenar el tiempo restante,
      cuando se interrumpe el proceso */
      Time resto;

      typedef list<Value> ElementList ;
      // Variable para gestionar la cola
      ElementList elements ;

};    // class ColaCajas
```

```
// ** inline ** //
inline
string ColaCajas::className() const
{
	return "ColaCajas" ;
}
#endif   //__COLACAJAS_H
```

C.14 colaCajas.cpp

```
#include "colaCajas.h"  // class Surtidor
#include "message.h"    // class ExternalMessage, InternalMessage
#include "mainsimu.h"   // MainSimulator::Instance().getParameter(...)

/*******************************************************************
* Function Name: ColaCajas
* Description:
*******************************************************************/
ColaCajas::ColaCajas( const string &name )
: Atomic( name )
, in(addInputPort("in"))                 // Puertos de entrada
, estado1(addInputPort("estado1"))
, estado2(addInputPort("estado2"))
, out1( addOutputPort("out1"))           // Puertos de salida
, out2( addOutputPort("out2"))
, outQ( addOutputPort("outQ"))
{
}

/*******************************************************************
* Function Name: initFunction
*******************************************************************/
Model &ColaCajas::initFunction()
{
	//Inicia la variable para gestionar la cola
	elements.erase( elements.begin(), elements.end() ) ;

	//Inicia con las dos cajas libres
	pasivo1 = true;
	pasivo2 = true;
//	pasivo2 = false;  //activando esta línea se anula la caja 2.

	estado = passive; // Se inicia el modelo en el estado passive
	nextChange(Time::Inf);  // con tiempo infinito

	return *this ;
}

/*******************************************************************
* Function Name: externalFunction
*******************************************************************/
Model &ColaCajas::externalFunction( const ExternalMessage &msg )
{
```

```
if(estado == passive) {

      if( msg.port() == in ) {
            /* Si llega una entidad al puerto "in", la mete en la
            cola */
            elements.push_back(msg.value());

            /* Planifica una transición interna para este mismo
            instante, para que se active la función de salida */
            estado = sendQ;
            nextChange(Time::Zero);
      }

      if(msg.port() == estado1) {
            /* Si llega una entidad al puerto "estado1"
            actualiza la variable pasivo1 */
            pasivo1 = true;

            /* Planifica una transición interna para este mismo
            instante, para que se active la función de salida */
            estado = sendQ2;
            nextChange(Time::Zero);
      }
      if(msg.port() == estado2) {
            pasivo2 = true;

            estado = sendQ2;
            nextChange(Time::Zero);
      }
}
if(estado == wait) {
      if( msg.port() == in ) {
            /* Si llega una entidad al puerto "in", la mete en la
            cola */
            elements.push_back(msg.value());

            /* Planifica una transición interna para este mismo
            instante, para que se active la función de salida */
            estado = sendQ;
            nextChange(Time::Zero);
      }
      if(msg.port() == estado1) {
            /* Si llega una entidad al puerto "estado1"
            actualiza la variable pasivo1 */
            pasivo1 = true;

            /* Planifica una transición interna para este mismo
            instante, para que se active la función de salida */
            estado = sendQ;
            nextChange(Time::Zero);
      }
      if(msg.port() == estado2) {
            pasivo2 = true;

            estado = sendQ;
            nextChange(Time::Zero);
      }
}
```

```
	else if (estado == busy) {
		if( msg.port() == in ) {
			/* Si llega una entidad al puerto "in", la mete en la
			cola */
			elements.push_back(msg.value());
		}
		if(msg.port() == estado1) {
			/* Si llega una entidad al puerto "estado1"
			actualiza la variable pasivo1 */
			pasivo1 = true;
		}
		if(msg.port() == estado2) {
			pasivo2 = true;
		}
	}
	return *this;
}

/*******************************************************************
* Function Name: internalFunction
********************************************************************/
Model &ColaCajas::internalFunction( const InternalMessage & )
{
    if(estado == sendQ) {
      if((pasivo1 == true) || (pasivo2 == true)) {
		/* Si hay una caja libre, coge la primera entidad de la
		cola */
		valor = elements.front();

		// La elimina de la cola
		elements.pop_front();

		/* Planifica una transición interna para este mismo
		instante, para que se active la función de salida */
		estado = busy;
		nextChange(Time::Zero);
      }
      else {
		/* Si no hay ninguna caja libre pasa al estado wait
		   con tiempo infinito */
		estado = wait;
		nextChange(Time::Inf);
      }
    }
    else if(estado == sendQ2) {
      //passivate();
      estado = passive;
      nextChange(Time::Inf);

    }
    else if(estado == busy) {
      if(!(elements.size() == 0)) {
		if((pasivo1 == true) || (pasivo2 == true)) {
			/* Si la cola no está vacía y hay una caja libre,
			provoca una transición interna en este instante,
			para que se active la función de salida */
			estado = sendQ;
			nextChange(Time::Zero);
		}
```

```
            else {
                  /* Si no hay ninguna caja libre pasa al estado wait
                     con tiempo infinito */
                  estado = wait;
                  nextChange(Time::Inf);
            }
      }
      else  {     //passivate();
            // Si la cola está vacía pasa al estado passivate
            estado = passive;
            nextChange(Time::Inf);
      }
    }
      return *this ;
}
/*******************************************************************
* Function Name: outputFunction
*******************************************************************/
Model &ColaCajas::outputFunction( const InternalMessage &msg )
{
      if((estado == sendQ) || (estado == sendQ2)) {
            /* Si hay alguna caja ocupada, aumenta el tamaño de la
            cola, porque el valor que envía es el del número de
            entidades en el conjunto de SetCajas */
            tcola = elements.size();
            if(pasivo1 == false) tcola = tcola + 1;
            if(pasivo2 == false) tcola = tcola + 1;
            sendOutput( msg.time(), outQ, tcola ) ;
      }
      else if(estado == busy) {
            if((pasivo1 == true) && (pasivo2 == false)) {
                  /* Si sólo está libre la caja1, le envía la entidad y
                  actualiza la variable pasivo1 */
                  sendOutput( msg.time(), out1, valor);
                  pasivo1 = false;
            }
            else if((pasivo1 == false) && (pasivo2 == true)) {
                  sendOutput( msg.time(), out2, valor);
                  pasivo2 = false;
            }
            else if((pasivo1 == true) && (pasivo2 == true)) {
                  /* Si las dos cajas están libres,
                  genera un número aleatorio con la función rand(),
                  y le aplica la operación módulo 2
                  para elegir el puerto de salida */
                  int n ;
                  n = rand() % 2;
                  if(n == 0) {
                        sendOutput( msg.time(), out2, valor);
                        pasivo2 = false;
                  }
                  else {
                        sendOutput( msg.time(), out1, valor);
                        pasivo1 = false;
                  }
            }
      }
      return *this ;
}
```

C.15 cajaSimple.h

```
#ifndef __CAJASIMPLE_H
#define __CAJASIMPLE_H

#include "atomic.h"      // class Atomic
#include "randlib.h"    // class randlib

class CajaSimple : public Atomic
{
public:
	//Default constructor
	CajaSimple( const string &name );

	virtual string className() const ;
protected:
	// Métodos de control
	Model &initFunction();
	Model &externalFunction( const ExternalMessage & );
	Model &internalFunction( const InternalMessage & );
	Model &outputFunction( const InternalMessage & );

private:
	const Port &in;    // Puerto de entrada
	Port &out;         // Puertos de salida
	Port &out1;
	Port &out2;
	Port &out3;
	Port &estado;

	/* Variable para almacenar temporalmente el valor de las
	entidades */
	Value valor;

};	// class CajaSimple

// ** inline ** //
inline
string CajaSimple::className() const
{
	return "CajaSimple" ;
}
#endif   //__CAJASIMPLE_H
```

C.16 cajaSimple.cpp

```
#include "cajaSimple.h" // class CajaSimple
#include "message.h"    // class ExternalMessage, InternalMessage
#include "mainsimu.h"   // MainSimulator::Instance().getParameter(...)

/*******************************************************************
* Function Name: CajaSimple
*******************************************************************/
```

```
CajaSimple::CajaSimple( const string &name )
: Atomic( name )
, in(addInputPort("in"))            // Puerto de entrada
, out( addOutputPort("out"))        //Puertos de salida
, out1( addOutputPort("out1"))
, out2( addOutputPort("out2"))
, out3( addOutputPort("out3"))
, estado( addOutputPort("estado"))
{
}

/*******************************************************************
* Function Name: initFunction
********************************************************************/
Model &CajaSimple::initFunction()
{
      return *this ;
}

/*******************************************************************
* Function Name: externalFunction
********************************************************************/
Model &CajaSimple::externalFunction( const ExternalMessage &msg )
{
      if(state() == passive) {
            if( msg.port() == in ){
                  /* Si llega una entidad al puerto "in",
                  empieza a procesarla */
                  valor = msg.value();
                  holdIn(active, Time(0,0,genunf(30, 120),0));
            }
      }
      return *this;
}

/*******************************************************************
* Function Name: internalFunction
********************************************************************/
Model &CajaSimple::internalFunction( const InternalMessage & )
{
      passivate();

      return *this ;
}

/*******************************************************************
* Function Name: outputFunction
********************************************************************/
Model &CajaSimple::outputFunction( const InternalMessage &msg )
{
      if(state() == active) {
            /* Variable para enviar a colaCajas e indicarle
               que la caja ha quedado libre */
            Value pasivo;

            /* Elimina la cantidad que se añadió al valor de la entidad
            en surtidor, y reconoce al surtidor en el que fue procesada
            la entidad */
```

```
		if((valor - 300000) > 0) {
			/* Si el valor de la entidad es mayor que 300000,
			   se trata del surtidor3 */
			Value val = valor - 300000;
			Value surt = 3;

			/* Envía por "out" hacia trands, y también fuera del
			sistema, la entidad que acaba de procesar */
			sendOutput(msg.time(), out, val);

			/* Envia por "out3" hacia surtidor3,
			una entidad con valor 3 */
			sendOutput(msg.time(), out3, surt);

			/* Envía por "estado" hacia colaCajas,
			un mensaje de que queda libre la caja */
			sendOutput(msg.time(), estado, pasivo);
		}

		else if((valor - 200000) > 0) {
			/* Si ahora es mayor que 200000
			se trata del surtidor2 */
			Value val = valor - 200000;
			Value surt = 2;
			sendOutput(msg.time(), out, val);
			sendOutput(msg.time(), out2, surt);
			sendOutput(msg.time(), estado, pasivo);
		}

		else if((valor - 100000) > 0) {
			/* Si es mayor que 100000
			se trata del surtidor1 */
			Value val = valor - 100000;
			Value surt = 1;
			sendOutput(msg.time(), out, val);
			sendOutput(msg.time(), out1, surt);
			sendOutput(msg.time(), estado, pasivo);
		}
	}

	return *this ;
}
```

C.17 transdCaja.h

```
#ifndef __TRANSDCAJA_H
#define __TRANSDCAJA_H

#include <fstream.h>    // class fstream
#include <list>         // class list
#include <map.h>        // class map
#include "atomic.h"     // class Atomic
```

```
class TransdCaja: public Atomic
{
public:
	//Default constructor
	TransdCaja( const string &name = "TransdCaja" ) ;

	virtual string className() const ;

protected:
	//Métodos de control
	Model &initFunction();
	Model &externalFunction( const ExternalMessage & );
	Model &internalFunction( const InternalMessage & );
	Model &outputFunction( const InternalMessage & );

private:
	typedef map< int, Time, less<int> > JobsList ;
	typedef list<Value> ElementsList;

	const Port &ariv ;	//Puertos de entrada
	const Port &solv ;

	//Variable para almacenar el tiempo de duración
	Time observ_time ;

	/*Variable para almacenar las entidades que llegan
		y sus tiempos de llegada */
	JobsList unsolved ;

	/*Variables para almacenar las entidades que llegan
		por cada uno de los cuatro puertos de entrada */
	ElementsList arrived, solved;

	//Variables auxiliares para los datos estadísticos
	Time parcial_ta, total_ta, tmax;
	double noCola_c, espera_c;

	//Variable para el archivo salida.txt
	ofstream archivo;

	const Time &observation_time() const ;

};	// class TransdCaja

// ** inline ** //
inline
string TransdCaja::className() const
{
	return "TransdCaja" ;
}

inline
const Time &TransdCaja::observation_time() const
{
	return observ_time ;
}

#endif	//__TRANSDCAJA_H
```

C.18 transdCaja.cpp

```
#include "transdCaja.h"      // base class
#include "message.h"         // class ExternalMessage
#include "mainsimu.h"        // class MainSimulator

/*******************************************************************
* Function Name: TransdCaja
* Description: Constructor
********************************************************************/
TransdCaja::TransdCaja( const string &name )
: Atomic( name )
, ariv( addInputPort( "ariv" ) )    // Puertos de entrada
, solv( addInputPort( "solv" ) )
{
      /* Valor por defecto. Será el que utilice este modelo */
      observ_time = "25:0:0:0" ;

      /* Si hay un valor asignado en el archivo del modelo compuesto
      correspondiente, se lo asigna*/
      if( MainSimulator::Instance().
                  existsParameter(description(),"observation_time" ) )
            observ_time = MainSimulator::Instance().
                  getParameter(description(), "observation_time" ) ;
}

/*******************************************************************
* Function Name: initFunction
********************************************************************/
Model &TransdCaja::initFunction()
{
      // Inicia las variables de tipo map y tipo list
      unsolved.erase( unsolved.begin(), unsolved.end() ) ;

      arrived.erase(arrived.begin(), arrived.end());
      solved.erase(solved.begin(), solved.end());

      // Inicializa las variables auxiliares
      total_ta = tmax = Time::Zero;
      noCola_c = 0;
      espera_c = 0;

      /* Abre el archivo salida.txt, y  sitúa  el cursor
      para escribir al final del archivo */
      archivo.open("salida.txt", ios::app);

      /* Inicia el modelo en el estado active con el tiempo
      que almacena la variable observ_time */
      holdIn( active, observation_time() ) ;
      return *this ;
}

/*******************************************************************
* Function Name: externalFunction
********************************************************************/
Model &TransdCaja::externalFunction( const ExternalMessage &msg )
{
```

```
if( msg.port() == ariv )
{
	if( unsolved.find( msg.value() ) != unsolved.end() )
	{
		MException e( string("Unresolved Work Id: ") +
		        msg.value() + " is duplicated." );
		e.addLocation( MEXCEPTION_LOCATION() );
		throw e;
		/*Si encuentra un valor en unsolved igual al de la
		Entidad que acaba de llegar por el puerto arrived,
		lanza un mensaje de error */
	}
	/* Introduce la entidad y el tiempo de su llegada
	en la variable unsolved */
	unsolved[ msg.value() ] = msg.time() ;

	/* Introduce la entidad en la lista arrived */
	arrived.push_back(msg.value());
}
if( msg.port() == solv )
{
	JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

	if( cursor == unsolved.end() )
	{
		MException e( string("Resolved Work Id: ") +
		            msg.value() + " Not Found!" );
		e.addLocation( MEXCEPTION_LOCATION() );
		throw e;
		/* Si no encuentra el valor de la entidad que acaba
		de llegar por el puerto solv,lanza un mensaje de
		error */
	}
	/* Actualiza las variables auxiliares */
	parcial_ta =  msg.time() - unsolved[msg.value()];
	total_ta = total_ta + parcial_ta;

	if(parcial_ta > tmax) {
		/* En esta variable se almacena el tiempo máximo de
		espera en la cola de las cajas */
		tmax = parcial_ta;
	}
	if(parcial_ta == Time::Zero) {
		/* En esta variable se contabilizan las entidades que
		no han esperado en la cola de las cajas */
		noCola_c = noCola_c + 1;
	}
	if(parcial_ta.asMsecs() > 60000) {
		/* En esta variable se contabilizan las entidades que
		han esperado en la cola del surtidor1 más de un
		minuto (60000 milisegundos)*/
		espera_c = espera_c + 1;
	}
	/* Introduce la entidad en la lista solved */
	solved.push_back(msg.value());

	/* Borra la entidad de la variable unsolved */
	unsolved.erase( cursor ) ;
}
```

```
	return *this ;
}

/*******************************************************************
* Function Name: internalFunction
*******************************************************************/
Model &TransdCaja::internalFunction( const InternalMessage &)
{
	/* La transición interna ocurre cuando termina el tiempo
	marcado por la variable observ_time, el modelo pasa al estado
	passive */
	passivate();

	/* Al final de la simulación, imprime en el archivo salida.txt
	los datos estadísticos de la cola de las cajas */

	/* Para redondear a dos decimales se utiliza la expresión:
	numero = double(int(numero*100.0+.5))/100.0; */

	archivo << "Total clientes que han pasado por las Cajas = " <<
			solved.size() << " clientes." << endl;
	archivo << "Total clientes que no han esperado en la cola = " <<
		noCola_c << " clientes. (" <<
		double(int((((noCola_c*100)/solved.size())*100.0 + .5))/
		100.0) << "%)." <<endl;
	archivo << "Total clientes que han esperado más de un minuto = "
		<< espera_c << " clientes. (" <<
		double(int((((espera_c*100)/solved.size())*100.0 + .5))/
		100.0) << "%)." <<endl;
	archivo << "Tiempo máximo de espera en la cola de Cajas = " <<
		double(int((double(int(tmax.asMsecs()/1000))/60)*100.0 +
		.5))/100.0 << " minutos." << endl;
	if(!(solved.size() == 0)) {
		archivo << "Tiempo medio de espera total en la cola de
		Cajas = " <<
		double(int(((double(int(total_ta.asMsecs()/1000))/60)/
		solved.size())*100.0 + .5))/100.0 << " minutos." << endl;
		if((solved.size() - noCola_c)) {
			archivo << "Tiempo medio de espera de los que han
			estado en la cola = " <<
			double(int(((double(int(total_ta.asMsecs()/1000))/60)
			/(solved.size() - noCola_c))*100.0 + .5))/100.0 << "
			minutos." << endl;
		}
	}

	//Cierra el archivo salida.txt
	archivo.close();

	return *this ;
}

/*******************************************************************
* Function Name: outputFunction
*******************************************************************/
Model &TransdCaja::outputFunction( const InternalMessage &msg )

{ }	/* En este modelo no se produce ninguna salida*/
```

C.19 setCajas.ma

```
[top]
%Creación de los componentes. Se crean una instancia de colaCajas,
%dos de cajaSimple y una de transdCaja
components : colaCajas@colaCajas caja1@cajaSimple caja2@cajaSimple
tranCaja@transdCaja

in : in                                   % Puerto de entrada
out : out outEnd1 outEnd2 outEnd3 outQ    % Puertos de salida

% Conexión de los enlaces entre los puertos
Link : in in@colaCajas
Link : in ariv@tranCaja

Link : out1@colaCajas in@caja1
Link : out2@colaCajas in@caja2

Link : estado@caja1 estado1@colaCajas
Link : estado@caja2 estado2@colaCajas

Link : out1@colaCajas solv@tranCaja
Link : out2@colaCajas solv@tranCaja

Link : outQ@colaCajas outQ

Link : out@caja1 out
Link : out@caja2 out

Link : out1@caja1 outEnd1
Link : out2@caja1 outEnd2
Link : out3@caja1 outEnd3

Link : out1@caja2 outEnd1
Link : out2@caja2 outEnd2
Link : out3@caja2 outEnd3
```

C.20 netGasolinera.ma

```
[top]
%Creación de los componentes. Se crean una instancia de entrada,
%tres de surtidor, una de transdSurt y se declaran
%los modelos acoplados ef y setCajas
components : entrada@entrada surtidor1@surtidor surtidor2@surtidor
surtidor3@surtidor tranSurt@transdSurt ef setCajas

in : in              % Puerto de entrada
out : out            % Puerto de salida

% Conexión de los enlaces entre los puertos
Link : out1@entrada ariv@tranSurt
Link : out2@entrada ariv@tranSurt
Link : out3@entrada ariv@tranSurt
```

```
Link : transd@surtidor1 solved1@tranSurt
Link : transd@surtidor2 solved2@tranSurt
Link : transd@surtidor3 solved3@tranSurt

Link : out@ef in@entrada
Link : in in@entrada

Link : out1@entrada in@surtidor1
Link : out2@entrada in@surtidor2
Link : out3@entrada in@surtidor3

Link : out@surtidor1 in@setCajas
Link : outQ@surtidor1 inQ1@entrada
Link : out@surtidor2 in@setCajas
Link : outQ@surtidor2 inQ2@entrada
Link : out@surtidor3 in@setCajas
Link : outQ@surtidor3 inQ3@entrada

Link : outEnd1@setCajas inEnd@surtidor1
Link : outEnd2@setCajas inEnd@surtidor2
Link : outEnd3@setCajas inEnd@surtidor3

Link : out@setCajas out
Link : out@setCajas in@ef


%Código del modelo acoplado ef
[ef]
components : gen@generador transd@transd

in : in
out : out

Link : in solved@transd
Link : out@transd stop@gen
Link : out@gen arrived@transd
Link : out@gen out

[gen]
distribution : exponential
mean : 300


%Código del modelo acoplado setCajas
[setCajas]
components : colaCajas@colaCajas caja1@cajaSimple caja2@cajaSimple
tranCaja@transdCaja

in : in
out : out outEnd1 outEnd2 outEnd3 outQ

Link : in in@colaCajas
Link : in ariv@tranCaja

Link : out1@colaCajas in@caja1
Link : out2@colaCajas in@caja2
```

```
Link : estado@caja1 estado1@colaCajas
Link : estado@caja2 estado2@colaCajas

Link : out1@colaCajas solv@tranCaja
Link : out2@colaCajas solv@tranCaja

Link : outQ@colaCajas outQ

Link : out@caja1 out
Link : out@caja2 out

Link : out1@caja1 outEnd1
Link : out2@caja1 outEnd2
Link : out3@caja1 outEnd3

Link : out1@caja2 outEnd1
Link : out2@caja2 outEnd2
Link : out3@caja2 outEnd3
```

Lista de Referencias y Bibliografía

[Breitenec09] Katharina Breitenecker, Helmuth Böck, Mario Villa. *"Modelling Conservation Quantities using Cellular Automata"* Proceedings of 6th Vienna International Conference on Mathematical Modelling (MATHMOD 2009), Vienna, Austria, Full Papers CD Volume. pp 2584 – 2587. (2009)

[Eckel03] Bruce Eckel. *"Piensa en Java"*. 2ª Edición. Prentice Hall. (2003)

[Shannon76] Robert Shannon, James D. Johannes. *"Systems simulation: the art and science"* IEEE Transactions on Systems, Man and Cybernetics. Vol.6(10). pp. 723-724. (1976)

[Tecnum00] Varios autores. *"Aprenda Java como si estuviera en primero"*. Campus Tecnológico de la Universidad de Navarra. (2000). *http://www.tecnun.es/asignaturas/Informat1/ayudainf/aprendainf/Java/Java2.pdf*

[Tecnum04] Varios autores. *"Aprenda C++ Básico como si estuviera en primero"*. Campus Tecnológico de la Universidad de Navarra. (2004). *http://www.tecnun.es/asignaturas/Informat1/ayudainf/aprendainf/Cpp/basico/cppbasico.pdf*

[Urquía03] Alfonso Urquía. *"Simulación. Texto Base de Teoría".* Texto base de la asignatura Simulación. 3º curso Ingeniería Técnica en Informática de Gestión. UNED. Curso 2003/04. (2003) *http://www.uned.es/543072/Files/SimulacionTeoria_2009_10.pdf*

[Urquía08] Alfonso Urquía. *"Modelado de Sistemas mediante DEVS. Teoría y práctica"*. Texto base de la asignatura Modelado de Sistemas Discretos. 5º curso ETS Ingeniería Informática. UNED. Curso 2008/09. (2008) *www.euclides.dia.uned.es/aurquia/Files/textoBase_MSD_2008_09.pdf*

[Wainer96] Gabriel A. Wainer. "*Introducción a la Simulación de Eventos Discretos*". (1996). *www.sce.carleton.ca/faculty/wainer/papers/96-005.ps*

[Wainer05] Gabriel A. Wainer. *"CD++. A tool for DEVS and Cell-DEVS Modeling and Simulation. User's Guide."* (2005). *http://cell-devs.sce.carleton.ca*

[Zeigler76] Bernard P. Zeigler. *"Theory of Modelling and Simulation".* Wiley Interscience. (1976).

[Zeigler00] Bernard P. Zeigler, Tag Gon Kim, Herbert Praehofer. *"Theory of Modelling and Simulation".* Academic Press, New York. (2000).

[Zeigler05] Bernard P. Zeigler, Hessam S. Sarjoughian. *"Introduction to DEVS Modeling and Simulation with JAVA: Developing Comonent-Based Simulation Models"*. (2005). *http://www.acims.arizona.edu/PUBLICATIONS/publications.shtml*

Siglas, Abreviaturas y Acrónimos

BNF	Backus Normal Form (también Backus Naur Form). Notación empleada para expresar gramáticas. Es una manera formal de describir lenguajes formales.
CD++	Implementación de DEVS desarrollada por la Universidad de Carleton, Ottawa, Canadá.
DEDS	Discrete Events Dynamic Systems. Sistemas Dinámicos de Eventos Discretos.
DEVS	Discrete EVent system Specification. Formalismo para el Modelado de Sistemas.
DEVSJAVA	Implementación de DEVS desarrollada por la Universidad de Arizona, EEUU.
JDEVS	Implementación de DEVS desarrollada por la Universidad de Córcega, Italia.