

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



2003

Representación gráfica de modelos DEVS y modificaciones a CD++ para su simulación

Autores

Alejandro Dobniewski y Gastón Christen

Director

Dr. Gabriel Wainer

Abstract.....	2
1 Introducción	3
2 El formalismo DEVS	4
2.1 DEVS clásico	4
2.1.1 Definiciones previas.....	4
2.1.2 Modelos DEVS atómicos	4
2.1.3 Modelos DEVS atómicos con puertos.....	6
2.1.4 Modelos DEVS Acoplados:	7
2.2 Modelos DEVS Paralelo	8
3 Simulador abstracto DEVS.....	10
3.1 Características del simulador abstracto	10
3.2 Mensajes.....	11
3.3 Simulador para DEVS Paralelo	12
4 La aplicación CD++	16
5 GGAD, una nueva representación de modelos DEVS	17
5.1 Motivación	17
5.2 Modelos GGAD	20
5.3 Lenguaje GADscript	21
5.4 Representación grafica	26
5.5 Relación entre DEVS y GGAD.....	27
5.6 Traducción de representación gráfica a GADscript.....	28
5.7 Compatibilidad e interacción con modelos atómicos DEVS clásicos	30
6 Implementación de modelos atómicos GGAD en CD++	31
6.1 Consideraciones de diseño	31
6.2 Diagrama de clases.....	33
6.3 Algoritmos para interpretar modelos GGAD	33
6.4 Log de eventos	33
6.5 Funciones nativas	34
7 Ejemplos.....	35
7.1 Ascensor simple	35
7.1.1 Descripción.....	35
7.1.2 Modelos.....	35
7.1.3 Simulación.....	39
7.2 Alternating Bit Protocol	40
7.2.1 Modelo Sender	41
7.2.2 Modelo Receiver	42
7.2.3 Simulación.....	43
7.3 Load balancer:	45
8 Conclusiones y desarrollo futuro	57

Abstract

DEVS es un formalismo universal para modelar sistemas de eventos discretos. La aplicación CD++ implementa este formalismo en C++ y permite la definición de modelos proveyendo una interfase para la programación en C++ de los componentes básicos o modelos atómicos DEVS.

Con el objetivo de acercar la simulación de modelos DEVS a un público que no domina el lenguaje de programación C++ definimos una representación gráfica y textual de los distintos componentes de un modelo y su interacción con el sistema.

Adicionalmente implementamos esta especificación con una aplicación que permite generar gráficamente los modelos y extendimos el motor de simulación CD++ para permitir su interpretación y ejecución.

El conjunto de aplicaciones resultante permite crear modelos DEVS de una forma más intuitiva y accesible sin por ello resignar flexibilidad o poder expresivo.

DEVS is an universal formalism for modeling discrete event systems. CD++ implements this formalism in C++ and allows the definition of models providing an interface for coding the basic components or atomic models in C++.

We define a graphical and textual representation of the different components of a model and its interaction with the system with the goal of bringing simulation to a public not proficient in C++ programming.

Additionally we implemented this specification with an application capable of graphically generate models, and we extended the simulation engine CD++ to allow its parsing and execution.

The resultant toolkit allows the creation of DEVS models in a more intuitive and accessible way without resigning flexibility or expressive power.

1 Introducción

La simulación es una técnica de modelado en la cual la relación causa/efecto de un sistema es capturada en el modelo que lo representa, permitiendo entonces a dicho modelo generar el mismo comportamiento que ocurriría en el sistema real. La simulación de modelos debe ser vista como una herramienta que permite experimentar con distintas alternativas o estrategias para luego analizar que impacto tienen esas decisiones sobre el comportamiento global del sistema

En la vida diaria podemos observar gran cantidad de sistemas dinámicos, como por ejemplo, las redes de computadoras, sistemas de producción automatizados, de control de tráfico aéreo, y en general sistemas de comando, de control, de información, o de comunicaciones, entre otros.

Todas las actividades en estos sistemas se deben a la ocurrencia asincrónica de eventos discretos, algunos controlados (tales como el pulsado de una tecla) y otros no (como la falla espontánea de un equipo). Esta característica es la que lleva a definir el término de Sistemas de Eventos Discretos.

Las herramientas matemáticas que hoy disponemos fueron desarrolladas durante los últimos doscientos años para modelar y analizar los procesos conducidos por el tiempo que generalmente uno encuentra en la naturaleza. El proceso de adaptar estas herramientas y desarrollar nuevas para los sistemas conducidos por eventos tiene solo unos pocos años. Por este motivo, encontramos en la teoría de los sistemas de eventos discretos no sólo una serie de herramienta específicas para atacar problemas de modelización, simulación y análisis de sistemas altamente ligados a la práctica de la ingeniería y a los problemas de la informática, sino también un campo fértil para el desarrollo de nuevas técnicas y teorías debido a la cantidad de problemas aún abiertos en el área.

Dentro de los formalismos mas populares de representación de sistemas de eventos discretos (DES) están las Redes de Petri, las Statecharts, Grafcet, Grafos de Eventos y muchas generalizaciones y particularizaciones de los mismos.

Orientado a los problemas de modelización y simulación de DES, en la década del 70, el matemático Bernard Zeigler propuso un formalismo general para la representación de dichos sistemas. Este formalismo, denominado DEVS (Discrete EVENT System specification), es de hecho el formalismo más general para el tratamiento de DES. El hecho de estar fundado en la base de la teoría de sistemas, lo convierte en un formalismo universal, y por lo tanto, todos los otros formalismos mencionados en el párrafo anterior pueden ser absorbidos por DEVS (es decir, todos los modelos representables en dichos formalismos pueden ser representados en DEVS).

Mas aún, problemas más generales que la modelización y simulación, como ser análisis y diseño, no solo ya en DES, sino también en Sistemas Híbridos, pueden ser abordados a partir del formalismo DEVS.

2 El formalismo DEVS

DEVS es un formalismo universal para modelar y simular sistemas donde las variables son discretas a tiempo continuo. Puede verse como una forma de especificar sistemas cuyas entradas, estados y salidas son constantes en segmentos, y cuyas transiciones se identifican como eventos discretos.

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, lo que trae algunas ventajas: en los formalismos con una única granularidad es difícil describir los modelos donde hay muchos procesos operando en distintas escalas de tiempo, y la simulación no es eficiente ya que los estados deben actualizarse en el momento con menor incremento de tiempo, lo cual desperdicia tiempo cuando se aplica a los procesos más lentos.

Un modelo DEVS se construye en base a un conjunto de modelos básicos (atómicos), que se combinan para formar modelos acoplados. Los modelos atómicos son objetos independientes modulares, con variables de estado y parámetros, funciones de transición internas, externas, de salida y avance de tiempo. Un modelo acoplado especifica cómo se conectan las entradas y salidas de los componentes. Los nuevos modelos también pueden usarse para armar modelos de mayor nivel, creando jerarquías de modelos.

2.1 DEVS clásico

2.1.1 Definiciones previas

Una *base de tiempo* es una estructura de la forma

$$\text{time} = (T, <)$$

donde T es un conjunto y $<$ es una relación de orden sobre los elementos de T . $<$ es transitiva, irreflexiva y antisimétrica. $<$ puede ser total o parcial.

Se define como *trayectoria* (o *señal*) una función de la forma $f: T \rightarrow A$ donde T es una base de tiempo. Una restricción de f a un intervalo temporal $\langle t_1, t_2 \rangle$ es llamada *segmento* y se escribe:

$$w: \langle t_1, t_2 \rangle \rightarrow A \text{ o } w \langle t_1, t_2 \rangle$$

El conjunto de todos los segmentos sobre A y T se denota como (A, T) .

Un segmento $w: \langle t_1, t_2 \rangle \rightarrow R$ sobre una base de tiempo continua es llamado continuo si es continuo en todos los puntos $t \in \langle t_1, t_2 \rangle$. Un segmento continuo en piezas (piecewise) es continuo en todos los puntos excepto un número finito de puntos $t' \in \langle t_1, t_2 \rangle$.

Un segmento de eventos representa una secuencia de eventos ordenada en el tiempo. Sea $w: \langle t_1, t_2 \rangle \rightarrow A$ un segmento sobre una base de tiempo continua y un conjunto arbitrario $A = \{ \dots \}$. Entonces w es un segmento de eventos si hay un número finito de puntos en el tiempo $t_1, t_2, \dots, t_{n-1} \in \langle t_1, t_2 \rangle$ tales que $w(t_i) = a_i \in A$ para $i=1, \dots, n-1$ y $w(t) = \emptyset$ para todo otro $t \in \langle t_1, t_2 \rangle$. \emptyset representa el evento nulo y no puede pertenecer a A .

Los segmentos de eventos son utilizados en los modelos de eventos discretos como trayectorias de entrada y salida.

2.1.2 Modelos DEVS atómicos

Un modelo DEVS está representado por una estructura.

$$M = \langle X, S, Y, \Gamma_{int}, \Gamma_{ext}, O, \tau_a \rangle$$

donde:

X : Conjunto de valores de entrada.

Y: Conjunto de valores de salida.

S: Conjunto de estados secuencial.

Γ_{int} : So S Función de transición interna.

Γ_{ext} : $Q \times X \rightarrow S$ Función de transición externa, donde $Q = \{(s,e)/s \in S, 0 \leq e \leq ta(s)\}$

Q So Y Función de salida.

$ta: S \rightarrow R_0^+ \cup \{\phi\}$. Función de avance de tiempo (time advance).

Estos elementos son interpretados de la siguiente manera:

En todo momento el sistema esta en un estado $s \in S$. Si no ocurre ningún evento externo el sistema permanecerá en el estado s por el tiempo $ta(s)$, siendo $ta(s)$ un número real, cero o infinito. En el caso que $ta(s)$ sea 0, decimos que s es un estado transitorio dado que por el tiempo nulo no pueden intervenir eventos externos. En el otro caso, cuando $ta(s)$ es infinito, el sistema esperará por siempre hasta que un evento externo lo interrumpa; decimos que s es un estado pasivo.

Cuando el tiempo transcurrido es igual a $ta(s)$ el sistema emite el valor $Q(s)$ y cambia el estado a $\Gamma_{int}(s)$. Cabe destacar que las salidas del sistema unicamente pueden ocurrir inmediatamente antes de las transiciones internas. Por lo tanto, no es posible que un evento externo ocasione una salida. Para ese caso es necesario agregar un estado con $ta(s)$ igual a cero.

Si un evento externo $x \in X$ ocurre antes del tiempo limite, es decir cuando el sistema esta en un estado (s,e) con $e < ta(s)$, el sistema cambia al estado $\Gamma_{ext}(s, e, x)$.

Resumiendo, la función de transición interna indica el nuevo estado del sistema cuando no ocurren eventos externos desde la última transición y la función de transición externa indica el nuevo estado del sistema cuando un evento externo ocurre, este nuevo estado es determinado por la entrada x el estado actual s y por cuanto tiempo el sistema ha permanecido en este estado e . En ambos casos el sistema esta en un nuevo estado s' con un nuevo tiempo limite $ta(s')$.

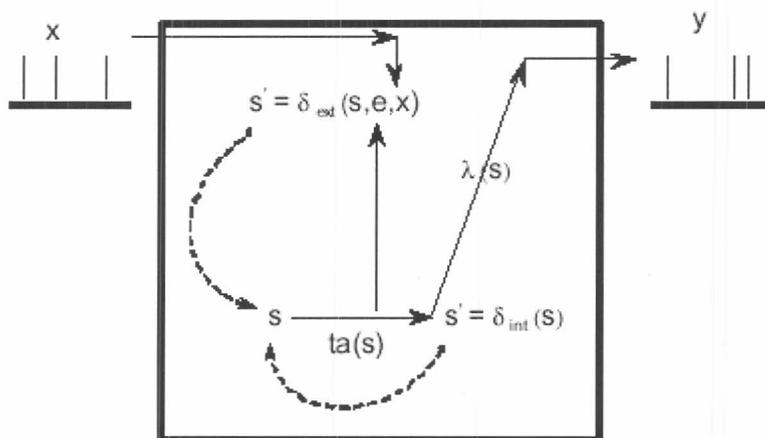


Figura 1 Funcionamiento de un modelo DEVS

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

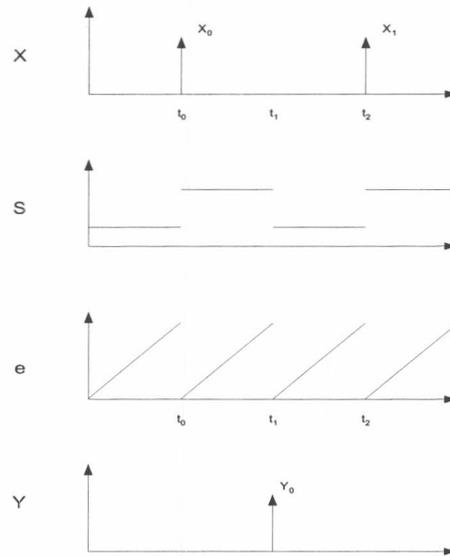


Figura 2 Trayectorias

Aquí la trayectoria de entrada es una serie de eventos que ocurren a intervalos de tiempo tales como t_0 y t_2 . En medio de estos tiempos de arribo de eventos puede haber otros tales como t_1 que son tiempos de arribo de eventos internos. Los últimos son distinguibles en la trayectoria de estado, la cual es una serie de segmentos que representan los estados que van cambiando según los eventos externos e internos. La trayectoria de tiempo transcurrido es una serie de segmentos diagonales que muestran el flujo del tiempo en un reloj que mide el tiempo transcurrido y es puesto a cero ante la ocurrencia de cada evento.

Finalmente la trayectoria de salida muestra los eventos de salida que son producidos por la función de salida inmediatamente antes de aplicar la función de transición interna. Es interesante notar que la función de salida no altera el estado del modelo.

2.1.3 Modelos DEVS atómicos con puertos

La incorporación de puertos de entrada y salida a los modelos DEVS simplifica la representación de los mismos. Ahora tenemos un nuevo modelo compuesto por:

$$M = \langle X, S, Y, \Gamma_{int}, \Gamma_{ext}, O, ta \rangle$$

donde:

$X = \{(p, v) / p \text{ Inports, } v \in X_p\}$ Conjunto de puertos de entrada y valores

$Y = \{(p, v) / p \text{ Outports, } v \in Y_p\}$ Conjunto de puertos de salida y valores

S : Conjunto de estado secuencial.

Γ_{int} : So S Función de transición interna.

Γ_{ext} : $Q_x \times X_o \times S$ Función de transición externa, donde $Q = \{(s, e) / s \in S, 0 \leq e \leq ta(s)\}$

O : So Y Función de salida.

$ta: S \rightarrow R_0^+ \cup \phi$. Función de avance de tiempo (time advance).

En los modelos DEVS clásicos, solo un puerto recibe un valor durante un evento externo.

2.1.4 Modelos DEVS Acoplados:

El formalismo DEVS permite construir modelos a partir de componentes. La especificación en el caso de modelos DEVS con ports incluye la interfaz externa (ports de entrada y salida y sus valores), los componentes (que deben ser a su vez modelos DEVS) y las relaciones de acoplamiento.

Un modelo acoplado DEVS se define como:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select \rangle$$

Donde:

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ es el conjunto de puertos de entrada y sus valores;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ es el conjunto de puertos de salida y sus valores;

D es el conjunto de nombres de componentes;

Las siguientes restricciones se aplican a los componentes:

Los componentes son modelos DEVS:

para cada $d \in D$

$M_d = (X_d, Y_d, S, \Gamma_{ext}, \Gamma_{int}, \Gamma_{con}, Q, ta)$ es un modelo DEVS básico

$X_d = \{(p,v) \mid p \in IPorts, v \in X_p\}$;

$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p\}$;

Los acoplamientos cumplen las siguientes condiciones:

ξ *acoplamientos de entradas externas (EIC)* conectan las entradas externas del modelo a las entradas de los componentes.

$EIC = \{(N, ip_N), (d, ip_d) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$

ξ *acoplamientos de salidas externas (EOC)* conectan las salidas de los componentes a las salidas externas del modelo:

$EOC = \{(d, op_d), (N, op_N) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$

ξ *acoplamientos internos (IC)* conectan las salidas de los componentes a las entradas de los componentes del modelo:

$IC = \{(a, op_a), (b, ip_b) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$

No se permiten loops (el puerto de salida de un componente no puede estar conectado al puerto de entrada del mismo componente).

$((d, op_d), (e, ip_e)) \in IC$ implica $d \neq e$.

ξ $Select: 2^D - \{\emptyset\} \rightarrow D$, la función de desempate

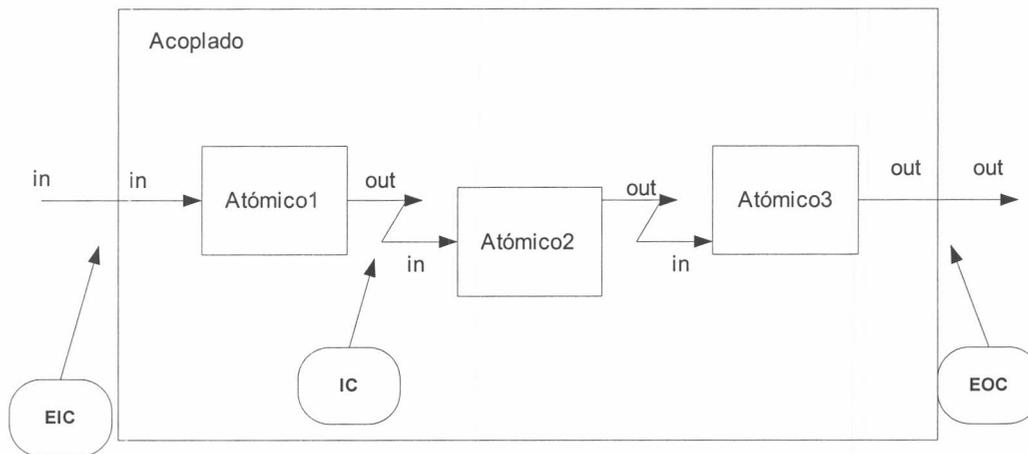


Figura 3 Modelo DEVS Acoplado

2.2 Modelos DEVS Paralelo

El formalismo DEVS Paralelo [Cho94b] mantiene las propiedades útiles del DEVS Clásico y elimina las restricciones de serialización que impiden la ejecución múltiple en un entorno paralelo.

Chow requiere que las siguientes propiedades se mantengan:

- ξ Manejo de colisiones: el comportamiento en caso de colisión debe ser controlable por el modelador.
- ξ Paralelismo: el formalismo no debe usar ninguna función de serialización que prohíba posibles concurrencias.
- ξ Uniformidad: la construcción jerárquica debe tener comportamiento uniforme: diferentes construcciones jerárquicas del mismo modelo deben exhibir el mismo comportamiento.

Los modelos atómicos son la construcción más básica, la cual puede ser combinada con otros modelos en modelos acoplados. Los modelos acoplados DEVS paralelo pueden ser usados de la misma manera que los modelos atómicos. Por lo tanto pueden construirse modelos en forma jerárquica como en DEVS Clásico.

Un modelo DEVS Paralelo es descrito de la siguiente manera:

$$DEVS = \langle X_M, Y_M, S, \Gamma_{ext}, \Gamma_{int}, \Gamma_{con}, Q, ta \rangle$$

donde:

$X_M = \{(p,v) \mid p \in IPorts, v \in X_p\}$ es el conjunto de pares de puerto de entrada y el valor recibido;

$Y_M = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ es el conjunto de pares de puerto de salida y el valor emitido;

S es el conjunto de estados secuenciales;

$\Gamma_{ext}: Q \times X_M^b \rightarrow S$ es la función de transición externa;

$\Gamma_{int}: S \rightarrow S$ es la función de transición interna;

$\Gamma_{con}: Q \times X_M^b \rightarrow S$ es la función de confluencia;

$O: S \rightarrow Y_M^b$ es la función de salida;

$ta : S \rightarrow R_0^+ \cup \phi$ es la función de time advance;

donde $Q := \{ (s, e) \mid s \in S, 0 \leq e \leq ta(s) \}$ es el conjunto de *estados totales*.

Diferencias entre DEVS Clásico y DEVS Paralelo:

- ξ La interfase del modelo incluye puertos y valores.
- ξ Las funciones de transición externa y salida pueden recibir más de un evento simultáneamente, porque han sido extendidas para recibir bolsas de eventos.
- ξ Se agrega una nueva función de confluencia. Esta función permite desempatar entre la función de transición externa y la interna cuando hay eventos de ambas clases simultáneamente.

La semántica de los modelos DEVS Paralelo se define como sigue:

En un instante dado un modelo básico esta en el estado s . Mientras no ocurran eventos externos permanece en ese estado por un periodo de tiempo $ta(s)$. Cuando ocurre una transición interna el sistema emite el valor $Q(s)$ y cambia al estado $\Gamma_{int}(s)$. Si uno o mas eventos externos $E = \{ x_1 \dots x_n \mid x \in X_M \}$ ocurren antes que el tiempo $ta(s)$ transcurra, o sea cuando el sistema esta en el estado total (s, e) con $e \leq ta(s)$, el nuevo estado estará dado por $\Gamma_{ext}(s, e, E)$. Cuando una transición externa y una transición interna coinciden, es decir que evento externo E ocurre cuando $e = ta(s)$, el nuevo estado del sistema podría estar dado por $\Gamma_{ext}(\Gamma_{int}(s), e, E)$ o $\Gamma_{int}(\Gamma_{ext}(s, e, E))$. Para evitar un comportamiento fijo, se puede definir el comportamiento más adecuado con la función Γ_{con} . Por lo tanto en el formalismo DEVS Paralelo, en presencia de colisiones, el nuevo estado del sistema será el definido por $\Gamma_{con}(s, E)$.

Un modelo acoplado DEVS Paralelo se define como:

$$CM = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC \rangle$$

donde

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ es el conjunto de puertos de entrada y sus valores;

$Y = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ es el conjunto de puertos de salida y sus valores;

D es el conjunto de nombres de componentes;

Las siguientes restricciones se aplican a los componentes:

Los componentes son modelos DEVS:

para cada $d \in D$

$M_d = (X_d, Y_d, S, \Gamma_{ext}, \Gamma_{int}, \Gamma_{con}, Q, ta)$ es un modelo DEVS básico

$X_d = \{(p, v) \mid p \in IPorts, v \in X_p\}$;

$Y_d = \{(p, v) \mid p \in OPorts, v \in Y_p\}$;

Los acoplamientos cumplen las siguientes condiciones:

- ξ *acoplamientos de entradas externas (EIC)* conectan las entradas externas del modelo a las entradas de los componentes.

$EIC = \{(N, ip_N), (d, ip_d) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$

- ξ *acoplamientos de salidas externas (EOC)* conectan las salidas de los componentes a las salidas externas del modelo:

$EOC = \{(d, op_d), (N, op_N) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$

ξ *acoplamientos internos (IC)* conectas las salidas de los componentes a las entradas de los componentes del modelo:

$$IC \quad \{(a, op_a), (b, ip_b) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$$

No se permiten loops (el puerto de salida de un componente no puede estar conectado al puerto de entrada del mismo componente).

$$((d, op_d), (e, ip_d)) \quad IC \text{ implica } d \zeta e.$$

ξ Restricciones de inclusión de rango: los valores enviados desde un puerto de origen deben estar en el rango de valores aceptados del puerto de destino.

$$((N, ip_N), (d, ip_d)) \quad EIC : X_{ipN} \subseteq X_{ipd}$$

$$((a, op_a), (N, op_N)) \quad EOC : Y_{opa} \subseteq Y_{opN}$$

$$((a, op_a), (b, ip_b)) \quad IC : Y_{opa} \subseteq X_{ipb}$$

3 Simulador abstracto DEVS

El objetivo de un simulador es generar los estados y trayectorias de salida para un modelo dadas los segmentos de entrada y estado inicial del mismo. Un simulador abstracto es una descripción algorítmica de como llevar a cabo las instrucciones implícitas en los modelos DEVS para generar su comportamiento. Para los modelos DEVS existe un simulador abstracto genérico que permite implementar simuladores basados en distintos formalismos [Zei00].

En el simulador abstracto se definen tres clases de entidades:

- ξ Simulador: trata los modelos atómicos. A cada modelo atómico se le asocia un simulador con el rol de mandar la ejecución de distintas funciones del modelo.
- ξ Coordinador: trata los modelos acoplados. A cada modelo acoplado se asocia un coordinador con el rol de comandar los controladores de los modelos que constituyen el acoplado.
- ξ Coordinador raíz: genera la simulación ligada a los coordinadores de mayor nivel.

Se denominan genéricamente procesadores a los simuladores y coordinadores.

3.1 Características del simulador abstracto

- ξ Cada modelo básico de un formalismo es implementado por una clase propia.
- ξ Cada clase de modelo atómico tiene su propia clase que la simula.
- ξ Un modelo acoplado de cada formalismo es soportado por una clase de modelos acoplados.
- ξ Cada clase de modelo acoplado tiene una clase simuladora llamada coordinador.
- ξ Los simuladores utilizan un protocolo basado en mensajes que permite coordinar la simulación con otros objetos.
- ξ Los simuladores y coordinadores siguen la jerarquía de la estructura del modelo. Los simuladores manejan los componentes al nivel atómico y los coordinadores los niveles sucesivos hasta la raíz del modelo.

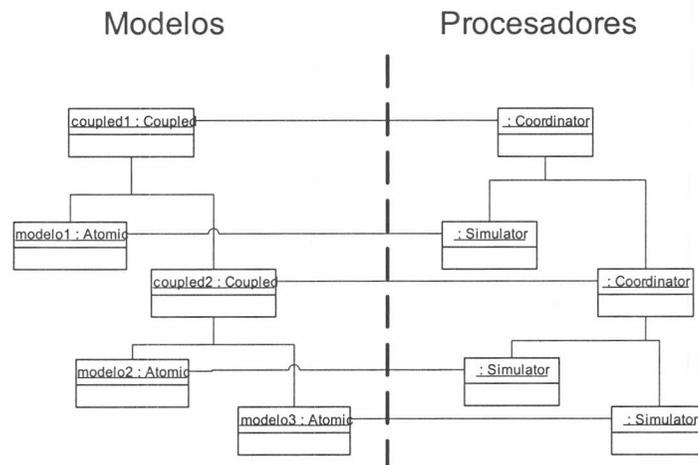


Figura 4 Relación entre modelos y simuladores

Un modelo atómico es manejado por un simulador que recibe los mensajes correspondientes a los eventos internos y externos respectivamente. A su vez el simulador ejecuta la función de transición correspondiente en el modelo.

Este diseño permite transformar en forma directa una especificación jerárquica de modelos DEVS directamente a código ejecutable. Otra importante ventaja es el soporte de interoperatividad entre formalismos. La interacción entre los componentes del sistema se realiza a través de protocolo de mensajes que permite que la implementación de un formalismo sea encapsulada en un simulador, aparentando ser simuladores DEVS estándar a su coordinador. Este concepto de simulación multiformalismo es llamado DEVS-Bus.

3.2 Mensajes

La simulación se realiza por pasaje de mensajes entre los simuladores y coordinadores. Los mensajes pueden ser de eventos internos, eventos externos, salida o de sincronización de la simulación. Los mensajes contienen datos como el origen, instante de tiempo del evento y puerto de destino y valor.

Mensaje X

Indica un evento externo. Un simulador que recibe este mensaje llama a la función de transición externa del componente considerada, y responde con un mensaje **listo** que indica que la transición de estado ha terminado y la hora prevista del próximo evento interno. Un coordinador lo reenvía al simulador apropiado.

Mensaje *

Indica un evento interno. El simulador invoca la función de salida y luego a la transición interna del modelo, generando un mensaje Y y un mensaje listo. Si lo recibe un coordinador lo transmite a su hijo con el menor tiempo de próximo evento.

Mensaje Y

Los mensaje Y son generados por la función de salida de un modelo atómico. Cuando un coordinador recibe un mensaje Y de su hijo inminente¹, consulta el esquema de acoplamiento de salida externa para ver si debe ser transmitido a su padre, y su esquema de acoplamiento interno para obtener el hijo y sus respectivos ports de entrada al cual hay que mandar el mensaje.

Mensaje listo

¹ Inminente: que tiene mínimo tiempo al próximo evento.

Un mensaje **listo** indica al padre que su hijo terminó con su tarea. Cuando un coordinador ha recibido los mensajes **listo** de todas las influencias (en el caso de un mensaje Y ascendente) o los receptores (en el caso de mensaje X descendente), calcula el mínimo de sus hijos (lista de tiempos de próximos eventos) y determina su nuevo hijo inminente para cuando se reciba el siguiente mensaje *. También envía este nuevo mínimo como la hora de su próximo evento interno en un mensaje listo a su padre.

3.3 *Simulador para DEVS Paralelo*

En [Cho94b] se introduce un simulador abstracto para DEVS Paralelo. Este simulador es apropiado para procesamiento paralelo, pero como no distingue entre mensajes entre procesos de mensajes dentro de un proceso, no lo es para procesamiento distribuido. Como los hijos de un coordinador pueden estar distribuidos, enviar un mensaje a los hijos puede implicar enviar mensajes a todos los nodos de la red. El costo de envío de mensajes puede ser significativo cuando es distribuido (Figura 5).

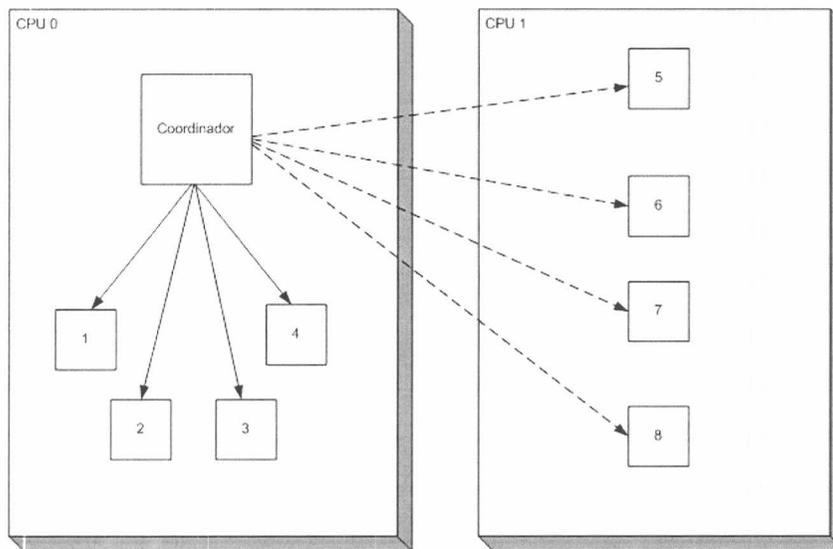


Figura 5 Un coordinador envía mensajes a sus hijos.

En [Tro01] se muestra un simulador para procesamiento distribuido utilizando un coordinador por cada proceso y modelo acoplado. Para minimizar el pasaje de mensajes entre procesos se introduce el concepto de coordinadores maestros y esclavos. Solo uno de los coordinadores de un modelo acoplado envía y recibe mensajes del padre del modelo. Ese coordinador es llamado maestro. Los demás coordinadores de ese modelo se comunican con el maestro y se llaman esclavos (Figura 6).

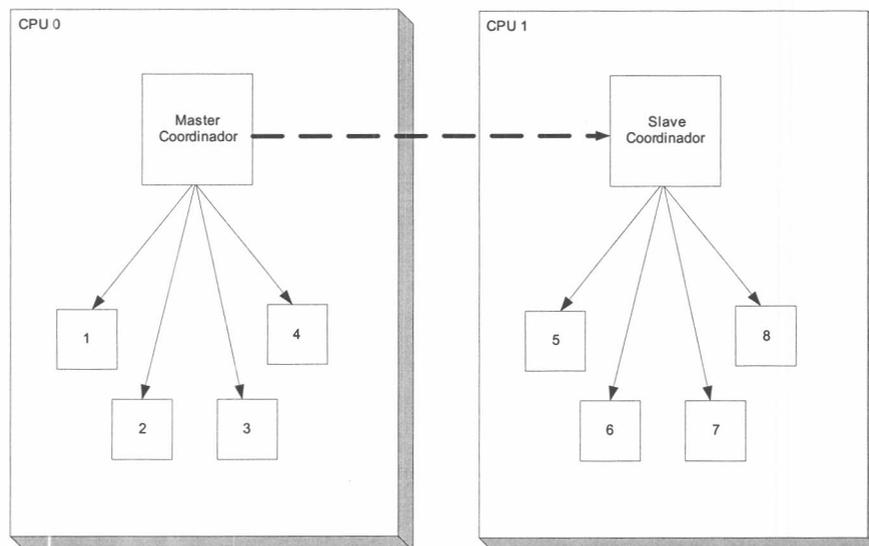


Figura 6 Un maestro envía un mensaje a través del esclavo.

Se asume que los mensajes preservan su ordenamiento original.

A continuación se muestran los algoritmos del simulador abstracto.

Simulador Devs Paralelo

```

SIMULATOR
when a ( @ , t ) message is received
    if t = tN then
        y := Q(s)
        send ( y , t ) to the parent coordinator
        send ( done , t ) to the parent coordinator
    end if
    else raise error
end when
when a ( q , t ) message is received
    lock the bag
    Add event q to the bag
    unlock the bag
end when
when a ( * , t ) message is received
    case tL δ t < tN
        e := t - tL
        s := Γext( s , e , bag )
        empty bag
    end case
    case t = tN and bag is empty
        s := Γint( s )
    end case
    case t = tN and bag not is empty
        s := Γcon( s , bag )
        empty bag
    end case
    case t > tN or t < tL
        raise error
    end case
    tL := t
    tN := ta ( s )
    send ( done , tN ) to parent coordinator
end when
    
```

Función auxiliar coordinador

coordinator : $M \times P \rightarrow C$

where

M is a coupled model

P is a DEVS processor

S is a *coordinator* (*master or slave*)

coordinator (M, j) = i , where i is the *coordinator* associated to coupled M that is local to child j . The following restrictions apply for the function to be well defined:

j is a DEVS processor associated to a dependant of M

i is one of the *coordinators* associated with M

Coordinador Maestro

MASTER COORDINATOR

when a ($@, t$) message is received from parent coordinator

if $t = t_N$ **then**

$t_L := t$

for all imminent child processors i with minimum t_N

send ($@, t$) to child i

cache i in the *synchronize* set

end for

wait until ($done, t$)'s have been received from all imminent processors

send ($done, t$) to parent coordinator

end if

else raise error

end when

when a (y, t) message is received from child i

for all influencees, j of child i

if j is a local processor

$q := z_{i,j}(y)$

send (q, t) to child j

cache j in the *synchronize* set

else

$s := \text{coordinator}(self, j)$

if s *slave-sync* set **then**

send (y, i, t) to s

cache s in the *slave-sync* set

cache s in the *synchronize* set

end if

end if

end for

if $self$ I_i (y is to be transmitted upward) **then**

$y := z_{i, self}(y)$

send (y, t) to parent coordinator

end if

clear *slave-sync* set

end when

when a (y, i, t) message is received from a slave s

cache s in the *slave-sync* set and proceed as if a (y, t) message had been received from child i

end when

Coordinador esclavo

SLAVE COORDINATOR

```

when a (@, t) message is received from parent coordinator
  if t = tN then
    tL := t
    for all imminent child processors i with minimum tN
      send (@, t) to child i
      cache i in the synchronize set
    end for
    wait until (done, t)'s have been received from all imminent processors
    send (done, t) to parent coordinator
  end if
  else raise error
end when
when a (y, t) message is received from child i
  sent_to_master := false
  for all influencees, j of child i
    if j is a local processor
      q := zii(y)
      send (q, t) to child j
      cache j in the synchronize set
    else
      if not sent_to_master
        send (y, t) to parent coordinator
        sent_to_master := true
      end if
    end if
  end for
  if self Ii (y is to be transmitted upward) then
    if not sent_to_master
      send (y, t) to parent coordinator
    end if
  end if
end when
when a (y, i, t) message is received from parent coordinator
  sent_to_master := true
  proceed as if a (y, t) message had been received from child i
end when
when a (q, t) message is received from parent coordinator lock the bag
  Add event q to the bag
  unlock the bag
end when
when a (*, t) message is received from parent coordinator
  if tL δ t δ tN
    for all q bag
      for all receivers of q, j Iself
        if j is a local processor
          q := zself,j(q)
          send (q, t) to j
          cache j in the synchronize set
        else
          do nothing
        end if
      end for
    end for
  end if
  empty bag
  for all i in the synchronize set
    send (*, t) to i
  end for

```

```

wait until all ( done, tN)'s are received
tL := t
tN := minimum of components' tN's
clear the synchronize set
send ( done, tN) to parent coordinator
else raise an error
end when
    
```

4 La aplicación CD++

CD++ es una aplicación basada en objetos que permite simular modelos DEVS. El diseño es un framework de clases en lenguaje C++ que implementan los simuladores abstractos. Estas clases pueden ser extendidas para implementar nuevos formalismos.

Los modelos atómicos son implementados por objetos (clases en código C++) que interactúan con otros objetos para implementar un formalismo. De esta forma distintos modelos atómicos pueden implementar distintos formalismos. Un modelo atómico puede ser un componente específico, o puede ser una implementación de un formalismo. Por ejemplo los modelos atómicos Generator o Queue son modelos específicos que pueden parametrizarse en forma limitada y sirven como bloques de construcción de modelos más complejos. En cambio los modelos basados en el formalismo CELL-DEVS permiten escribir especificaciones mucho más flexibles dentro de un modelo por medio de un lenguaje de especificaciones con mayor poder expresivo.

La arquitectura de CD++ está basada en dos clases abstractas: Model y Processor. Model define modelos conceptuales y Processor implementa los mecanismos de simulación. Estas clases abstractas son extendidas por subclases concretas que implementan los modelos y simuladores específicos. La jerarquía de procesadores está relacionada con la de modelos. Cada clase de modelo tiene una clase de procesador que la simula (Figura 7). Los procesadores corresponden a los simuladores y coordinadores del simulador abstracto [Zei00].

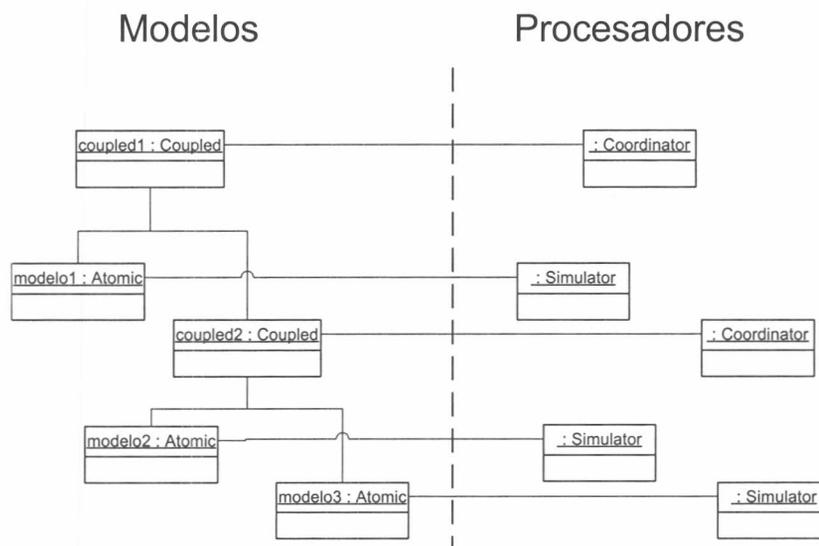


Figura 7 Relaciones entre modelos y procesadores

Model tiene dos subclases directas: Atomic y Coupled. Ambas clases son abstractas. El primero representa los modelos atómicos y el segundo los modelos acoplados. Cuando se desea implementar un nuevo modelo se debe extender una de ellas. Atomic tiene métodos que se deben implementar para crear un modelo atómico. Dichos métodos definen el comportamiento del nuevo modelo.

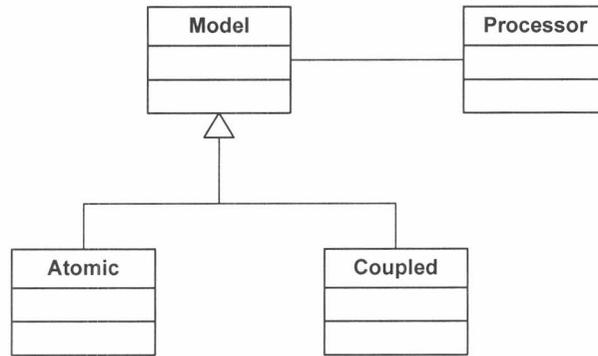


Figura 8 Jerarquía de Modelos de CD++

ξ *initFunction()*: es invocado cuando se inicializa la simulación. Se puede utilizar para definir valores iniciales.

ξ *externalFunction()*: procesa mensajes de eventos externos.

ξ *internalFunction()*: procesa mensajes de eventos internos.

ξ *outputFunction()*: implementa la función de salida.

Hay otros métodos heredados que se pueden sobrecargar para definir el comportamiento del modelo.

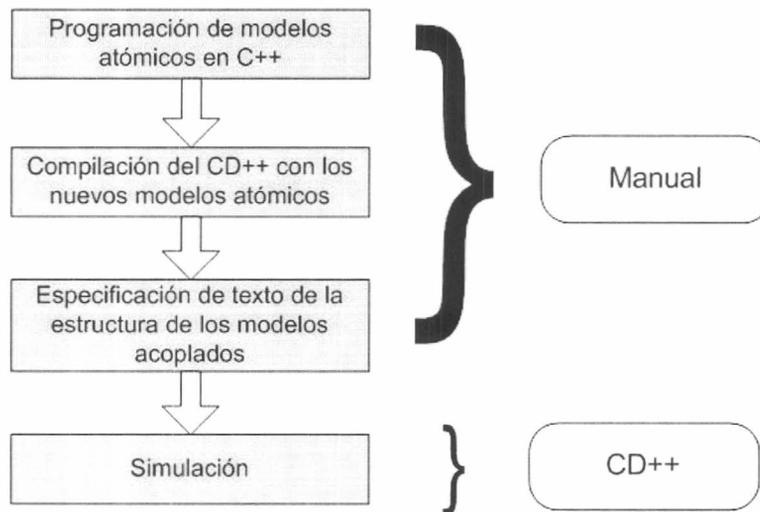
Coupled implementa los modelos acoplados, permitiendo la construcción jerárquica y modular de modelos complejos en base a otros modelos.

El flujo de datos en la simulación se realiza a través de mensajes. Hay una jerarquía de mensajes que implementa cada tipo de mensaje mediante una clase. Los mensajes contienen la información del evento, hora, puerto y valor. CD++ soporta el formalismo DEVS Paralelo que permite mas de un mensaje por puerto como DEVS Clasico. Para ello el metodo **externalFunction()** esta sobrecargado para recibir "bags" de mensajes

5 GGAD, una nueva representación de modelos DEVS

5.1 Motivación

La aplicación CD++ permite crear modelos DEVS acoplados a través de una especificación de modelos acoplados que identifica a los modelos incluidos en el modelo acoplado, y como se interconectan entre si. Los modelos atómicos son normalmente programados en C++ y compilados junto con la aplicación CD++.



Uno de los objetivos de este trabajo es encontrar una forma de representación de modelos atómicos que no requiera el uso de lenguajes de programación y que permita el acceso a la creación de modelos DEVS a un público más amplio.

En ese sentido, analizamos la posibilidad de especificar los modelos en forma textual y gráfica. Cada una de ellas tiene sus ventajas

Especificación textual	Especificación gráfica
ξ Mayor expresividad	ξ Menor expresividad
ξ Menos intuitiva	ξ Más intuitiva
ξ Herramientas disponibles (editores de texto)	ξ Necesidad de generar una herramienta

Una representación textual de modelos atómicos DEVS facilita la especificación de los mismos a un público no experto en programación. Más aún, una aproximación a esta representación textual a través de una especificación gráfica de los modelos facilita una visión más intuitiva del problema, y complementa a la definición de modelos acoplados de CD++ en el sentido que permite llegar a la especificación desde un diseño gráfico de la interconexión de los distintos modelos.

Por lo tanto decidimos utilizar ambas formas de especificación, utilizando el formato gráfico para la generación inicial de los modelos atómicos y su interconexión con otros modelos, y el formato textual para cubrir las brechas de falta de poder expresivo del gráfico.

Si bien la posibilidad de generar modelos atómicos mediante un lenguaje sencillo de especificación de modelos DEVS implica imponer limitaciones de poder expresivo frente a la flexibilidad de un lenguaje de programación como C++, para una parte de los casos el lenguaje de especificación alcanza el objetivo de especificar correctamente el modelo atómico, evitando así la necesidad de recurrir a la programación C++. Para los casos en los que las posibilidades que brinda el lenguaje sencillo de especificación de modelos atómicos no es suficiente, y para cerrar la brecha entre simplicidad y poder expresivo, definimos un mecanismo para extender la expresividad del lenguaje a través de llamadas a funciones desarrolladas en C++.

Entonces lo que se logra es disminuir fuertemente la necesidad de programar en C++, y en los casos en que así se requiera, se programan componentes (funciones) más pequeños y por ende más reusables y sencillos de realizar.

Zeigler, Praehofer y otros muestran en [Zei95] una notación gráfica simple para DEVS. Extendiendo esa representación hemos formalizado GGAD, un lenguaje de representación gráfica de modelos DEVS, tanto atómicos como acoplados, y su correspondiente traducción a un formato textual que describe la estructura y atributos de los modelos atómicos y/o acoplados que puede ser utilizado como input del simulador DEVS.

Mostraremos una especificación de un lenguaje gráfico para modelos DEVS, una aplicación que permite el diseño gráfico de modelos DEVS y puede traducirlos a un lenguaje de especificación textual equivalente, y por último un motor de simulación con la capacidad de interpretar dichos modelos.

El lenguaje de especificación de modelos atómicos DEVS creado se llama GADScript y la herramienta de diseño gráfico de modelos DEVS (atómicos y acoplados) se denomina GGADTool.

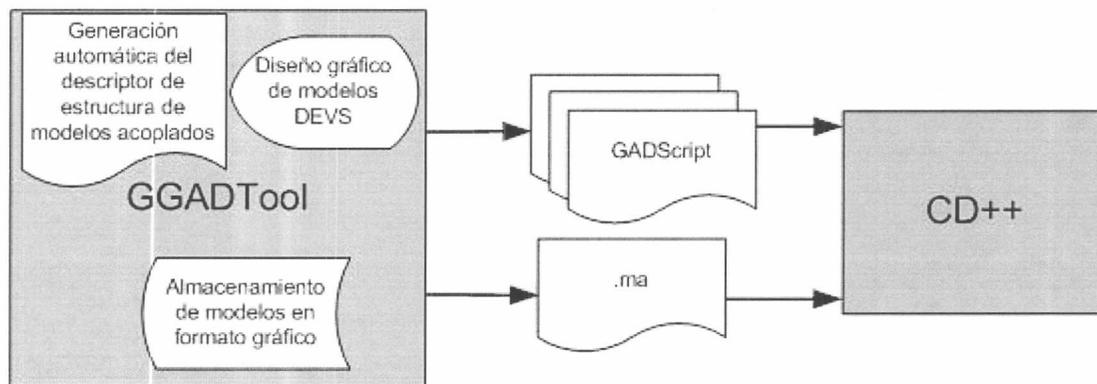
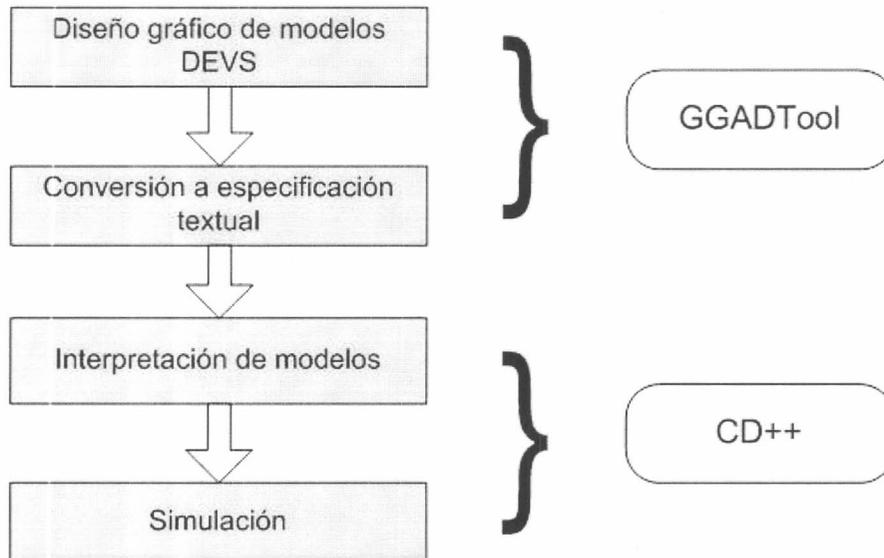


Figura 9 División de roles

En la figura podemos apreciar la división de funcionalidad. La herramienta GGADTool se utiliza para diseñar gráficamente los modelos atómicos y acoplados. Estos pueden ser almacenados para su modificación posterior. En el momento en que se desee ejecutar la simulación sobre el modelo creado, se generaran automáticamente los archivos con formato GADScript que describen la estructura y comportamiento de los distintos modelos atómicos diseñados y el archivo de definición de la estructura del modelo acoplado.

5.2 Modelos GGAD

Ahora describiremos los modelos atómicos GGAD. Los mismos están compuestos por estados, variables, puertos y transiciones.

Estados

Un modelo atómico GGAD en un momento dado de la simulación está en un determinado estado. El estado está dividido en dos partes: un estado nominado y los valores del conjunto de variables del modelo.

Los estados nominados son representados gráficamente por círculos con su nombre en el centro, y por ello son llamados "burbujas". Se utilizan en el mismo sentido que los estados en los autómatas finitos. En general en los modelos GGAD cuando hablemos de estados nos estaremos refiriendo solo a las "burbujas", no incluyendo a las variables. Estas entidades son llamadas fases.

Los estados tienen una propiedad llamada *time advance*, que indica el tiempo durante el cual el modelo permanecerá en este estado en caso de no recibir eventos externos.

Variables

Las variables son globales al modelo y sirven para representar valores, por ejemplo un contador o una cantidad. Las variables pueden ser inicializadas con un valor en la especificación del modelo, comportándose en este caso como parámetros del mismo.

Puertos

La interacción con otros modelos de la simulación y el exterior se realiza a través de puertos de entrada y de salida. Los puertos son identificados por un nombre. Los puertos de entrada reciben mensajes con valores que pueden ser salidas de otros modelos o de la entrada al propio sistema simulado. De la misma forma los puertos de salida pueden estar conectados a las entradas de otros modelos o a salidas del sistema. De cualquier manera al modelo GGAD solo le concierne la recepción y el envío a través de sus puertos y no tiene control alguno del origen o destino de los mensajes, siendo ello responsabilidad del simulador.

Es importante notar que cuando ocurre un evento externo se reciben bolsas (bags) de mensajes, lo que implica que pueden contener más de un mensaje, incluso varios mensajes del mismo puerto. Si el modelo tiene puertos a y b algunos ejemplos de mensajes podrían ser :

```
{(a,1), (b,1)}  
{(a,1)}  
{(b,1)}  
{(a,0),(a,1),(b,1)}
```

Transiciones

Los cambios de estado del modelo se producen al ejecutar transiciones. Cada estado puede estar asociado a una o varias transiciones que definen cuál es el nuevo estado del modelo y cómo cambian los valores de las variables.

Las transiciones pueden ser internas o externas dependiendo de a qué tipo de evento responden. Solo se dispara la ejecución de una transición cuando el modelo recibe un evento. No pueden existir transiciones "espontáneas".

Las transiciones externas están asociadas a la aparición de mensajes en los puertos de entrada del modelo. Están definidas por una expresión y una lista de acciones. De la evaluación de la expresión se determina si la transición es "válida" para ser ejecutada. Solo puede haber una transición válida que atienda un evento, de lo contrario se produce una ambigüedad. Si se da este caso el modelo GGAD abortará la simulación.

Las acciones son asignaciones del resultado de evaluar expresiones a variables. Las acciones solo se ejecutan si la transición es válida.

Las transiciones internas se ejecutan al recibir un mensaje cuando transcurre el plazo del time advance sin que se reciban eventos externos. Las transiciones internas pueden tener asociadas una lista de salidas y una lista de acciones. Las salidas son una lista de (puerto, valor). El modelo emitirá cada valor por el puerto asociado en el momento de ejecutar la transición. Las acciones son exactamente iguales que en el caso de la transición externa.

Las transiciones no se referencian con un nombre sino por su composición, es decir su estado de origen, destino y la condición de validez.

5.3 Lenguaje GADscript

Para especificar los modelos atómicos GGAD hemos definido un lenguaje llamado GADscript. El mismo permite definir modelos con las características mencionadas en el punto anterior y puede ser utilizado como entrada para un simulador.

GADscript se define con la siguiente gramática:

Ggad	→	ModelName GGADT_EOL GgadRules
ModelName	→	GGADT_LBRACKET GGADT_ID GGADT_RBRACKET
GgadRules	→	GgadRule GGADT_EOL GgadRules
GgadRules	→	GgadRule GGADT_EOL
GgadRules	→	GgadRule
GgadRule	→	InDecl
GgadRule	→	OutDecl
GgadRule	→	StateDecl
GgadRule	→	VarDecl
GgadRule	→	StateDef
GgadRule	→	InitialState
GgadRule	→	IntDef
GgadRule	→	ExtDef
GgadRule	→	VarDef
InDecl	→	GGADT_IN GGADT_COLON PortInIdList
OutDecl	→	GGADT_OUT GGADT_COLON PortOutIdList
VarDecl	→	GGADT_VAR GGADT_COLON VarIdList
VarDef	→	GGADT_VARIABLEID GGADT_COLON GGADT_CONSTANT
StateDecl	→	GGADT_STATE GGADT_COLON StateIdList
StateDef	→	GGADT_STATEID GGADT_COLON GGADT_TIME_CONSTANT
StateDef	→	GGADT_STATEID GGADT_COLON GGADT_INFINITE
InitialState	→	GGADT_INITIAL GGADT_COLON GGADT_STATEID
IntDef	→	GGADT_INT GGADT_COLON GGADT_STATEID GGADT_STATEID

	PortValueOutList	Actions
PortValueOutList	→	λ
PortValueOutList	→	GGADT_PORTID GGADT_OUTPUT Expresion PortValueOutList
ExtDef	→	GGADT_EXT GGADT_COLON GGADT_STATEID GGADT_STATEID Expresion GGADT_INPUT GGADT_CONSTANT Actions
Expresion	→	FunctionCall
Expresion	→	GGADT_PORTID
Expresion	→	GGADT_VARIABLEID
Expresion	→	GGADT_CONSTANT
FunctionCall	→	GGADT_FUNCTIONID GGADT_LPAR ActualParamList GGADT_RPAR
ActualParamList	→	Expresion
ActualParamList	→	Expresion GGADT_COMMA ActualParamList
StateIdList	→	StateIdList GGADT_ID
StateIdList	→	GGADT_ID
PortInIdList	→	PortInIdList GGADT_ID
PortInIdList	→	GGADT_ID
PortOutIdList	→	PortOutIdList GGADT_ID
PortOutIdList	→	GGADT_ID
VarIdList	→	VarIdList GGADT_ID
VarIdList	→	GGADT_ID
Actions	→	GGADT_BEGIN ActionList GGADT_END
Actions	→	λ
ActionList	→	Action GGADT_SEMICOLON
ActionList	→	ActionList Action GGADT_SEMICOLON
Action	→	GGADT_VARIABLEID GGADT_ASSIGNMENT Expresion

Cada modelo esta compuesto de un nombre entre corchetes (por ej: "[modelo1]"), seguido de una lista de reglas. Cada regla esta terminada con un caracter de fin de linea.

Descripción de las reglas

Las reglas en GADscript definen los componentes de un modelo GGAD. Comienzan con un identificador del tipo de regla seguido por dos puntos (':') y los atributos de la regla. El identificador puede ser una palabra reservada o referenciar a un objeto ya definido. En el primer caso la regla define objetos y les asigna un tipo. En el segundo caso la regla permite definir atributos adicionales de un objeto.

Regla in

Sintaxis:

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

Declara puertos de entrada del modelo.

Port1 a Portn son nombres de puertos. Los nombres deben corresponderse con puertos definidos en el simulador (archivo .ma) para conectar este modelo con otros

Ejemplos

```
in : entrada
```

```
in : a b c
```

Regla out

Sintaxis:

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

Declara puertos de salida del modelo.

Port1 a Portn son nombres de puertos. Los nombres deben corresponderse con puertos definidos en el simulador (archivo .ma) para conectar este modelo con otros

Regla var

Sintaxis

```
var : variable1 variable2 ... variablen
```

Declara variables pertenecientes al modelo.

variable1 variable2 ... variablen son nombres de variables.

Regla state

Sintaxis

```
state : state1 state2 ... staten
```

Declara estados del modelo. Corresponden a las "burbujas" de la representación gráfica. No confundir los estados con el estado total del modelo, pues este último incluye además las variables del sistema.

Regla de definición de estados

```
statename : time_advance
```

donde statename es el nombre de un estado previamente definido con la regla state y time_advance es el Time advance correspondiente al estado. El time_advance debe estar definido como una constante de tiempo de CD++ de la forma hh:mm:ss:nn o como "infinite".

Esta regla es optativa, sino se define un time advance el estado tendrá por defecto infinite.

Regla de estado inicial

Sintaxis:

```
inital : statename
```

Declara cual es el estado “burbuja” inicial del modelo. Si no se declara un estado ocurrirá un error en tiempo de ejecución de la simulación. El estado inicial debe haber sido declarado previamente con una regla `state`.

Regla de definición de transición interna

Sintaxis

```
int : ESTADO_INICIAL ESTADO_FINAL ( PUERTO_DE_SALIDA ! Expresion )+ { lista de acciones }
```

Define una transición interna desde `ESTADO_INICIAL` a `ESTADO_FINAL`. La lista de pares de puerto de salida y expresiones definen la función de salida para `ESTADO_INICIAL` de forma tal que:

$$O(ESTADO_INICIAL) = \{ (PUERTO_DE_SALIDA ! Expresion)^+ \}$$

Para una descripción de las acciones ver la sección de acciones.

`ESTADO_INICIAL` y `ESTADO_FINAL` deben haber sido previamente declarados como estados con la regla `state`.

Los puertos de salida deben haber sido declarados como tales previamente con la regla `out`.

Regla de definición de transición externa

Sintaxis

```
ext : ESTADO_INICIAL ESTADO_FINAL Expresión ? constante { lista de acciones }
```

Define una transición externa desde `ESTADO_INICIAL` a `ESTADO_FINAL`. En el momento de recibir un evento externo cuando el simulador se encuentra en el estado `ESTADO_INICIAL` se evalúan las expresiones de las transiciones externas que comienzan en ese estado. El operador de entrada (“?”) funciona como una comparación con la restricción que el segundo operando debe ser una constante. El simulador seleccionará la primera transición cuya expresión evalúe al valor requerido. El orden de evaluación es dependiente de la implementación, por lo cual no se deben diseñar modelos que dependan del mismo.

Para una descripción de las acciones ver la sección de acciones.

Expresiones y acciones

Estas no son reglas sino partes de las mismas.

Expresiones

Las expresiones pueden ser:

- ξ constantes numericas
- ξ puertos
- ξ variables
- ξ funciones

Constantes numericas

Ejemplos:

ξ 5
 ξ -10
 ξ 0

Puertos

Los puertos pueden ser de entrada o de salida dependiendo de si estamos especificando transiciones externas o internas respectivamente.

En las transiciones externas estan definidos los puertos que reciben valores. Las transacciones externas que hagan referencia en su expresión a puertos de entrada que no hayan recibido un mensaje serán ignoradas (el puerto tiene un valor indefinido y por lo tanto la expresión no puede ser evaluada).

En el caso de transiciones internas se usan para indicar los puertos por los que se emiten valores. Se puede utilizar cualquier puerto de salida del modelo. No se puede leer el valor de un puerto de salida dentro del modelo que lo emite.

Variables

Las variables pertenecen al modelo.

Su valor puede ser consultado en el momento de la evaluación de la condición de una transición externa (como parte de su expresión) , puede ser utilizado en una expresion de una transición interna para definir el valor que será emitido en la función de salida asociada, y puede ser modificado en una acción durante la ejecución de una transición.

Funciones

Las funciones se implementan en codigo C++ compilado junto con el simulador. Pueden recibir como parametros cualquiera de los otros tipos de expresiones. Esto es, constantes, puertos, variables u otras funciones anidadas.

Actualmente están definidas las siguientes funciones:

Función	Descripción
Add(n1,n2)	devuelve la suma de n1 y n2
And(n1,n2)	retorna verdadero si ambos parámetros son verdaderos
Any(port1)	retorna verdadero si el port port1 recibió un valor valido en este evento
Between(n1,n2,n3)	retorna verdadero si n2 es mayor que n1 y menor que n3
Compare(n1,n2,n3,n4,n5)	retorna n3, n4 o n5 si n1 es mayor, igual o menor que n2 respectivamente
Divide(n1,n2)	retorna n1 dividido n2
Equal(n1,n2)	retorna verdadero si n1 es igual a n2
Greater(n1,n2)	retorna verdadero si n1 > n2
Less(n1,n2)	retorna verdadero si n1 < n2
Minus(n1,n2)	devuelve n1-2
Multiply(n1,n2)	retorna n1 multiplicado por n2
Not(n)	retorna la negación de n
NotEqual(n1,n2)	retorna verdadero si n1 es distinto de n2
Or(n1,n2)	retorna verdadero si uno de los parámetros es verdadero
Pow(n1,n2)	retorna n1 a la potencia n2
Rand(n1,n2)	retorna un valor aleatorio entre n1 y n2
Value(n)	retorna el valor de n

Acciones

Las acciones son la única forma de cambiar el estado total del modelo, además de cambiar el estado "burbuja" mediante una transición. Las acciones permiten cambiar el valor de las variables asignándoles el resultado de una expresión.

5.4 Representación gráfica

La representación gráfica de modelos es una simplificación de la representación textual que permite una forma sencilla de diseñar modelos DEVS. No encontramos práctico o útil representar gráficamente todos los objetos (por ejemplo expresiones). En cambio se representan los que tienen un significado estructural para el modelo a simular como es el caso de los estados y su interconexión a través de transiciones externas e internas.

Estados

Los estados no representan el estado total del modelo sino que son un componente más del mismo, junto con las variables.

Se representan gráficamente con un círculo y el nombre del estado adentro

Ejemplo:



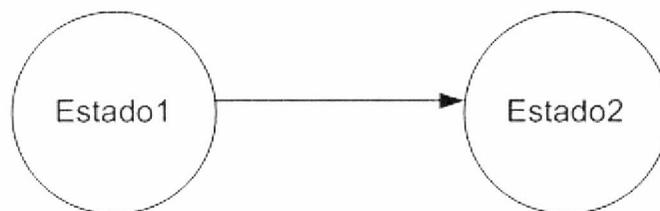
Transiciones

Hay dos tipos de transiciones: externas e internas.

Transiciones externas

Se representan gráficamente con una línea continua con una flecha en la dirección al destino de la transición.

Ejemplo:



Transiciones internas

Se representan gráficamente con una línea discontinua con una flecha en la dirección al destino de la transición.

Ejemplo:

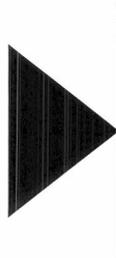


Puertos

Los puertos estan asociados al modelo. Representan las entradas y salidas del mismo.

Puertos de entrada

Se representan graficamente con un triángulo apuntando hacia el interior del modelo.
Ejemplo:



Puertos de salida

Se representan graficamente con un triángulo apuntando hacia fuera del modelo.
Ejemplo:



5.5 Relación entre DEVS y GGAD

Los modelos atómicos GGAD representan un subconjunto de los modelos atómicos DEVS.

En terminos de DEVS un modelo GGAD se representa como:

$$GGAD = \langle X_M, Y_M, S, \Gamma_{ext}, \Gamma_{int}, \Gamma_{com}, Q_{ta} \rangle$$

donde:

$X_M = \{(p,v) \mid p \in IPorts, v \in X_p\}$ es el conjunto de pares de puerto de entrada y el valor recibido;

$Y_M = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ es el conjunto de pares de puerto de salida y el valor emitido;

$S = B \times P(V)$ es el conjunto de estados secuenciales,

donde:

$B = \{ b \mid b \text{ Burbujas} \}$ es el conjunto de estados del modelo.

$V = \{ (v,n) \mid v \text{ Variables, } n \in R_0 \}$ es el conjunto de variables del modelo y sus valores.

$P(V)$ es el conjunto partes de V .

$\Gamma_{ext}: Q \times X_M^b \rightarrow S$ es la función de transición externa;

$\Gamma_{int}: S \rightarrow S$ es la función de transición interna;

$\Gamma_{con}: Q \times X_M^b \rightarrow S$ es la función de confluencia;

$O: S \rightarrow Y_M^b$ es la función de salida;

$ta: S \rightarrow R_0^+ \cup \phi$ es la función de time advance;

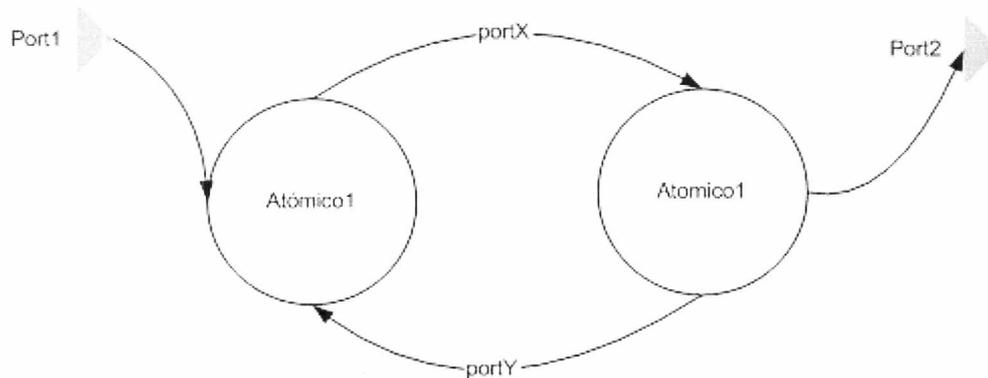
donde $Q := \{ (s, e) \mid s \in S, \theta \in \delta \text{ e } \delta ta(s) \}$ es el conjunto de *estados totales*.

La función de confluencia se define como el resultado de aplicar primero la función de transición interna y luego la externa. Este es el comportamiento por defecto en CD++.

5.6 Traducción de representación gráfica a GADscript

Como ejemplo de la relación entre el formato gráfico y GADscript que es el input para CD++, se muestra un modelo acoplado sencillo compuesto por dos modelos atómicos.

El modelo acoplado es el siguiente:



y su representación textual, según el archivo de inicialización de CD++ es:

```
[Top]
components : atómico1@GGad atómico2@GGad
out : Port2
in : Port1
Link : Port1 Port1@atómico1
Link : Port2@atómico2 Port2
Link : PortX@atómico1 PortX@atómico2
Link : PortY@atómico2 PortY@atómico1

[atómico1]
source : atómico1.cdd

[atómico2]
source : atómico2.cdd
```

El formato de este archivo .ma de CD++ es el siguiente:

En la sección [Top] se describe el modelo acoplado raíz del sistema.

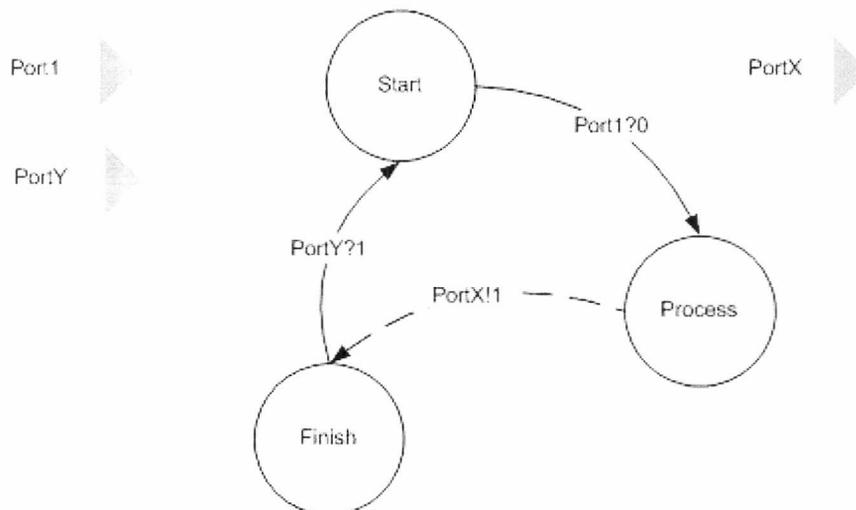
- ξ La primera línea “components” contiene una lista de los modelos que componen a este acoplado con la notación Nombre_de_modelo@Clase_de_modelo (en el caso de modelos atómicos GGAD, se los identifica por tener la clase de modelo GGAD)
- ξ La segunda y tercera línea “out” e “in” listan los nombres de los puertos de entrada y salida del modelo acoplado que se está definiendo
- ξ Luego aparecen varias sentencias “link” que describen las conexiones entre puertos de los distintos modelos que lo componen

Por cada modelo listado en “components” aparece una sección para definirlo. En el caso que el modelo sea atómico, dentro de esta sección aparecerán los parámetros que el mismo tenga definidos con los valores a ser instanciados durante la simulación.

Si el modelo era acoplado, en su sección repetirá el formato de la sección [Top] para describir su estructura interna.

Nótese que con la utilización de GGADTool, este archivo .ma es generado automáticamente por la herramienta según el diseño gráfico del modelo acoplado.

El modelo atómico1 es el que sigue



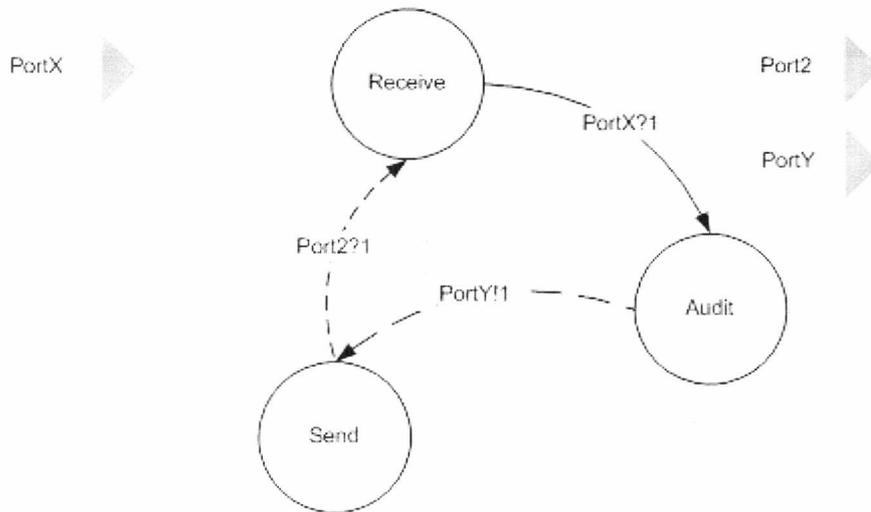
y su representación textual GADscript es::

Atomico1.cdd

```

[atomoic1]
in: Port1 PortY
out: PortX
state: Start Process Finish
int: Process Finish PortX!1
ext: Start Process Value(Port1)?0
ext: Finish Start Value(PortY)?1
Start: 0:0:10:0
Process: 0:0:10:0
Finish: 0:0:10:0
    
```

Por último, tenemos el modelo atómico2 y su representación usando GADscript:



Atomico2.cdd

```

[atomico2]
in: PortX
out: PortY Port2
state: Receive Audit Send
int: Audit Send PortY!1
int: Send Receive Port2!1
ext: Receive Audit Value(PortX)?1
Receive: 0:0:10:0
Audit: 0:0:10:0
Send: 0:0:10:0
    
```

5.7 Compatibilidad e interacción con modelos atómicos DEVS clásicos

El modelo GGAD es una representación de modelos DEVS. No requiere un simulador específico sino que utiliza el mismo simulador que un modelo DEVS Clásico.

El nuevo formato de modelo atómico fue agregado a CD++ de manera que para el simulador no hubiera diferencia con un modelo atómico clásico (programado en C++ y compilado junto con CD++). Esto es posible porque la interface de los modelos atómicos GGAD es idéntica a la de los modelos clásicos.

Por lo tanto, un modelo acoplado DEVS puede estar compuesto por la cantidad de submodelos (atómicos o acoplados) que se deseen y en el caso de los atómicos pueden ser clásicos o GGAD.

Tomando el ejemplo teórico del punto anterior, nótese que en el listado de modelos que componen al modelo acoplado los modelos atómicos GGAD son identificados por la clase de modelos "ggad". En la sección donde se define el modelo ggad, el atributo "source" indica la locación del archivo en GADscript que define a ese modelo atómico.

```

[Top]
components : atomico1@GGad atomico2@GGad
out : Port2
in : Port1
Link : Port1 Port1@atomico1
Link : Port2@atomico2 Port2
Link : PortX@atomico1 PortX@atomico2
    
```

```
Link : PortY@atomico2 PortY@atomico1

[atomico1]
source : atomico1.cdd

[atomico2]
source : atomico2.cdd
```

En el caso de que el modelo atómico2 no fuera GGAD, sino que fuera una instancia de un modelo atómico clásico llamado "myModel" el cual a su vez tuviera la posibilidad de configurar el valor de 3 atributos llamados Property1, Property2 y Property3, el contenido del archivo sería el siguiente:

```
[Top]
components : atomico1@GGad atomico2@myModel
out : Port2
in : Port1
Link : Port1 Port1@atomico1
Link : Port2@atomico2 Port2
Link : PortX@atomico1 PortX@atomico2
Link : PortY@atomico2 PortY@atomico1

[atomico1]
source : atomico1.cdd

[atomico2]
Property1 : 2
Property2 : 11
Property3 : 43
```

En definitiva, los modelos atómicos GGAD son una extensión a las opciones que ofrece el simulador DEVS CD++, y como tal pueden ser combinados con otros tipos de modelos. De esta forma se pueden crear modelos acoplados con componentes basados en los distintos formalismos que permite utilizar CD++, como por ejemplo el formalismo CELLDEVS [BAR98].

6 Implementación de modelos atómicos GGAD en CD++

6.1 Consideraciones de diseño

Para implementar los modelos GGAD en CD++ se incorporó un nuevo tipo de modelo atómico. No fue necesario modificar el funcionamiento del simulador, sino que se extendió su comportamiento utilizando los mecanismos previstos para incorporar nuevos modelos. Solo se usaron las interfaces publicadas para dicho fin. El modelo atómico GGAD está desacoplado del funcionamiento del resto de CD++ para evitar interdependencias innecesarias con otros componentes. De esta forma se evitó que cambios en la estructura de CD++ impacten significativamente en el desarrollo. Físicamente la implementación del modelo atómico GGAD está provista por una biblioteca.

Por convención todos los objetos tienen el prefijo `ggad` (o una variante, por ej: GGAD). Se han utilizado solamente técnicas de programación válidas de ANSI C++ para facilitar una migración a otro compilador o plataforma en el futuro. Internamente el subsistema para los modelos Ggad tiene un diseño orientado a objetos, basado en clases interdependientes.

A continuación describiremos las clases más importantes y cómo se relacionan.

La clase `Ggad` implementa el modelo atómico y funciona como punto de acceso del simulador. Junto con `GgadDynamicState` (que mantiene el estado del modelo) son las únicas clases visibles por CD++. `Ggad` recibe los mensajes de CD++ y delega en `GgadImpl` la implementación del modelo. Al estar separada de la atención de eventos y la interacción con CD++ que es responsabilidad de `Ggad`, es posible cambiar la implementación del modelo sin afectar el resto del módulo, por ejemplo si se desea una implementación

mas eficiente en espacio o en tiempo. Ggad utiliza a GgadParser para construir un modelo (GgadImpl) a partir de una especificación. Ademas los identificadores son incorporados a una tabla de simbolos implementada por GgadSymbolTable.

Otra clase importante es GgadSyntaxNode que es una clase abstracta utilizada para armar un arbol sintactico de las expresiones y acciones.

Clase Ggad

Responsabilidades:

- ξ Atención de eventos del simulador
- ξ Logging de cambios de estado

Clase GgadImpl

Responsabilidades:

- ξ Implementar el modelo.

Clase GgadParser

Responsabilidades:

- ξ Construir un modelo a partir de una especificación textual.

Clase GgadSymbolTable

Responsabilidades:

- ξ Implementar una tabla de simbolos para registrar los objetos.

Clase GgadDynamicState

Responsabilidades:

- ξ Almacenar el estado dinamico del modelo.

Clase GgadDynamicStateAdapter

Responsabilidades

- ξ Esta clase implementa el pattern Adapter. Permite acceder al estado del modelo con una interfaz sin acceder directamente a la clase GgadDynamicState.

Clase GgadSyntaxNode

Clase abstracta. Define un árbol sintactico utilizando el pattern Parser. Cada subclase representa un nodo concreto del árbol. Los nodos son evaluados mediante el metodo execute(). Este metodo recibe como parámetro un objeto GgadDynamicStateAdapter que permite acceder al estado actual del modelo.

Responsabilidades:

- ξ Definir la interfaz un árbol sintactico.

Clase GgadActionCode

Clase abstracta. Subclase de GgadSyntaxNode. Esta clase es la superclase de los nodos de acciones de un árbol sintactico. Los nodos de acción pueden tener efectos laterales, como cambiar el valor de una variable.

6.2 Diagrama de clases

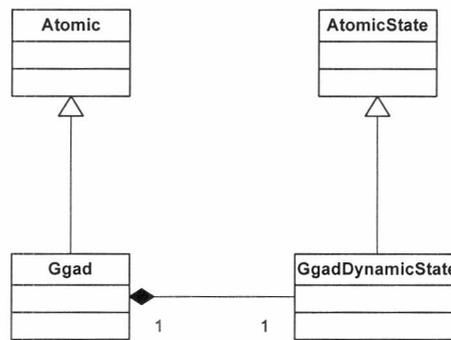


Figura 10 Clases Ggad y GgadDynamicState

6.3 Algoritmos para interpretar modelos GGAD

Función de transición externa

parametro: msgList

Buscar una transición externa cuya condición sea verdadera dados el estado del modelo y los mensajes.

Si encuentra una
 cambio el estado actual al estado destino.
 Aplicar las reglas asociadas.
 holdIn(timeadvance del nuevo estado).

Si encuentra mas de una
 error de ejecución.

Si no encuentro ninguna
 error de ejecución.

Fin

Función de transición interna

parametro: msg

Buscar la transición.

Si encuentra una
 cambio el estado actual al estado destino.
 Aplicar las reglas asociadas.
 holdIn(timeadvance del nuevo estado).

Si no encuentro ninguna
 error de ejecución.

Fin

Función de salida

Buscar las salidas asociada al estado.

Si existen salidas
 emitirlas.

Fin

6.4 Log de eventos

Para permitir un fácil seguimiento al ejecutar un modelo GGAD se produce un log que detalla los eventos de inicialización, entrada, salida y cambios de estado.

El formato general es el siguiente:

- ξ 1er columna: Tipo de evento.
- ξ 2da columna: Tiempo de la simulación en que ocurrió el evento

Luego hay un detalle que depende de cada evento, separado por el símbolo '!':

Evento Inicialización del modelo.

Este evento se registra cuando el modelo es inicializado.

Log: estado inicial, { (variable, valor inicial), (variable, valor inicial), ... }

Evento Mensaje de entrada

Este evento se produce una vez por cada mensaje que se recibe en un bag.

Log: puerto de entrada, valor recibido.

Evento Mensaje de salida

Este evento se produce una vez por cada mensaje que se envía al aplicar la función de salida.

Log: puerto de salida, valor enviado.

Evento Función de Transición Interna.

Este evento se produce al ejecutar la función de transición interna. Se registran el estado inicial de la transición, su estado final y el estado de las variables del sistema al finalizar la función de transición.

Log: burbuja inicial, burbuja final, { (variable, valor inicial), (variable, valor inicial), ... }

Evento Función de Transición Externa.

Este evento se produce al ejecutar la función de transición externa. Se registran el estado inicial de la transición, su estado final y el estado de las variables del sistema al finalizar la función de transición.

Log: burbuja inicial, burbuja final, { (variable, valor inicial), (variable, valor inicial), ... }

6.5 Funciones nativas

Se pueden incorporar funciones a GADscript implementándolas en lenguaje c++, utilizando un patrón conocido como objeto función. Las funciones son subclases de la clase GgadFunc. Esta clase es abstracta y provee el mecanismo genérico para definir parámetros y realizar la ejecución. Las funciones no deben provocar efectos laterales o modificaciones del estado de la simulación. Cada invocación a una función puede implicar la creación de una nueva instancia de un objeto función del tipo solicitado.

Para implementar una nueva función se deben implementar el método executeImpl() y el constructor de la clase. En el constructor se deben definir la cantidad de parámetros que debe recibir la función utilizando el método setFormalParams(). El modelo GGAD agregará los parámetros utilizando el método addParameter() tantas veces como sea necesario.

Si el número de parámetros no coincide con el especificado con setFormalParams() se producirá un error en tiempo de ejecución. El comportamiento de la función se especifica en el método executeImpl(). Este método puede consultar los parámetros recibidos con el método getParameter(), realizar las operaciones requeridas y retornar el valor resultante, que se recibirá en la expresión que está siendo evaluada.

Ejemplo de implementación de una función para sumar dos números:

```
#include "ggadfunc.h"
class GgadFuncAdd : public GgadFunc {
public:
    GgadFuncAdd() {
        setFormalParams(2);
    }
    virtual ~GgadFuncAdd() {}
    virtual std::string name() { return "add"; }
    virtual GgadValue executeImpl () {
        return (getParameter(0) + getParameter(1));
    }
    virtual GgadFuncAdd* create() { return new GgadFuncAdd(); }
};
```

7 Ejemplos

En esta sección procederemos a demostrar el uso de las herramientas expuestas para modelar distintos casos de la realidad. Nuestro objetivo es comprobar el poder expresivo de la notación definida (GADscript) y su correcta interpretación por parte de CD++ durante la ejecución. Mostraremos para cada caso una descripción introductoria de la situación a modelar, diagramas de los distintos modelos, su traducción a GADscript. Por último describiremos los resultados de la ejecución de la simulación.

7.1 Ascensor simple

7.1.1 Descripción

Este ejemplo modela el funcionamiento de un sistema compuesto por un ascensor y su controlador. El ascensor es comandado por los botones de llamado en cada piso o su equivalente dentro del ascensor. Los botones de llamado en los pisos no discriminan el sentido del llamado (para ir arriba o abajo). El controlador solo recibe un nuevo llamado cuando no está procesando uno anterior.

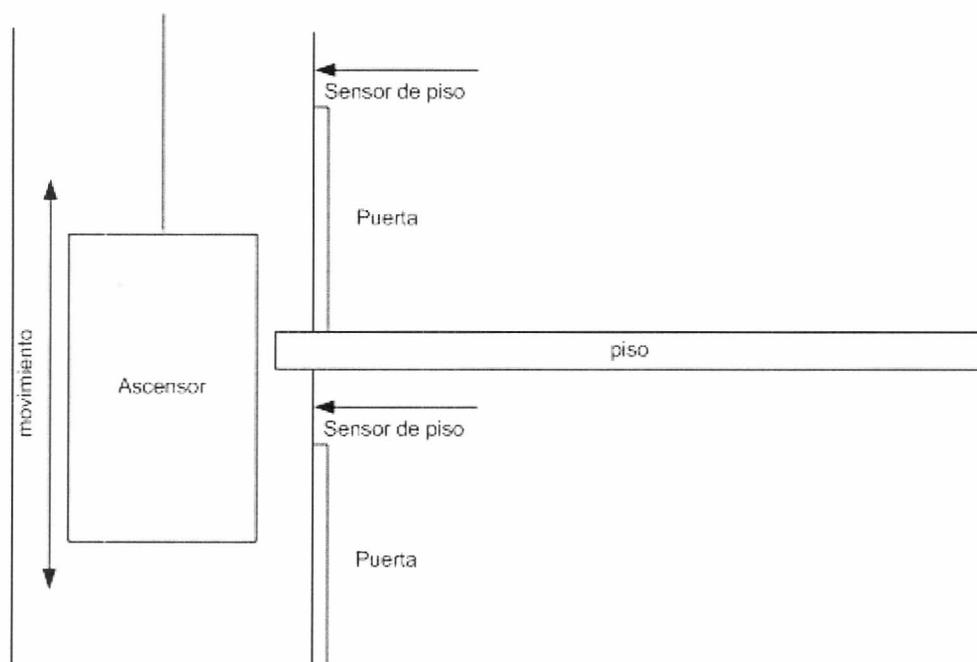
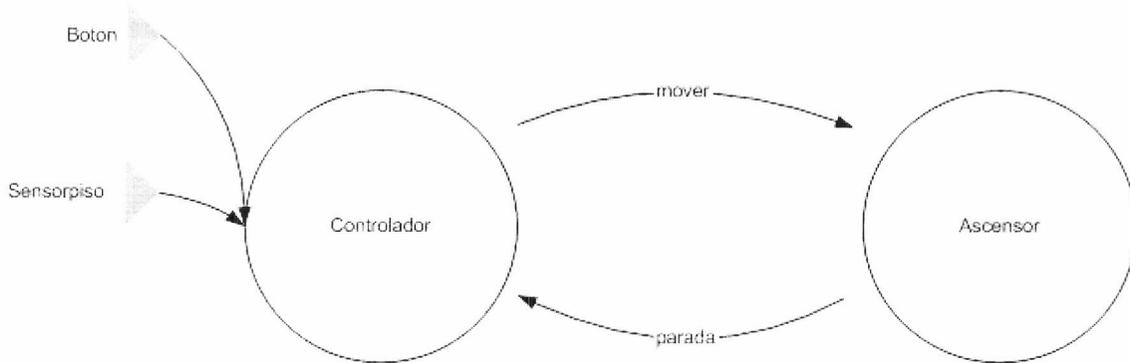


Figura 11 Diagrama conceptual del ascensor.

7.1.2 Modelos

El modelo acoplado puede verse en la siguiente figura:



Puertos de entrada

Puerto	Descripción
Boton	representa tanto a los botones del ascensor como a los botones en cada piso para llamarlo.
SensorPiso	es un sensor externo al sistema que avisa cuando el ascensor está por pasar por un piso.

Modelos atómicos

Modelo	Descripción
Controlador	dirige el movimiento del ascensor según los llamados recibidos.
Ascensor	Sube, baja y se detiene según las ordenes recibidas del controlador a través del puerto "mover". Cuando el ascensor se detiene, se lo comunica al controlador mediante el puerto "parada".

AscensorSimple.ma

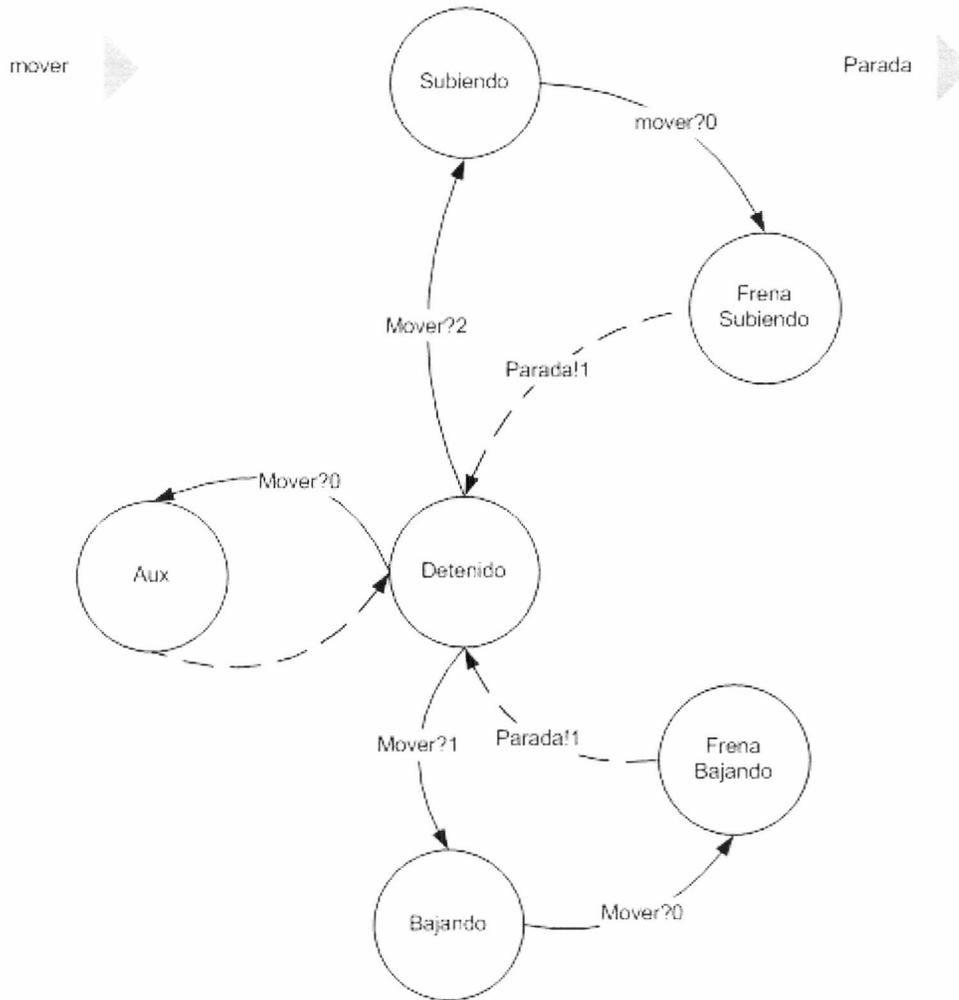
```

[Top]
components : controlado@GGad ascensor@GGad
in : boton sensorpiso
Link : mover@controlado mover@ascensor
Link : paro@ascensor parada@controlado
Link : boton boton@controlado
Link : sensorpiso sensor_piso@controlado

[controlado]
source : controlado.cdd

[ascensor]
source : ascensor.cdd
    
```

Aquí vemos el modelo atómico del Ascensor:



El funcionamiento es como sigue:

El ascensor solo acepta ordenes de ponerse en marcha cuando está detenido. Si recibe un 2 en el puerto “mover”, significa que suba y si recibe un 1 es que baje. Estando en movimiento espera recibir un 0 en el puerto “mover” para iniciar el frenado. Una vez que logra detenerse emite un 1 por el puerto “Parada” para advertir al controlador

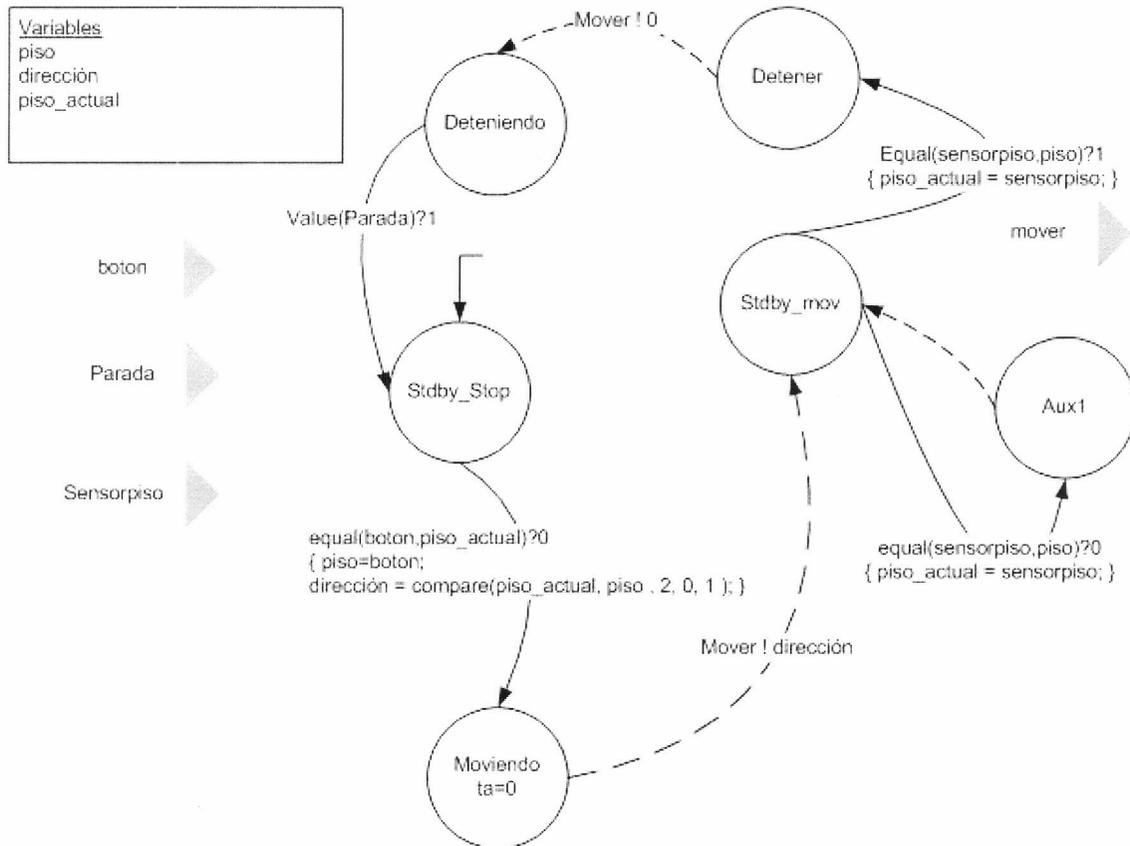
Ascensor.cdd

```

[ascensor]
in: mover
out: paro
state: detenido subiendo bajando frenabajan frenasubie aux
initial:detenido
int: frenasubie detenido paro!1
int: frenabajan detenido paro!1
int: aux detenido
ext: detenido subiendo Value(mover) ?2
ext: subiendo frenasubie Value(mover) ?0
ext: detenido bajando Value(mover) ?1
ext: bajando frenabajan Value(mover) ?0
ext: detenido aux Value(mover) ?0
detenido: 0:0:1000:0
subiendo: 0:0:1000:0
bajando: 0:0:1000:0
frenabajan: 0:0:1:0
    
```

```
frenasubie: 0:0:1:0
aux: 0:0:0:0
```

Y por último el modelo atómico del controlador:



El controlador funciona de la siguiente manera:

Tiene 3 puertos de entrada:

- ξ Boton: aquí recibe los llamados de los diferentes pisos o desde el tablero dentro del ascensor. El número recibido indica el piso al que hay que ir.
- ξ Parada: por este port llega un valor “1” cuando el ascensor se detiene
- ξ SensorPiso: recibe el número de piso al que esta por llegar el ascensor

Y un puerto de salida:

- ξ Mover: le pasa los comandos de movimiento al ascensor

También posee 3 variables:

- ξ Piso: almacena el piso al que está yendo
- ξ Dirección: es para calcular si tiene que ir arriba o abajo cuando es llamado
- ξ Piso_actual: último piso por el que pasó el ascensor

Su lógica de funcionamiento es la siguiente:

Inicialmente está en el estado StdByStop que significa que está esperando llamados y el ascensor está parado. Si le llega un pedido de un piso distinto al actual, almacena el piso destino y le pasa al ascensor el comando de movimiento correspondiente, terminando en el estado StdByMov.

En este estado va a esperar ciclando hasta que los sensores de piso le avisen que llegó al piso deseado, en el que va a mandar el comando para detener el ascensor a través del puerto “mover” en 0. Luego espera recibir la notificación de que el ascensor está parado antes de ponerse nuevamente en espera de un nuevo llamado en StdByStop.

Controlado.cdd

```
[controlado]
in: boton parada sensor_piso
out: mover
var: piso pisoactual direccion
state: Moviendo aux1 StdbyStop StdbyMov Deteniendo
int: Moviendo StdbyMov mover!direccion
int: Deteniendo StdbyStop mover!0
int: aux1 StdbyMov
ext: StdbyStop Moviendo Equal(boton,pisoactual)?1 {piso =
boton;direccion = compare(pisoactual,piso,2,0,1);}
ext: StdbyMov Deteniendo Equal(sensor_piso,piso)?1 {pisoactual =
sensor_piso;}
ext: StdbyMov aux1 Equal(sensor_piso,piso)?0
Moviendo: 0:0:0:0
aux1: 0:0:0:0
StdbyStop: 0:0:1000:0
StdbyMov: 0:0:1000:0
Deteniendo: 0:0:0:0
piso:0
pisoactual:0
direccion:0
```

7.1.3 Simulación

Para la ejecución de la simulación se utilizaron las siguientes entradas:

Tiempo	Puerto	Valor
00:00:5:00	boton	3
00:00:10:00	sensorpiso	1
00:00:14:00	sensorpiso	2
00:00:18:00	sensorpiso	3
00:00:27:00	boton	1
00:00:32:00	sensorpiso	2
00:00:36:00	sensorpiso	1

Para ejecutar la simulación de este modelo se ejecutó el comando:

```
cd++ -mascensorSimple.ma -eascensorSimple.ev -lascensorSimple.log -
oascensorSimple.out
```

Luego de finalizada la simulación, en los archivos "NombreDeModelo.translog" queda el registro de las transiciones efectuadas y las salidas que las mismas ocasionaron.

Controlado.translog

```
C 00:00:00:000 : stdbystop , (direccion=0) (piso=0) (pisoactual=0)
? 00:00:05:000 : boton , 3
E 00:00:05:000 : stdbystop , moviendo (direccion=2) (piso=3) (pisoactual=0)
O 00:00:05:000 : mover , 2
I 00:00:05:000 : moviendo , stdbymov (direccion=2) (piso=3) (pisoactual=0)
? 00:00:10:000 : sensorpiso , 1
E 00:00:10:000 : stdbymov , aux1 (direccion=2) (piso=3) (pisoactual=1)
I 00:00:10:000 : aux1 , stdbymov (direccion=2) (piso=3) (pisoactual=1)
? 00:00:14:000 : sensorpiso , 2
```

```

E 00:00:14:000 : stdbymov , aux1 (direccion=2) (piso=3) (pisoactual=2)
I 00:00:14:000 : aux1 , stdbymov (direccion=2) (piso=3) (pisoactual=2)
? 00:00:18:000 : sensorpiso , 3
E 00:00:18:000 : stdbymov , detener (direccion=2) (piso=3) (pisoactual=3)
O 00:00:18:000 : mover , 0
I 00:00:18:000 : detener , deteniendo (direccion=2) (piso=3) (pisoactual=3)
? 00:00:19:000 : parada , 1
E 00:00:19:000 : deteniendo , stdbystop (direccion=2) (piso=3) (pisoactual=3)
? 00:00:27:000 : boton , 1
E 00:00:27:000 : stdbystop , moviendo (direccion=1) (piso=1) (pisoactual=3)
O 00:00:27:000 : mover , 1
I 00:00:27:000 : moviendo , stdbymov (direccion=1) (piso=1) (pisoactual=3)
? 00:00:32:000 : sensorpiso , 2
E 00:00:32:000 : stdbymov , aux1 (direccion=1) (piso=1) (pisoactual=2)
I 00:00:32:000 : aux1 , stdbymov (direccion=1) (piso=1) (pisoactual=2)
? 00:00:36:000 : sensorpiso , 1
E 00:00:36:000 : stdbymov , detener (direccion=1) (piso=1) (pisoactual=1)
O 00:00:36:000 : mover , 0
I 00:00:36:000 : detener , deteniendo (direccion=1) (piso=1) (pisoactual=1)
? 00:00:37:000 : parada , 1
E 00:00:37:000 : deteniendo , stdbystop (direccion=1) (piso=1) (pisoactual=1)
    
```

Ascensor.translog

```

C 00:00:00:000 : detenido
? 00:00:05:000 : mover , 2
E 00:00:05:000 : detenido , subiendo
? 00:00:18:000 : mover , 0
E 00:00:18:000 : subiendo , frenasubie
O 00:00:19:000 : paro , 1
I 00:00:19:000 : frenasubie , detenido
? 00:00:27:000 : mover , 1
E 00:00:27:000 : detenido , bajando
? 00:00:36:000 : mover , 0
E 00:00:36:000 : bajando , frenabajan
O 00:00:37:000 : paro , 1
I 00:00:37:000 : frenabajan , detenido
    
```

7.2 Alternating Bit Protocol

Presentamos una implementación del conocido protocolo Alternating Bit Protocol con GGAD. Este protocolo ha sido ampliamente estudiado. La versión que utilizamos esta basada en [Hol91]. Existen dos entidades que se comunican, una envía mensajes y la otra es receptora, a través de un canal por el que se envían dos mensajes (M0 y M1) que representan una secuencia alternada de datos (M0M1M0...).

El emisor envía el primer mensaje y espera una confirmación del receptor de ese mensaje. Si lo recibe envía el siguiente mensaje y espera una confirmación del segundo. Si no se recibe confirmación, o se recibe confirmación del mensaje erroneo el emisor reenvía el mensaje. El nombre del protocolo proviene del hecho que ambos mantienen un estado representando el mensaje M0 o M1 y alternan entre ambos.

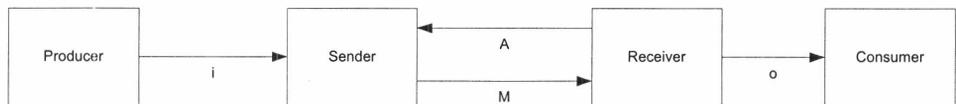


Figura 12 El emisor envía mensajes al receptor mientras que el receptor confirma los mensaje recibidos. Notese que hemos separado el productor y el consumidor del mensaje del protocolo.

Se suele representar el protocolo usando automatas finitos con extensiones para representar eventos temporales, como CFSM (communicating finite state automaton). Mostraremos una implementación de un protocolo en GGAD.

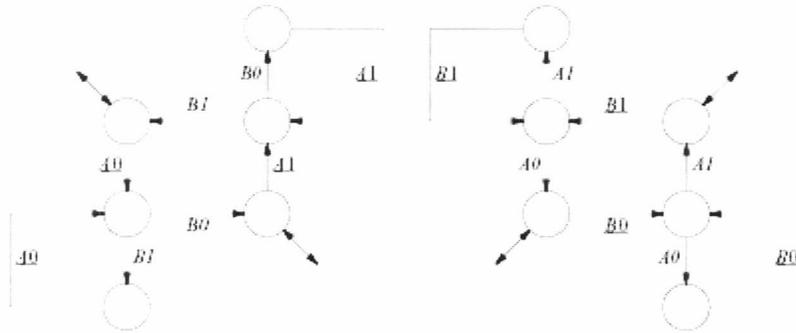


Figura 13 Protocolo de Bit Alternado original.

Para nuestra implementación utilizamos dos canales de datos, uno para enviar cada tipo de mensaje. Sin embargo solo es necesario un canal de confirmación que envíe el número de mensaje recibido (0 o 1). En la Figura 14 podemos ver el modelo acoplado. Para simular fallas incorporamos un canal con posibilidad de fallas en uno de los enlaces. El modelo channel puede reenviar el mensaje que recibe o puede ignorarlo, aleatoriamente, dependiendo de una tasa de error configurable.

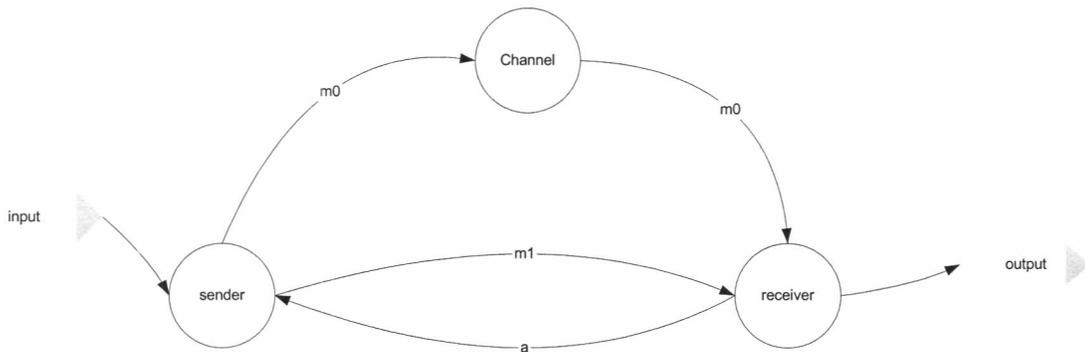


Figura 14 Modelo acoplado para Protocolo de Bit Alternado

En caso que un mensaje no llegue el emisor lo reenviara hasta recibir la confirmación de recepción.

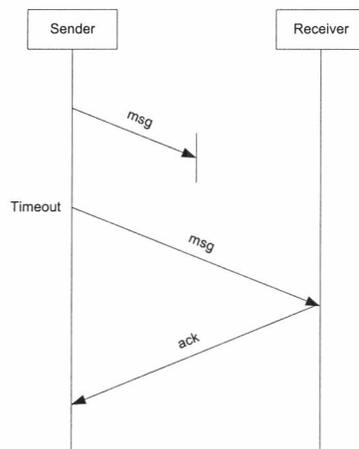


Figura 15 Secuencia de error de recepción.

7.2.1 Modelo Sender

El modelo es similar al original, con algunos estados y transiciones que fueron agregados para particionar la entrada en mensajes M0 y M1, lo que en el original no es tenido en cuenta. Es importante notar que no

se esta modelando un protocolo de sincronización entre la fuente de mensajes y Sender, por lo que podría ocurrir un overflow si se envian mas de 2 mensajes (la capacidad de memoria del protocolo) sin haber recibido confirmación.

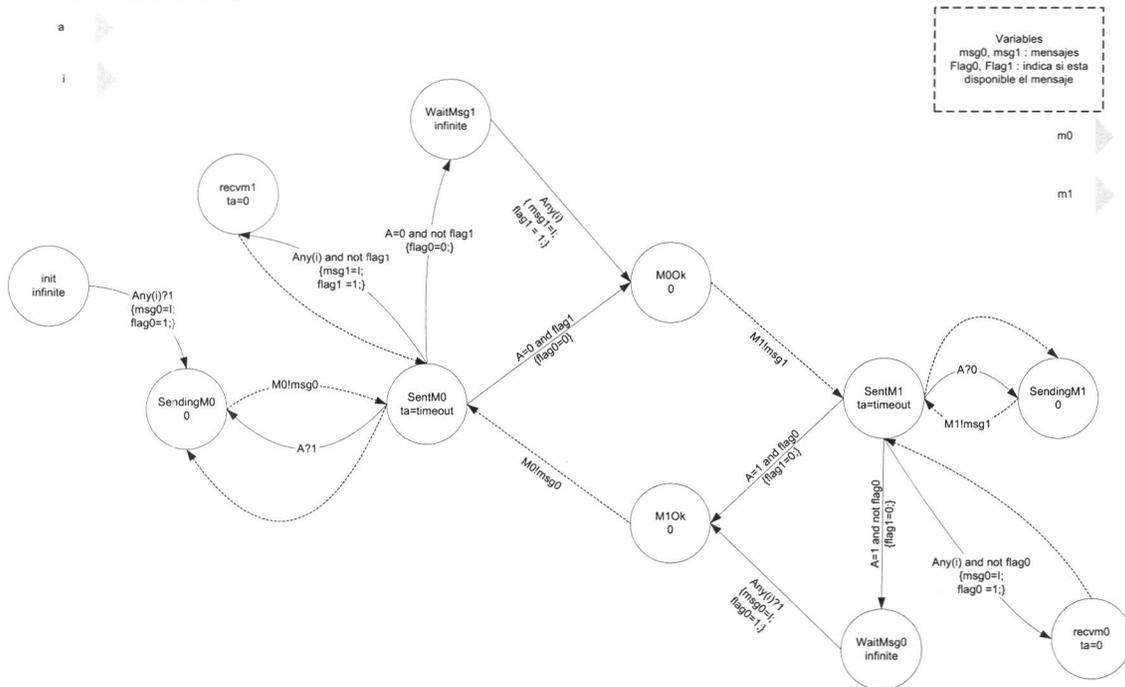


Figura 16 Modelo átomico Sender

```
[sender]
in: i a
out: m0 m1
var: msg0 msg1 flag0 flag1
state: start sentm0 m0ok mlok sentm1 rcvm0 rcvm1 waitmsg1
sendingm0 WaitMsg0 sendingm1
initial:start
int: sendingm0 sentm0 m0!msg0
int: rcvm1 sentm0
...
ext: sentm0 rcvm1 and( any(i), not( flag1 ) ) ? 1 {msg0 = i;flag1
= 1;}
ext: sentm0 m0ok and(any(a), equal(flag1,1) ) ? 1 {flag0 = 0;}
ext: sentm0 waitmsg1 and(any(a), notequal(flag1,1) ) ? 1 {flag0 =
0;}
ext: start sendingm0 any(i)?1 {msg0 = i;flag0 = 1;}
ext: sentm0 sendingm0 Value(a)?1
...
```

7.2.2 Modelo Receiver

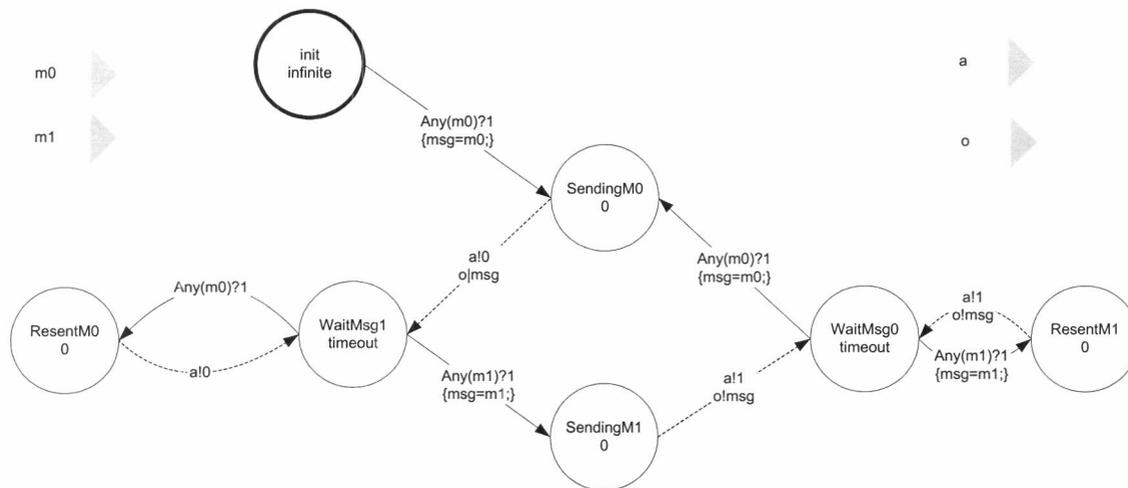


Figura 17 Modelo átomico receiver.

Modelo Channel

Este modelo representa un canal de comunicación con una tasa de error. Si se produce un error el mensaje no llega a destino. En caso contrario el mensaje llega sin demoras.

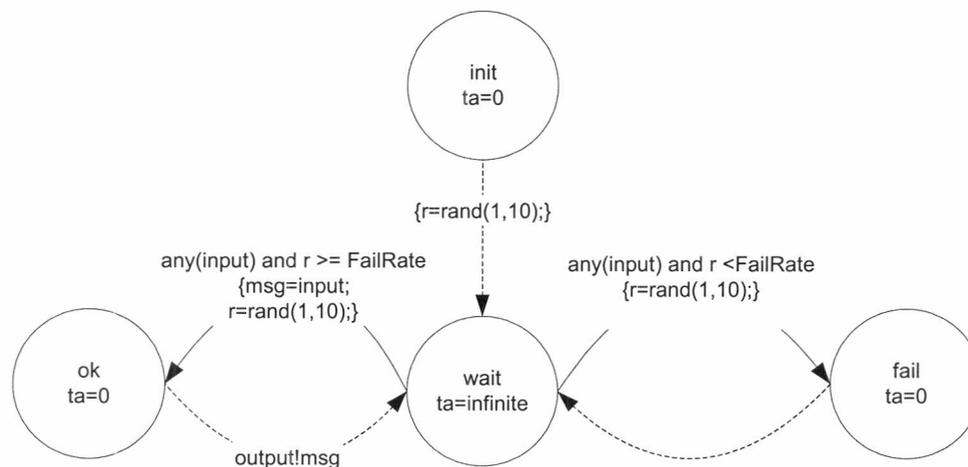


Figura 18 Modelo átomico Channel.

7.2.3 Simulación

Primero realizamos una simulación en la que no ocurren errores. Para ello inicializamos el parametro FailRate del modelo Channel. Al recibir datos en su entrada el Sender los envía y espera las confirmaciones. Vemos que las salidas corresponden a las entradas.

Entrada del sistema

00:00:05:00	input	1
00:00:14:00	input	2
00:00:27:00	input	3

Salida del sistema

00:00:05:000	output	1.00000
00:00:14:000	output	2.00000

```
00:00:27:000 output 3.00000
```

Analizando los comportamientos de los modelos vemos que son correctos: El canal deja pasar todos los mensajes que le llegan inmediatamente.

```
? 00:00:05:000 : input , 1
E 00:00:05:000 : wait , ok ( failrate=0) ( msg=1)
O 00:00:05:000 : output , 1
I 00:00:05:000 : ok , wait ( failrate=0) ( msg=1)
? 00:00:27:000 : input , 3
E 00:00:27:000 : wait , ok ( failrate=0) ( msg=3)
O 00:00:27:000 : output , 3
```

Luego intentamos con un error del 50%. Se puede ver que al principio el modelo sender funciona normalmente, pero el channel falla y sender no recibe confirmación del primer mensaje, por lo que reintenta enviarlo cada 10 segundos.

Entrada

```
00:00:5:00 input 1
00:00:50:00 input 2
00:01:30:00 input 3
```

Sender

```
C 00:00:00:000 : start , (flag0=0) (flag1=0) (msg0=0) (msg1=0)
? 00:00:05:000 : i , 1
E 00:00:05:000 : start , sendingm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
O 00:00:05:000 : m0 , 1
I 00:00:05:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
I 00:00:15:000 : sentm0 , sendingm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
O 00:00:15:000 : m0 , 1
I 00:00:15:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
I 00:00:25:000 : sentm0 , sendingm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
O 00:00:25:000 : m0 , 1
I 00:00:25:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
I 00:00:35:000 : sentm0 , sendingm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
O 00:00:35:000 : m0 , 1
I 00:00:35:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=1) (msg1=0)
? 00:00:35:000 : a , 0
```

En este instante receiver confirma el envío exitoso y sender recibe la confirmación del mensaje 0.

```
E 00:00:35:000 : sentm0 , waitmsg1 (flag0=0) (flag1=0) (msg0=1) (msg1=0)
? 00:00:50:000 : i , 2
E 00:00:50:000 : waitmsg1 , m0ok (flag0=0) (flag1=1) (msg0=1) (msg1=2)
O 00:00:50:000 : m1 , 2
I 00:00:50:000 : m0ok , sentm1 (flag0=0) (flag1=1) (msg0=1) (msg1=2)
? 00:00:50:000 : a , 1
```

Como el canal m0 es confiable el siguiente mensaje llego inmediatamente.

```
E 00:00:50:000 : sentm1 , waitmsg0 (flag0=0) (flag1=0) (msg0=1) (msg1=2)
? 00:01:30:000 : i , 3
E 00:01:30:000 : waitmsg0 , mlok (flag0=1) (flag1=0) (msg0=3) (msg1=2)
O 00:01:30:000 : m0 , 3
I 00:01:30:000 : mlok , sentm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
I 00:01:40:000 : sentm0 , sendingm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
O 00:01:40:000 : m0 , 3
I 00:01:40:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
I 00:01:50:000 : sentm0 , sendingm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
O 00:01:50:000 : m0 , 3
I 00:01:50:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
I 00:02:00:000 : sentm0 , sendingm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
O 00:02:00:000 : m0 , 3
I 00:02:00:000 : sendingm0 , sentm0 (flag0=1) (flag1=0) (msg0=3) (msg1=2)
```

```
? 00:02:00:000 : a , 0
E 00:02:00:000 : sentm0 , waitmsg1 (flag0=0) (flag1=0) (msg0=3) (msg1=2)
```

El siguiente mensaje también tuvo varios reintentos. Veamos el comportamiento de Channel. Para el primer mensaje los dos primeros intentos fallan, pero el tercero es exitoso.

```
C 00:00:00:000 : wait , (failrate=5) (msg=0) (r=0)
? 00:00:05:000 : input , 1
E 00:00:05:000 : wait , fail (failrate=5) (msg=0) (r=3.91339)
I 00:00:05:000 : fail , wait (failrate=5) (msg=0) (r=3.91339)
? 00:00:15:000 : input , 1
E 00:00:15:000 : wait , fail (failrate=5) (msg=0) (r=3.91953)
I 00:00:15:000 : fail , wait (failrate=5) (msg=0) (r=3.91953)
? 00:00:25:000 : input , 1
E 00:00:25:000 : wait , fail (failrate=5) (msg=0) (r=5.44049)
I 00:00:25:000 : fail , wait (failrate=5) (msg=0) (r=5.44049)
? 00:00:35:000 : input , 1
E 00:00:35:000 : wait , ok (failrate=5) (msg=1) (r=3.59857)
O 00:00:35:000 : output , 1
```

La salida del sistema es:

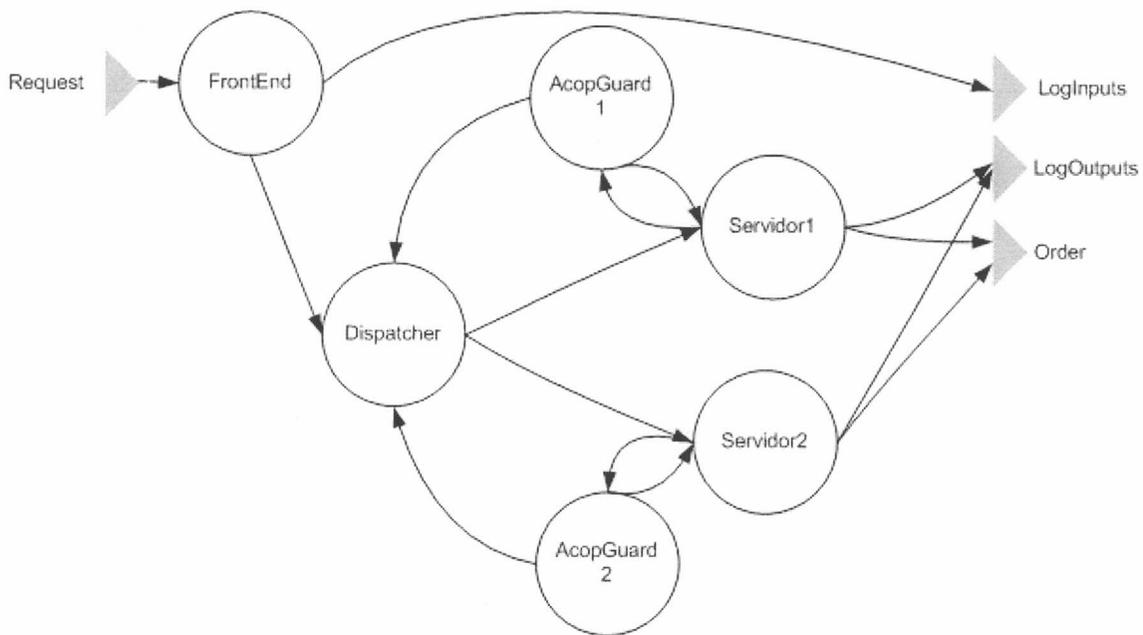
```
00:00:35:000 output      1.00000
00:00:50:000 output      2.00000
00:02:00:000 output      3.00000
```

Lo que indica que los mensajes m0 han sido retransmitidos al ocurrir errores, por lo que se produjo una demora hasta que pudieron ser transmitidos exitosamente en reintentos. El mensaje m1 no fue afectado pues el canal no tenía error.

7.3 Load balancer:

En este caso tenemos el modelado de un balanceador de carga http. El funcionamiento es el siguiente: Cuando llega un requerimiento HTTP al balanceador, éste debe decidir a cual de los 2 servidores web se lo asigna. Para ello chequea en su variable de estado cual fue el último servidor que procesó un requerimiento, y con esa información se lo pasa al otro.

El modelo acoplado puede verse en la siguiente figura:



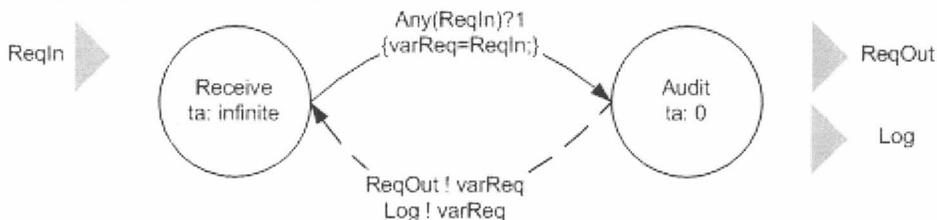
Hay 4 puertos:

- ξ Request: por este puerto llegan los requerimientos
- ξ Loginputs: aquí se registran los ingresos de requerimientos
- ξ Logoutputs: por este puerto salen los requerimientos una vez procesados
- ξ Order: registra el orden en el que fueron atendiendo los servidores a los requerimientos

El diseño del modelo consta de :

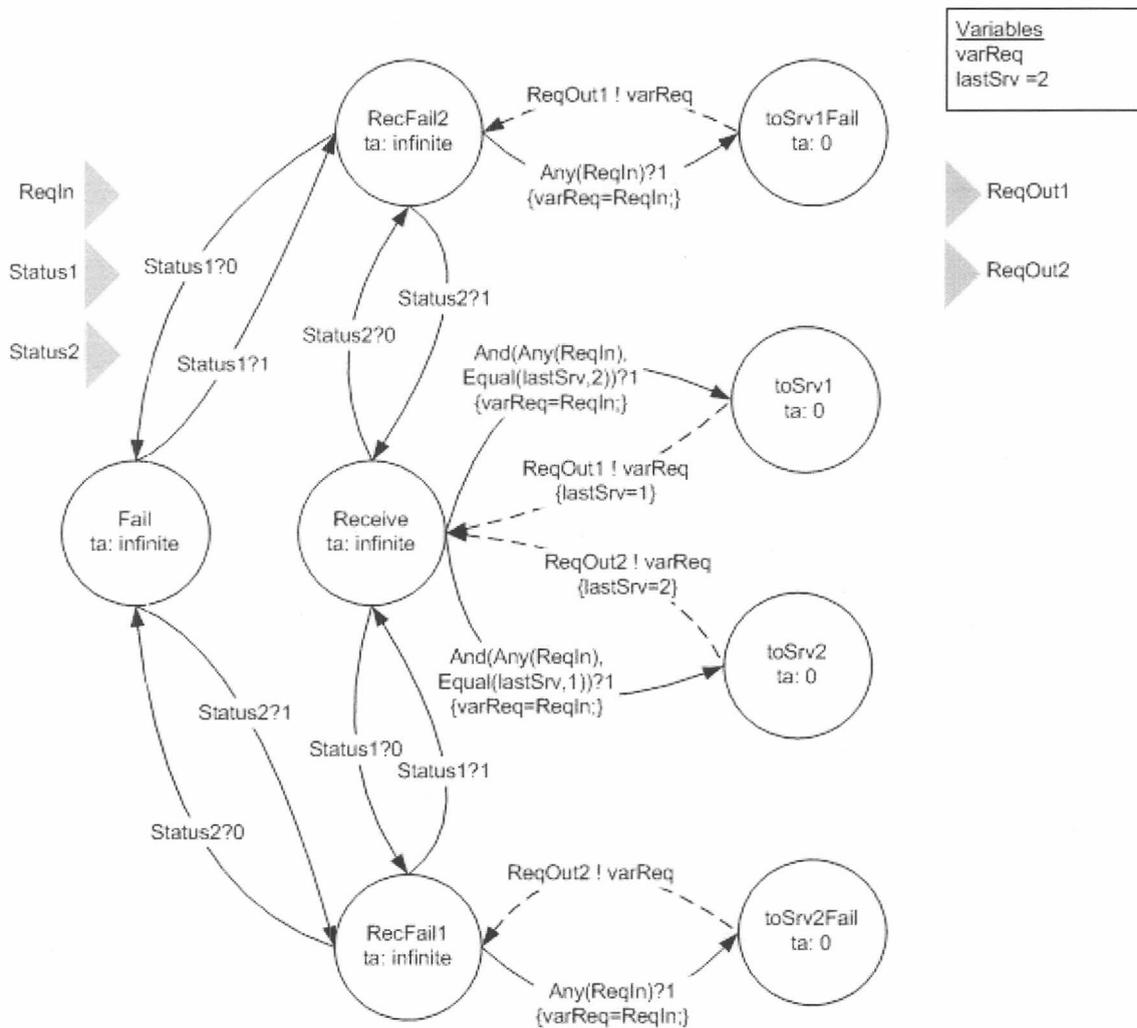
- ξ Un modelo atómico llamado FrontEnd que es el que recibe los request y registra la entrada de los mismos.
- ξ Un modelo atómico Dispatcher se encarga de enviarle el requerimiento al servidor en cuestión
- ξ Dos modelos atómicos de Servidor que procesan el request y lo envían al port de salida del modelo acoplado
- ξ Dos modelos acoplados llamados AcopGuard que monitorean el estado de los servidores, notificando en caso de falla al Dispatcher. Este modelo acoplado está compuesto por TickGen que es un generador de ticks, y el modelo Guardian, que a partir de cada tick chequea el estado del servidor asociado.

Modelo atómico FrontEnd:



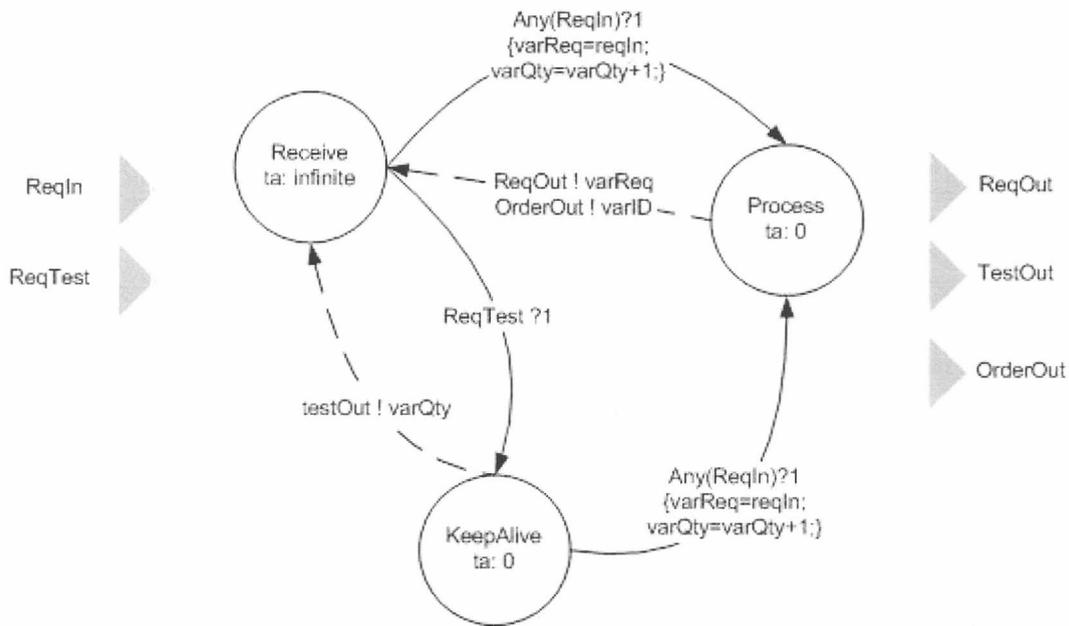
Para cada valor ingresado a través del puerto ReqIn, se almacena ese valor en una variable y luego se lo emite por los 2 puertos de salida

Modelo atómico Dispatcher



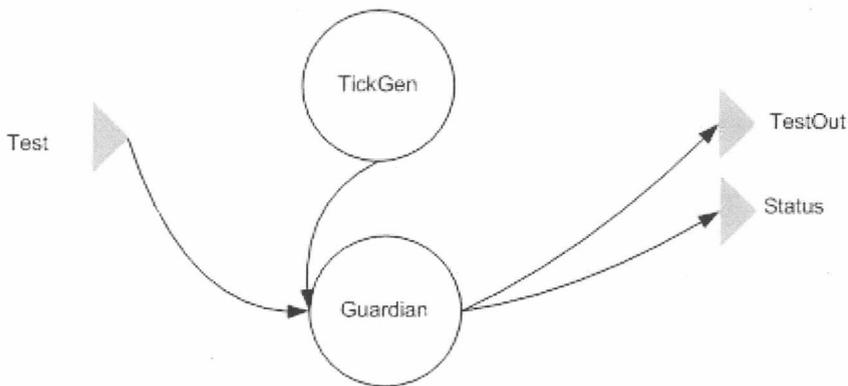
Este modelo está normalmente en el estado Receive a la espera de un request por el port ReqIn, en cuyo caso se lo envía al estado toSrv1 o toSrv2 alternativamente. Estos estados retornan inmediatamente al estado Receive con un output hacia el modelo atómico Servidor1 o Servidor2 respectivamente. En caso de que por el port Status1 o Status2 llegue un valor 0 (Falla) el nuevo estado de espera será RecFail1 o RecFail2. Allí se mantendrá recibiendo requests y enviandoselos al Servidor sobreviviente hasta que llegue un status avisando de que el servidor caído está nuevamente operativo o que falló el otro servidor, en cuyo caso se va al estado Fail en el cual no procesa más requests.

Modelo atómico Servidor (Servidor1 y Servidor2)



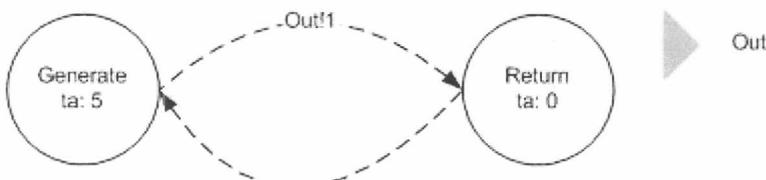
Este modelo atómico recibe dos tipos de requests, los que deben ser procesados y respondidos y los de test para monitorear la salud de este servidor. En el primer caso procesa el requerimiento y luego retorna

Modelo acoplado AcopGuard (Acopguard1 y Acopguard2):



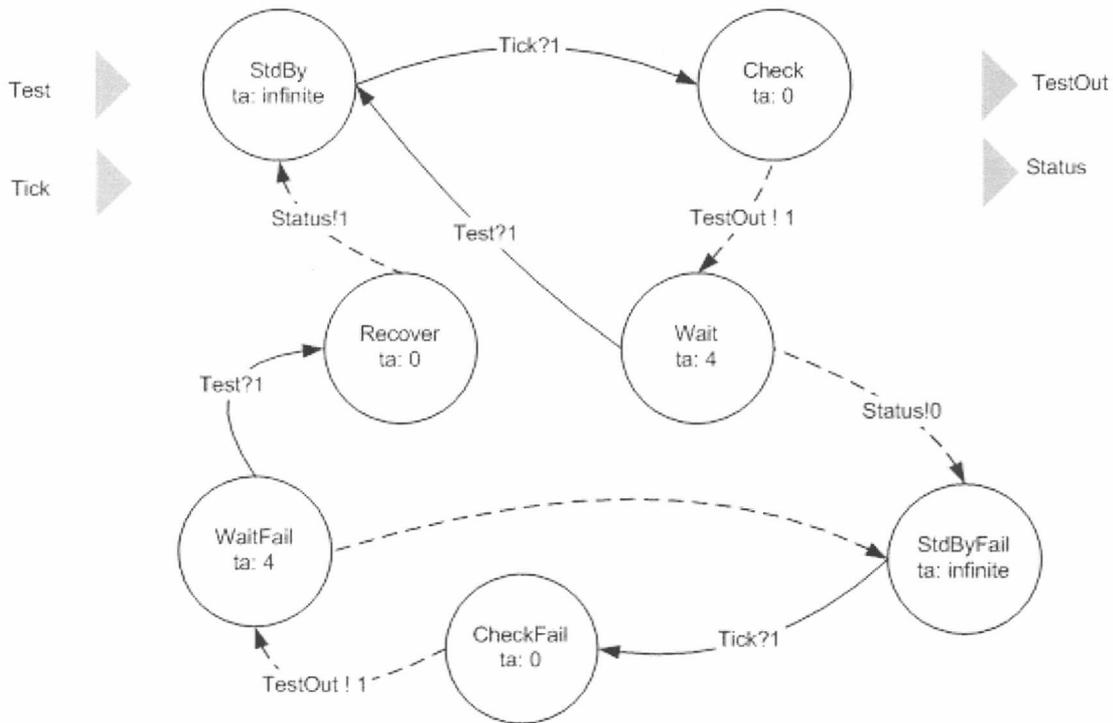
Este modelo se encarga de enviar cada cierto intervalo de tiempo un mensaje al modelo del servidor y esperar su respuesta. En caso de no recibir una respuesta después de un período de tiempo, le avisará al modelo Dispatcher a través del puerto de salida "status" y seguirá intentando contactar al servidor hasta que este responda

Modelo atómico TickGen (parte del acoplado Acopguard)



Este modelo se encarga de emitir un mensaje al modelo guardian cada un intervalo de tiempo, para activar el monitoreo del servidor.

Modelo atómico Guardian (parte del acoplado Acopguard)



Este modelo inicia su funcionamiento en el estado StdBy. Cada vez que llega un mensaje por el port Tick, emite un mensaje de monitoreo al servidor asociado por el puerto TestOut. Si no llega una respuesta en un intervalo de tiempo, emite un mensaje de falla de ese servidor al modelo del Dispatcher a través del puerto Status y se queda en un estado llamado StdByFail. Desde este momento seguirá monitoreando al servidor hasta que este responda, y en ese caso emitirá un mensaje de status OK al dispatcher y volverá al estado StdBy.

Especificación del modelo GGAD

LoadBalancer.ma

```

[Top]
components : FrontEnd@GGad Dispatcher@GGad Acopguard1 Acopguard2
Servidor1@GGad Servidor2@GGad
out : Loginputs Logoutputs Order
in : Request
Link : Request ReqIn@FrontEnd
Link : ReqOut@FrontEnd reqIn@Dispatcher
Link : Log@FrontEnd Loginputs
Link : reqOut1@Dispatcher reqIn@Servidor1
Link : reqOut2@Dispatcher reqIn@Servidor2
Link : status@Acopguard1 status1@Dispatcher
Link : status@Acopguard2 status2@Dispatcher
Link : testOut@Acopguard1 reqTest@Servidor1
Link : testOut@Servidor1 test@Acopguard1
Link : testOut@Acopguard2 reqTest@Servidor2
Link : testOut@Servidor2 test@Acopguard2
    
```

```
Link : orderOut@Servidor1 Order
Link : orderOut@Servidor2 Order
Link : reqOut@Servidor1 Logoutputs
Link : reqOut@Servidor2 Logoutputs

[FrontEnd]
source : FrontEnd.cdd

[Dispatcher]
source : Dispatcher.cdd

[Acopguard1]
components : TickGen@GGad Guardian@GGad
out : testOut status
in : test
Link : test test@Guardian
Link : outTick@TickGen tick@Guardian
Link : testOut@Guardian testOut
Link : status@Guardian status

[TickGen]
source : TickGen.cdd

[Guardian]
source : Guardian.cdd

[Acopguard2]
components : TickGen@GGad Guardian@GGad
out : testOut status
in : test
Link : test test@Guardian
Link : outTick@TickGen tick@Guardian
Link : testOut@Guardian testOut
Link : status@Guardian status

[TickGen]
source : TickGen.cdd

[Guardian]
source : Guardian.cdd

[Servidor1]
source : Servidor1.cdd

[Servidor2]
source : Servidor2.cdd
```

Frontend.cdd

```
[FrontEnd]
in: ReqIn
out: ReqOut Log
var: varReq
state: Receive Audit
initial:Receive
int: Audit Receive ReqOut!varReq
ext: Receive Audit Any(ReqIn)?1 {varReq = reqIn;}
Receive: 0:0:1000:0
Audit: 0:0:0:0
varReq:0
```

Dispatcher.cdd

```
[Dispatcher]
in: reqIn status1 status2
out: reqOut1 reqOut2
var: varReq lastSrv
state: Fail Receive RecFail1 RecFail2 toSrv1 toSrv1Fail toSrv2Fail
toSrv2
initial:Receive
int: toSrv1 Receive reqOut1!varReq {lastSrv = varReq;}
int: toSrv2 Receive reqOut2!varReq {lastSrv = varReq;}
int: toSrv1Fail RecFail2 reqOut1!varReq
int: toSrv2Fail RecFail1 reqOut2!varReq
ext: Receive toSrv1 And(Any(ReqIn),Equal(lastSrv,2))?1 {varReq =
reqIn;}
ext: Receive toSrv2 And(Any(ReqIn),Equal(lastSrv,1))?1 {varReq =
reqIn;}
ext: Receive RecFail2 Value(status2)?0
ext: RecFail2 Receive Value(status2)?1
ext: Receive RecFail1 Value(status1)?0
ext: RecFail1 Receive Value(status1)?1
ext: RecFail1 Fail Value(status2)?0
ext: Fail RecFail1 Value(status2)?1
ext: RecFail2 Fail Value(status1)?0
ext: Fail RecFail2 Value(status1)?1
ext: RecFail2 toSrv1Fail Value(reqIn)?1 {varReq = reqIn;}
ext: RecFail1 toSrv2Fail Value(reqIn)?1 {varReq = reqIn;}
Fail: 0:0:1000:0
Receive: 0:0:1000:0
RecFail1: 0:0:1000:0
RecFail2: 0:0:1000:0
toSrv1: 0:0:0:0
toSrv1Fail: 0:0:0:0
toSrv2Fail: 0:0:0:0
toSrv2: 0:0:0:0
varReq:0
lastSrv:2
```

Servidor1.cdd

```
[Servidor1]
in: reqIn reqTest
out: reqOut testOut orderOut
var: varReq varID varQty
state: Receive Process keepAlive
initial:Receive
int: Process Receive reqOut!varReq orderOut!varID
int: keepAlive Receive testOut!varQty
ext: Receive Process Any(reqIn)?1 {varReq = reqIn;varQty =
add(varQty,1);}
ext: Receive keepAlive Value(reqTest)?1
ext: keepAlive Process Any(reqIn)?1 {varReq = reqIn;varQty =
add(varQty,1);}
Receive: 0:0:1000:0
Process: 0:0:0:0
keepAlive: 0:0:0:0
varReq:0
varID:1
varQty:0
```

Servidor2.cdd

```
[Servidor2]
in: reqIn reqTest
out: reqOut testOut orderOut
var: varReq varID varQty
state: Receive Process keepAlive
initial:Receive
int: Process Receive reqOut!varReq orderOut!varID
int: keepAlive Receive testOut!varQty
ext: Receive Process Any(reqIn)?1 {varReq = reqIn;varQty =
add(varQty,1);}
ext: Receive keepAlive Value(reqTest)?1
ext: keepAlive Process Any(reqIn)?1 {varReq = reqIn;varQty =
add(varQty,1);}
Receive: 0:0:1000:0
Process: 0:0:0:0
keepAlive: 0:0:0:0
varReq:0
varID:2
varQty:0
```

TickGen.cdd

```
[TickGen]
out: outTick
state: Generate Return
initial:Generate
int: Generate Return outTick!1
int: Return Generate
Generate: 0:0:5:0
Return: 0:0:0:0
```

Guardian.cdd

```
[Guardian]
in: test tick
out: testOut status
state: StdBy Check Recover Wait StdByFail CheckFail WaitFail
initial:StdBy
int: Check Wait testOut!1
int: Wait StdByFail status!0
int: CheckFail WaitFail testOut!1
int: WaitFail StdByFail
int: Recover StdBy status!1
ext: StdBy Check Value(tick)?1
ext: Wait StdBy Value(test)?1
ext: StdByFail CheckFail Value(tick)?1
ext: WaitFail Recover Value(test)?1
StdBy: 0:0:1000:0
Check: 0:0:0:0
Recover: 0:0:0:0
Wait: 0:0:4:0
StdByFail: 0:0:1000:0
CheckFail: 0:0:0:0
WaitFail: 0:0:4:0
```

Para ejecutar la simulación de este modelo se ejecutó el comando:

```
cd++ -t00:00:30:00 -mloadBalancer.ma -eloadBalancer.ev -
lloadBalancer.log -oloadBalancer.out
```

El siguiente archivo muestra los eventos definidos para esta ejecución de la simulación:

LoadBalancer.ev

```
00:00:02:00 request 1
00:00:04:00 request 3
00:00:06:00 request 5
00:00:08:00 request 7
00:00:10:00 request 9
00:00:12:00 request 2
00:00:14:00 request 4
00:00:16:00 request 6
00:00:18:00 request 8
00:00:20:00 request 10
00:00:22:00 request 12
00:00:24:00 request 14
00:00:26:00 request 16
00:00:28:00 request 18
```

El siguiente es el contenido del archivo LoadBalancer.out, que muestra los valores emitidos por el modelo acoplado en los puertos order y logoutputs. (Aquí ordenado para mayor simplicidad en columnas distintas)

```
00:00:02:000 order 1.00000
00:00:04:000 order 2.00000
00:00:06:000 order 1.00000
00:00:08:000 order 2.00000
00:00:10:000 order 1.00000
00:00:12:000 order 2.00000
00:00:14:000 order 1.00000
00:00:16:000 order 2.00000
00:00:18:000 order 1.00000
00:00:20:000 order 2.00000
00:00:22:000 order 1.00000
00:00:24:000 order 2.00000
00:00:26:000 order 1.00000
00:00:28:000 order 2.00000
```

```
00:00:02:000 logoutputs 1.00000
00:00:04:000 logoutputs 3.00000
00:00:06:000 logoutputs 5.00000
00:00:08:000 logoutputs 7.00000
00:00:10:000 logoutputs 9.00000
00:00:12:000 logoutputs 2.00000
00:00:14:000 logoutputs 4.00000
00:00:16:000 logoutputs 6.00000
00:00:18:000 logoutputs 8.00000
00:00:20:000 logoutputs 10.00000
00:00:22:000 logoutputs 12.00000
00:00:24:000 logoutputs 14.00000
00:00:26:000 logoutputs 16.00000
00:00:28:000 logoutputs 18.00000
```

Podemos observar que el orden de atención de requerimientos de los servidores fue balanceado 50% a cada uno como estaba previsto. En el caso de los valores emitidos por el port Logoutputs podemos chequear que son los mismos valores y en el mismo orden que los del archivo de eventos, comprobándose también que hizo lo esperado.

- ξ En los archivos “NombreDeModelo.translog” podemos ver como fueron cambiando de estado los modelos atómicos. En este caso, y por simplicidad, veremos unicamente los correspondientes al dispatcher, el servidor1 y el guardian1 para los primeros 10 segundos de la simulación.

Dispatcher.translog

```
C 00:00:00:000 : receive , (lastsrv=2) (varreq=0)
? 00:00:02:000 : reqin , 1
E 00:00:02:000 : receive , tosrv1 (lastsrv=2) (varreq=1)
O 00:00:02:000 : reqout1 , 1
I 00:00:02:000 : tosrv1 , receive (lastsrv=1) (varreq=1)
? 00:00:04:000 : reqin , 3
E 00:00:04:000 : receive , tosrv2 (lastsrv=1) (varreq=3)
O 00:00:04:000 : reqout2 , 3
I 00:00:04:000 : tosrv2 , receive (lastsrv=2) (varreq=3)
? 00:00:06:000 : reqin , 5
```

```

E 00:00:06:000 : receive , tosrv1 (lastsrv=2) (varreq=5)
O 00:00:06:000 : reqout1 , 5
I 00:00:06:000 : tosrv1 , receive (lastsrv=1) (varreq=5)
? 00:00:08:000 : reqin , 7
E 00:00:08:000 : receive , tosrv2 (lastsrv=1) (varreq=7)
O 00:00:08:000 : reqout2 , 7
I 00:00:08:000 : tosrv2 , receive (lastsrv=2) (varreq=7)
? 00:00:10:000 : reqin , 9
E 00:00:10:000 : receive , tosrv1 (lastsrv=2) (varreq=9)
O 00:00:10:000 : reqout1 , 9
I 00:00:10:000 : tosrv1 , receive (lastsrv=1) (varreq=9)
.....
    
```

Servidor1.translog

```

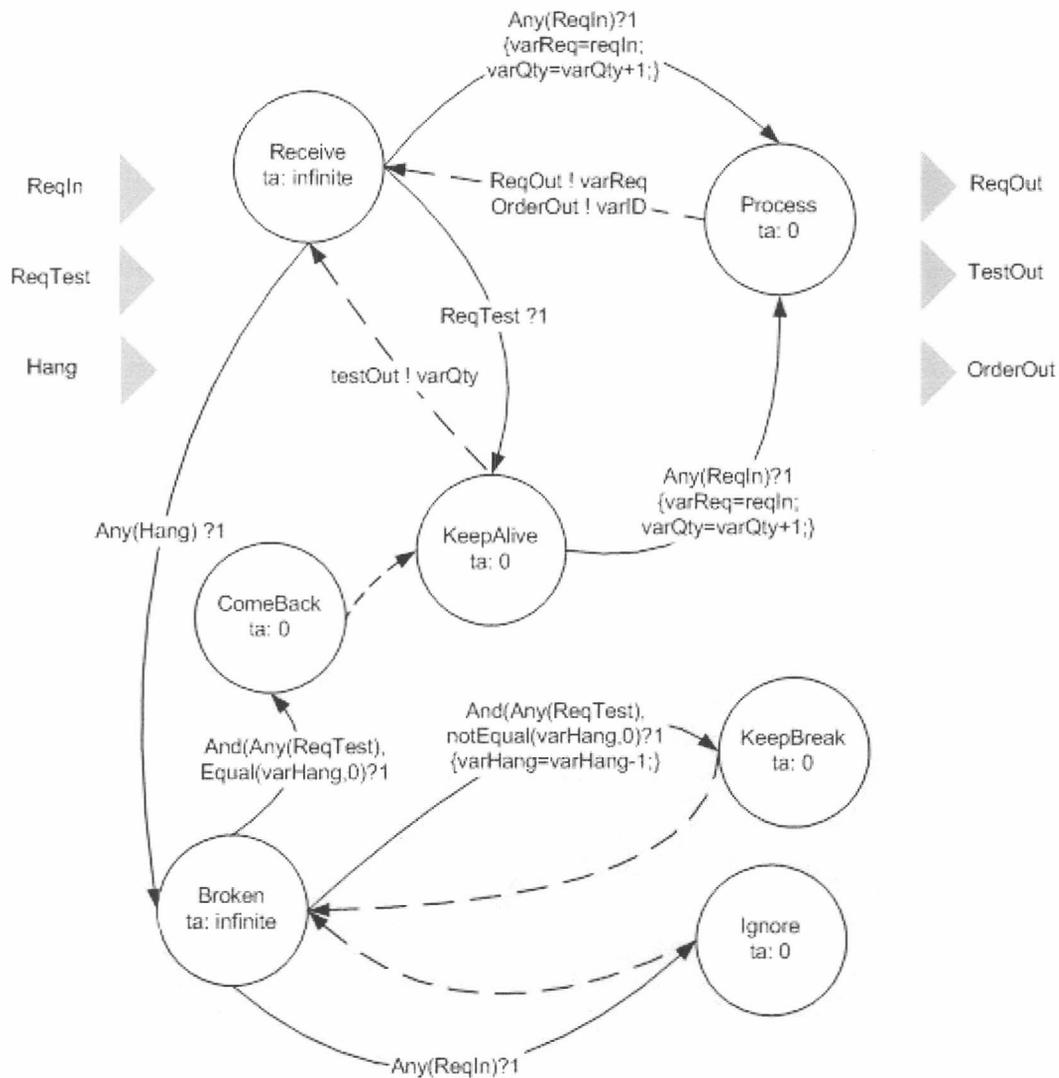
C 00:00:00:000 : receive , (varid=1) (varqty=0) (varreq=0)
? 00:00:02:000 : reqin , 1
E 00:00:02:000 : receive , process (varid=1) (varqty=1) (varreq=1)
O 00:00:02:000 : orderout , 1
O 00:00:02:000 : reqout , 1
I 00:00:02:000 : process , receive (varid=1) (varqty=1) (varreq=1)
? 00:00:05:000 : reqtest , 1
E 00:00:05:000 : receive , keepalive (varid=1) (varqty=1) (varreq=1)
O 00:00:05:000 : testout , 1
I 00:00:05:000 : keepalive , receive (varid=1) (varqty=1) (varreq=1)
? 00:00:06:000 : reqin , 5
E 00:00:06:000 : receive , process (varid=1) (varqty=2) (varreq=5)
O 00:00:06:000 : orderout , 1
O 00:00:06:000 : reqout , 5
I 00:00:06:000 : process , receive (varid=1) (varqty=2) (varreq=5)
? 00:00:10:000 : reqtest , 1
E 00:00:10:000 : receive , keepalive (varid=1) (varqty=2) (varreq=5)
O 00:00:10:000 : testout , 2
I 00:00:10:000 : keepalive , receive (varid=1) (varqty=2) (varreq=5)
? 00:00:10:000 : reqin , 9
E 00:00:10:000 : receive , process (varid=1) (varqty=3) (varreq=9)
O 00:00:10:000 : orderout , 1
O 00:00:10:000 : reqout , 9
I 00:00:10:000 : process , receive (varid=1) (varqty=3) (varreq=9)
.....
    
```

Guardian1.translog

```

C 00:00:00:000 : stdby
? 00:00:05:000 : tick , 1
E 00:00:05:000 : stdby , check
O 00:00:05:000 : testout , 1
I 00:00:05:000 : check , wait
? 00:00:05:000 : test , 1
E 00:00:05:000 : wait , stdby
? 00:00:10:000 : tick , 1
E 00:00:10:000 : stdby , check
O 00:00:10:000 : testout , 1
I 00:00:10:000 : check , wait
? 00:00:10:000 : test , 2
E 00:00:10:000 : wait , stdby
? 00:00:15:000 : tick , 1
E 00:00:15:000 : stdby , check
O 00:00:15:000 : testout , 1
I 00:00:15:000 : check , wait
? 00:00:15:000 : test , 4
E 00:00:15:000 : wait , stdby
.....
    
```

Para evaluar el correcto funcionamiento del dispatcher balanceando carga entre los servidores, modificamos la implementación del servidor para permitir simular una falla. El nuevo modelo atómico servidor, con la parte nueva grisada, es el siguiente



Ahora el modelo tiene un nuevo puerto de entrada llamado Hang. En caso de recibir un valor n por dicho puerto, ignorará los siguientes n mensajes de monitoreo (obviamente también ignorará los mensajes del dispatcher)

Para la siguiente simulación, se generó un nuevo port del modelo LoadBalancer llamado Hang1 que está conectado al port Hang del Servidor1.

El nuevo archivo LoadBalancer.ev para la simulación es idéntico al anterior, salvo que en el instante 00:09:00 introduce un evento de falla poniendo un valor 2 en el puerto Hang1. Esto significa que el servidor1 debería ignorar mensajes por el lapso de 2 eventos en el port ReqTest

Loadbalancer.ev

```

00:00:02:00 request 1
00:00:04:00 request 3
00:00:06:00 request 5
00:00:08:00 request 7
00:00:09:00 hang1 2
00:00:10:00 request 9
00:00:12:00 request 2
00:00:14:00 request 4
00:00:16:00 request 6
00:00:18:00 request 8
00:00:20:00 request 10
00:00:22:00 request 12
    
```

```
00:00:24:00 request 14
00:00:26:00 request 16
00:00:28:00 request 18
```

Luego de correr la simulación puede verse en el archivo de salida el resultado:

LoadBalancer.out

```
00:00:02:000 order      1.00000
00:00:02:000 logoutputs  1.00000
00:00:04:000 order      2.00000
00:00:04:000 logoutputs  3.00000
00:00:06:000 order      1.00000
00:00:06:000 logoutputs  5.00000
00:00:08:000 order      2.00000
00:00:08:000 logoutputs  7.00000
00:00:12:000 order      2.00000
00:00:12:000 logoutputs  2.00000
00:00:14:000 order      2.00000
00:00:14:000 logoutputs  4.00000
00:00:16:000 order      2.00000
00:00:16:000 logoutputs  6.00000
00:00:18:000 order      2.00000
00:00:18:000 logoutputs  8.00000
00:00:20:000 order      2.00000
00:00:20:000 logoutputs 10.00000
00:00:22:000 order      1.00000
00:00:22:000 logoutputs 12.00000
00:00:24:000 order      2.00000
00:00:24:000 logoutputs 14.00000
00:00:26:000 order      1.00000
00:00:26:000 logoutputs 16.00000
00:00:28:000 order      2.00000
00:00:28:000 logoutputs 18.00000
```

Puede notarse que en el instante 09:00 de la simulación cuando entro el valor 2 por el puerto Hang1, el modelo atómico Servidor1 dejó de atender requerimientos del dispatcher. En ese momento el valor 9 que ingresó por el port Request en el instante 10:00 le correspondía al servidor1 y justamente por eso no se lo ve en el Output (fue ignorado).

Luego el Guardian1 detectó la falla del servidor1 y le envió al dispatcher el aviso en el instante 14:00

Guardian1.translog

```
? 00:00:05:000 : tick , 1
E 00:00:05:000 : stdby , check
O 00:00:05:000 : testout , 1
I 00:00:05:000 : check , wait
? 00:00:05:000 : test , 1
E 00:00:05:000 : wait , stdby
? 00:00:10:000 : tick , 1
E 00:00:10:000 : stdby , check
O 00:00:10:000 : testout , 1
I 00:00:10:000 : check , wait
O 00:00:14:000 : status , 0
I 00:00:14:000 : wait , stdbyfail
? 00:00:15:000 : tick , 1
E 00:00:15:000 : stdbyfail , checkfail
O 00:00:15:000 : testout , 1
I 00:00:15:000 : checkfail , waitfail
I 00:00:19:000 : waitfail , stdbyfail
? 00:00:20:000 : tick , 1
E 00:00:20:000 : stdbyfail , checkfail
```

```
O 00:00:20:000 : testout , 1
I 00:00:20:000 : checkfail , waitfail
? 00:00:20:000 : test , 2
E 00:00:20:000 : waitfail , recover
O 00:00:20:000 : status , 1
I 00:00:20:000 : recover , stdby
? 00:00:25:000 : tick , 1
E 00:00:25:000 : stdby , check
O 00:00:25:000 : testout , 1
I 00:00:25:000 : check , wait
? 00:00:25:000 : test , 3
E 00:00:25:000 : wait , stdby
? 00:00:30:000 : tick , 1
E 00:00:30:000 : stdby , check
O 00:00:30:000 : testout , 1
I 00:00:30:000 : check , wait
? 00:00:30:000 : test , 4
E 00:00:30:000 : wait , stdby
```

A partir de allí, el dispatcher solo envió requerimientos al servidor2, los cuales pueden verse desde el instante 14:00 hasta el 20:00 en el output.

En el instante 20:00 el Guardian1 detectó que el servidor1 volvió a responder y se lo avisó al dispatcher, el cual recomenzó a balancear los requerimientos entre los dos servidores.

8 Conclusiones y desarrollo futuro

Hemos presentado una nueva forma de especificar modelos DEVS a partir de una definición gráfica de los componentes de los modelos, y su traducción a GADscript, un lenguaje formal de especificación de modelos atómicos DEVS.

De esta manera, se ha mostrado como diseñar y especificar modelos DEVS ejecutables por CD++ sin necesidad de programar en C++, facilitando entonces la actividad de generar nuevos modelos a un público no habituado a lenguajes de programación como el C++.

Implementamos esta especificación creando la herramienta GGADTool para asistir al usuario en el diseño gráfico y traducción automatizada de modelos a GADscript, y modificando a CD++ para agregarle la funcionalidad de interpretación y ejecución de GADscript.

El nivel de complejidad asociado a utilizar técnicas de simulación DEVS para modelar la realidad se ha simplificado sin por eso descuidar en demasía el poder expresivo de los modelos resultantes.

Consideramos que ha partir de este trabajo se abren nuevos temas para desarrollar:

- ξ Es posible ampliar las capacidades expresivas de la representación gráfica y GADscript, permitiendo expresiones mas complejas y funciones definibles por el usuario. Posiblemente de la experimentación se haga evidente la necesidad de nuevas estructuras del lenguaje que sin afectar la expresibilidad de los modelos en el formalismo DEVS simplifiquen la notación de los mismos.
- ξ Las capacidades de la herramienta de diseño pueden ser ampliadas en lo que respecta a interfaz de usuario. Tambien puede desarrollarse la exportación de modelos a otros motores de simulación más allá de CD++.
- ξ La ejecución de modelos GADscript puede ser implementada en otros motores de simulación, puesto que no tiene dependencias de CD++.

Referencias y bibliografía

- [Cho94a] CHOW, ALEX C.; ZEIGLER, BERNARD P. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Winter Simulation Conference Proceedings*, Orlando, Florida, 1994. SCS.
- [Cho94b] CHOW, ALEX C. ; KIM, DOO H.; ZEIGLER, BERNARD P. "Abstract Simulator for the parallel DEVS formalism". *AI, Simulation, and Planning in High Autonomy Systems*. Dec., 1994.
- [BAR98] BARYLKO, A., BEYOGLONIAN, J. "CD++: an environment for Cell-DEVS modelling and simulation" . Tesis de Licenciatura. Departamento de Computación FCEyN – UBA. 1998.
- [Mar97] MARTIN, D.; MCBRAYER, T.; RADHAKRISHNAN, R.; WILSEY, P. "TimeWarp Parallel Discrete Event Simulator". Technical Report. Computer Architecture Design Laboratory, University of Cincinnati. December 1997.
- [Hol91] HOLZMANN, GERARD J. "Design and Validation of Computer Protocols". Prentice Hall. 1991.
- [Tro01] TROCCOLI, ALEJANDRO. "Modificaciones a CD++ para simulación paralela y distribuida de modelos Cell-DEVS". Tesis de Licenciatura. Departamento de Computación FCEyN – UBA. 2001.
- [Wai96] WAINER, GABRIEL. "Introducción a la simulación de sistemas de eventos discretos". Technical Report 96-005. Departamento de Computación FCEyN – UBA. 1996.

[Wai99] RODRIGUEZ, D.; WAINER, GABRIEL. "New Extensions to the CD++ tool". In Proceedings of SCS Summer Multiconference on Computer Simulation. 1999.

[Wai00] WAINER, GABRIEL. "Improved cellular models with parallel Cell-DEVS". In Transactions of the SCS. June 2000.

[Wai01] WAINER, GABRIEL; GIAMBIASI, NORBERT. "Timed Cell-DEVS: modelling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", to be published by Springer-Verlag. 2001.

[Zei90] ZEIGLER, BERNARD P.. Object Oriented Simulation with Hierarchical, Modular Models. Academic Press, San Diego, California, 1990.

[Zei95] ZEIGLER, BERNARD P.; KIM, T.; PRAEHOFER, H; SONG H. "DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems". Hybrid Systems II. Lecture Notes in Computer Science 999 páginas 529-551. Springer 1995.

[Zei00] ZEIGLER, BERNARD P.; KIM, T.; PRAEHOFER, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.

1 APENDICE: EJEMPLO DE APLICACIÓN

1.1 Radar fotográfico:

1.1.1 Descripción

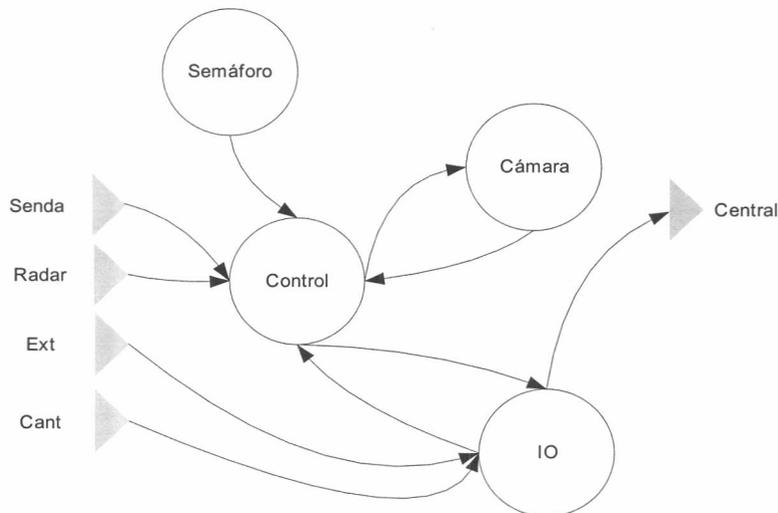
Este ejemplo modela el funcionamiento de un radar fotográfico para controlar la velocidad máxima permitida y el paso de semáforos en rojo.

Hay un controlador que recibe información de sensores en la senda peatonal que detectan la presencia de un vehículo, de un radar que toma la velocidad a los autos y del semáforo que informa la luz que tiene encendida.

A su vez el controlador comanda la cámara de fotográfica y envía una señal de alerta cuando se queda sin rollo para seguir sacando fotos.

1.1.2 Modelos

El modelo acoplado puede verse en la siguiente figura:



Puerto	Descripción
Senda (in)	Por este puerto son recibidas las señales que advierten de un vehículo sobre la senda peatonal.
Radar (in)	Advierte sobre un vehículo que supera la velocidad máxima permitida.
Ext (in)	Permite darle comandos al controlador (por ejemplo que ha sido reemplazado el rollo de fotos)
Cant (in)	Le avisa al controlador cuantas fotos tiene el nuevo rollo
Central (out)	Comunicación con la central para enviar alertas o reportes.

Modelo	Descripción
--------	-------------

Control	Administra el sistema de fotomultas.
Semáforo	Cicla entre la luz verde, amarilla y roja enviando información al controlador de cual es su luz actual
Cámara	Saca fotos
IO	Realiza tareas de comunicaciones

RadarFot.ma

```
[Top]
components : semaforo@GGad control@GGad camara@GGad IO@GGad
out : central
in : senda radar ext cant
Link : senda senda@control
Link : radar radar@control
Link : luz@semaforo semaforo@control
Link : foto@control take@camara
Link : ready@camara ready@control
Link : cant cant@IO
Link : empty@control empty@IO
Link : restart@IO full@control
Link : ext extern@IO
Link : central@IO central

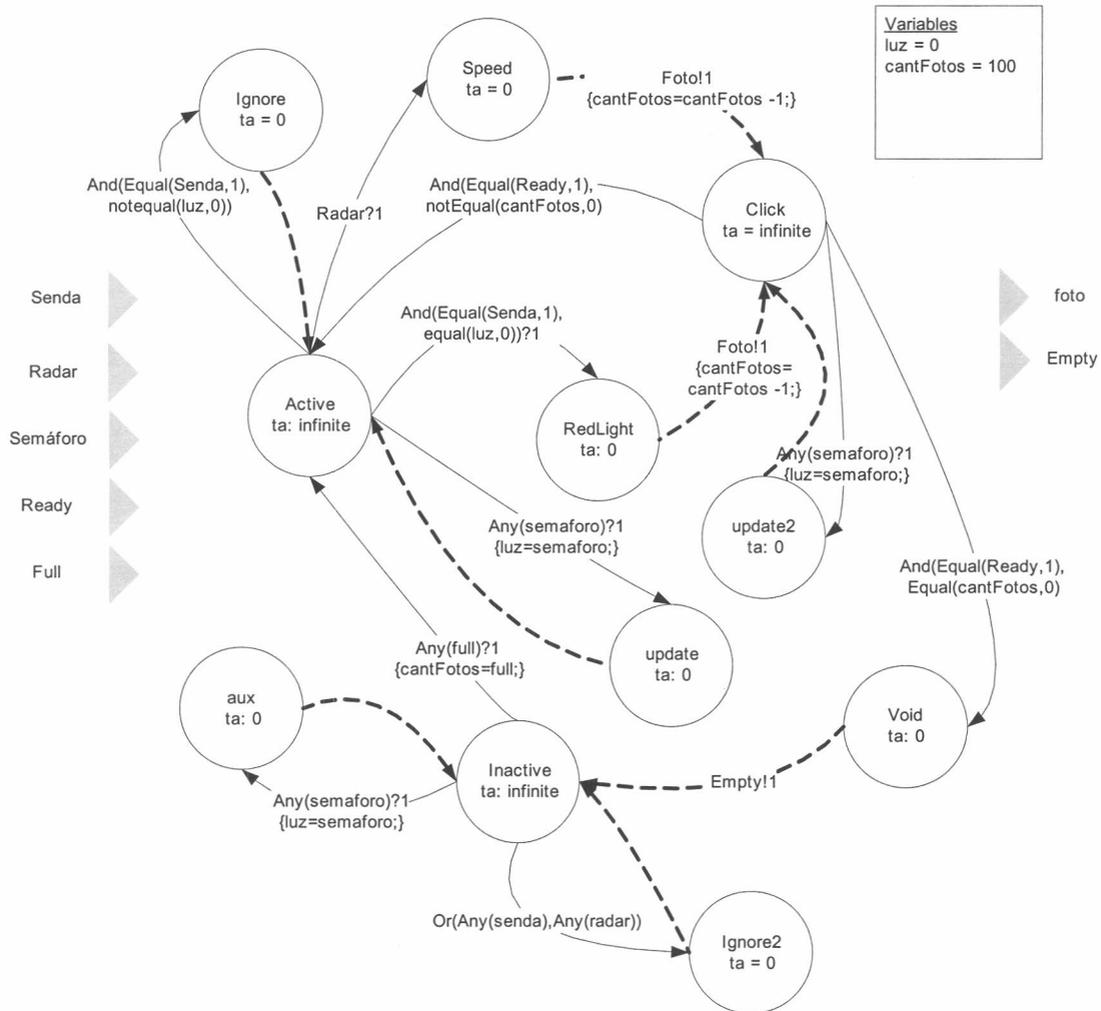
[semaforo]
source : semaforo.cdd

[control]
source : control.cdd

[camara]
source : camara.cdd

[IO]
source : IO.cdd
```

Modelo atómico Control:



Funcionamiento:

El modelo arranca en estado Active. En caso de recibir un valor en el radar indicando alta velocidad o sentir un vehículo sobre la senda peatonal al mismo tiempo que el semáforo está en rojo instruye a la cámara fotográfica para que lo registre y luego vuelve a la posición active. Si su contador de fotos disponibles llega a cero, envía el aviso a través del port Empty y espera a que le llegue el comando de rollo nuevo por el port Full. Cada vez que el semáforo cambia de luz, el modelo actualiza su variable luz para registrarlo.

Control.cdd

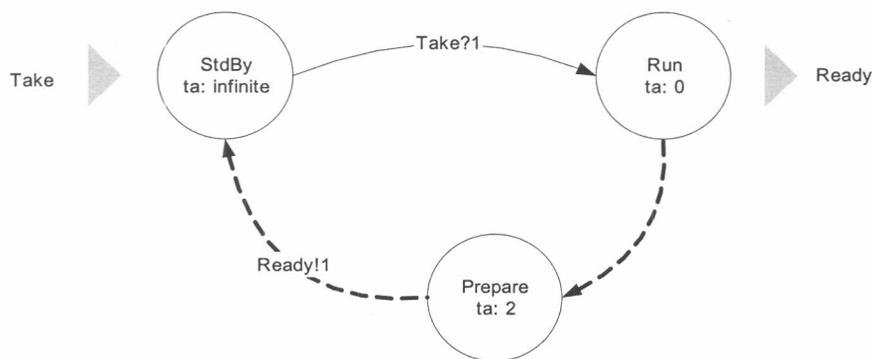
```
[control]
in: senda radar semaforo ready full
out: foto empty
var: luz cantfotos
state: active speed void inactive redlight update click update2
aux ignore ignore2
initial:active
int: speed click foto!1 {cantfotos = minus(cantfotos,1);}
int: redlight click foto!1 {cantfotos = minus(cantfotos,1);}
int: update2 click
int: void inactive empty!1
int: aux inactive
int: update active
int: ignore active
```

```

int: ignore2 inactive
ext: active speed Value(radar)?1
ext: click active and(equal(ready,1),notequal(cantfotos,0))?1
ext: active redlight And(Equal(senda,1),equal(luz,0))?1
ext: click update2 Any(semaforo)?1 {luz = semaforo;}
ext: active update Any(semaforo)?1 {luz = semaforo;}
ext: inactive aux Any(semaforo)?1 {luz = semaforo;}
ext: inactive active Any(full)?1 {cantfotos = full;}
ext: active ignore And(equal(senda,1),notEqual(luz,0))?1
ext: inactive ignore2 or(any(senda),any(radar))?1
ext: click void and(equal(ready,1),equal(cantfotos,0))?1
active:infinite
speed:00:00:00:00
void:00:00:00:00
inactive:infinite
redlight:00:00:00:00
update:00:00:00:00
click:infinite
update2:00:00:00:00
aux:00:00:00:00
ignore:00:00:00:00
ignore2:00:00:00:00
luz:0
cantfotos:2

```

Modelo atómico Cámara



Funcionamiento:

La cámara está en estado StdBy hasta que recibe la orden de sacar una foto en el port Take. El tiempo de avance del rollo está representado en el time advance del estado Prepare, el cual cuando termina emite la señal en el puerto Ready.

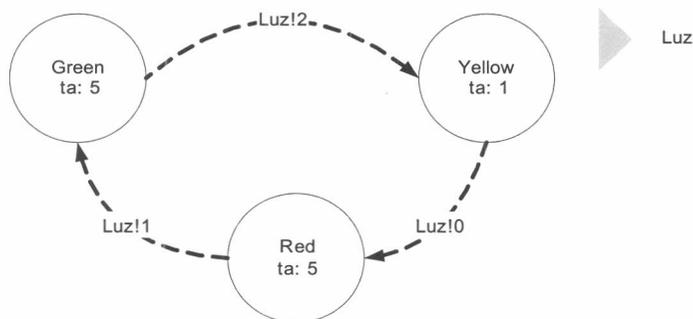
Camara.cdd

```

[camara]
in: take
out: ready
state: stdby run prepare
initial:stdby
int: run prepare
int: prepare stdby ready!1
ext: stdby run Value(take)?1
stdby:infinite
run:00:00:00:00
prepare:00:00:02:00

```

Modelo atómico Semáforo



Funcionamiento:

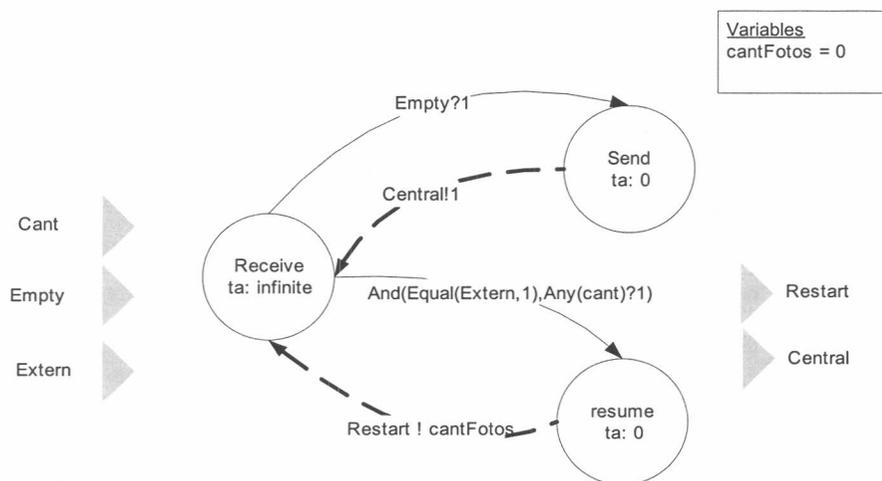
El semáforo cicla entre la luz verde, amarilla y roja a intervalos de tiempo regulares. Cada vez que cambia de luz, lo avisa a través del port Luz.

Semaforo.cdd

```

[semaforo]
out: luz
state: green yellow red
int: green yellow luz!2
int: yellow red luz!0
int: red green luz!1
green:00:00:05:00
yellow:00:00:01:00
red:00:00:05:00
    
```

Modelo atómico IO:



Funcionamiento:

Este modelo se encarga de la comunicación con la central. Si recibe una señal en el puerto Empty, la envía a través del puerto Central. Cuando se le advierte en el puerto Ext que el rollo ha sido cambiado, informa este hecho al modelo Control a través del puerto Restart. En este caso le enviará la cantidad de fotos del nuevo rollo (que debe haber recibido en el puerto Cant).

IO.cdd

```
[IO]
in: empty extern cant
out: restart central
var: cantFotos
state: receive send resume
initial:receive
int: send receive central!1
int: resume receive restart!cantFotos
ext: receive send Value(empty)?1
ext: receive resume And(extern,Any(cant))?1 {cantFotos = cant;}
receive:infinite
send:00:00:00:00
resume:00:00:00:00
cantFotos:0
```

1.1.3 Simulación

Para la ejecución de la simulación se utilizaron las siguientes entradas:

Tiempo	Puerto	Valor
00:00:02:00	senda	1
00:00:04:00	radar	1
00:00:12:00	senda	1
00:00:15:00	senda	1
00:00:19:00	senda	1
00:00:24:00	radar	1
00:00:26:00	ext	1
00:00:26:00	cant	10
00:00:29:00	senda	1
00:00:34:00	senda	1
00:00:42:00	senda	1

Para ejecutar la simulación de este modelo se ejecutó el comando:

```
cd++ -t00:00:43:00 -mradarfot.ma -eradarfot.ev -lradarfot.log -
oradarfot.out
```

Los siguientes archivos .translog muestran una traza de la ejecución en los cuatro modelos atómicos que componen este acoplado.

En esta simulación se comenzó con solo 2 fotos en el rollo, por lo que en el instante 00:00:21:000 puede observarse que el controlador saca la última foto y pasa al estado “void” emitiendo una señal por el puerto “Empty”. Este pedido es procesador por el modelo IO, quien en el instante 00:00:26:000 le avisa por el puerto “full” que se han recargado 10 fotos. En ese lapso de tiempo el controlador paso al estado “inactive” en el que solo procesa información proveniente del modelo semáforo para mantener actualizada la luz activa.

Control.translog

```
C 00:00:00:000 : active , (cantfotos=2) (luz=0)
? 00:00:02:000 : senda , 1
E 00:00:02:000 : active , redlight (cantfotos=2) (luz=0)
O 00:00:02:000 : foto , 1
I 00:00:02:000 : redlight , click (cantfotos=1) (luz=0)
? 00:00:04:000 : radar , 1
? 00:00:04:000 : ready , 1
```

```
E 00:00:04:000 : click , active (cantfotos=1) (luz=0)
? 00:00:05:000 : semaforo , 2
E 00:00:05:000 : active , update (cantfotos=1) (luz=2)
I 00:00:05:000 : update , active (cantfotos=1) (luz=2)
? 00:00:06:000 : semaforo , 0
E 00:00:06:000 : active , update (cantfotos=1) (luz=0)
I 00:00:06:000 : update , active (cantfotos=1) (luz=0)
? 00:00:11:000 : semaforo , 1
E 00:00:11:000 : active , update (cantfotos=1) (luz=1)
I 00:00:11:000 : update , active (cantfotos=1) (luz=1)
? 00:00:12:000 : senda , 1
E 00:00:12:000 : active , ignore (cantfotos=1) (luz=1)
I 00:00:12:000 : ignore , active (cantfotos=1) (luz=1)
? 00:00:15:000 : senda , 1
E 00:00:15:000 : active , ignore (cantfotos=1) (luz=1)
I 00:00:15:000 : ignore , active (cantfotos=1) (luz=1)
? 00:00:16:000 : semaforo , 2
E 00:00:16:000 : active , update (cantfotos=1) (luz=2)
I 00:00:16:000 : update , active (cantfotos=1) (luz=2)
? 00:00:17:000 : semaforo , 0
E 00:00:17:000 : active , update (cantfotos=1) (luz=0)
I 00:00:17:000 : update , active (cantfotos=1) (luz=0)
? 00:00:19:000 : senda , 1
E 00:00:19:000 : active , redlight (cantfotos=1) (luz=0)
O 00:00:19:000 : foto , 1
I 00:00:19:000 : redlight , click (cantfotos=0) (luz=0)
? 00:00:21:000 : ready , 1
E 00:00:21:000 : click , void (cantfotos=0) (luz=0)
O 00:00:21:000 : empty , 1
I 00:00:21:000 : void , inactive (cantfotos=0) (luz=0)
? 00:00:22:000 : semaforo , 1
E 00:00:22:000 : inactive , aux (cantfotos=0) (luz=1)
I 00:00:22:000 : aux , inactive (cantfotos=0) (luz=1)
? 00:00:24:000 : radar , 1
E 00:00:24:000 : inactive , ignore2 (cantfotos=0) (luz=1)
I 00:00:24:000 : ignore2 , inactive (cantfotos=0) (luz=1)
? 00:00:26:000 : full , 10
E 00:00:26:000 : inactive , active (cantfotos=10) (luz=1)
? 00:00:27:000 : semaforo , 2
E 00:00:27:000 : active , update (cantfotos=10) (luz=2)
I 00:00:27:000 : update , active (cantfotos=10) (luz=2)
? 00:00:28:000 : semaforo , 0
E 00:00:28:000 : active , update (cantfotos=10) (luz=0)
I 00:00:28:000 : update , active (cantfotos=10) (luz=0)
? 00:00:29:000 : senda , 1
E 00:00:29:000 : active , redlight (cantfotos=10) (luz=0)
O 00:00:29:000 : foto , 1
I 00:00:29:000 : redlight , click (cantfotos=9) (luz=0)
? 00:00:31:000 : ready , 1
E 00:00:31:000 : click , active (cantfotos=9) (luz=0)
? 00:00:33:000 : semaforo , 1
E 00:00:33:000 : active , update (cantfotos=9) (luz=1)
I 00:00:33:000 : update , active (cantfotos=9) (luz=1)
? 00:00:34:000 : senda , 1
E 00:00:34:000 : active , ignore (cantfotos=9) (luz=1)
I 00:00:34:000 : ignore , active (cantfotos=9) (luz=1)
? 00:00:38:000 : semaforo , 2
E 00:00:38:000 : active , update (cantfotos=9) (luz=2)
I 00:00:38:000 : update , active (cantfotos=9) (luz=2)
? 00:00:39:000 : semaforo , 0
```

```
E 00:00:39:000 : active , update (cantfotos=9) (luz=0)
I 00:00:39:000 : update , active (cantfotos=9) (luz=0)
? 00:00:42:000 : senda , 1
E 00:00:42:000 : active , redlight (cantfotos=9) (luz=0)
O 00:00:42:000 : foto , 1
I 00:00:42:000 : redlight , click (cantfotos=8) (luz=0)
```

Camara.translog

```
C 00:00:00:000 : stdby ,
? 00:00:02:000 : take , 1
E 00:00:02:000 : stdby , run
I 00:00:02:000 : run , prepare
O 00:00:04:000 : ready , 1
I 00:00:04:000 : prepare , stdby
? 00:00:19:000 : take , 1
E 00:00:19:000 : stdby , run
I 00:00:19:000 : run , prepare
O 00:00:21:000 : ready , 1
I 00:00:21:000 : prepare , stdby
? 00:00:29:000 : take , 1
E 00:00:29:000 : stdby , run
I 00:00:29:000 : run , prepare
O 00:00:31:000 : ready , 1
I 00:00:31:000 : prepare , stdby
? 00:00:42:000 : take , 1
E 00:00:42:000 : stdby , run
I 00:00:42:000 : run , prepare
```

IO.translog

```
C 00:00:00:000 : receive , (cantfotos=0)
? 00:00:21:000 : empty , 1
E 00:00:21:000 : receive , send (cantfotos=0)
O 00:00:21:000 : central , 1
I 00:00:21:000 : send , receive (cantfotos=0)
? 00:00:26:000 : cant , 10
? 00:00:26:000 : extern , 1
E 00:00:26:000 : receive , resume (cantfotos=10)
O 00:00:26:000 : restart , 10
I 00:00:26:000 : resume , receive (cantfotos=10)
```

Semaforo.translog

```
C 00:00:00:000 : green ,
O 00:00:05:000 : luz , 2
I 00:00:05:000 : green , yellow
O 00:00:06:000 : luz , 0
I 00:00:06:000 : yellow , red
O 00:00:11:000 : luz , 1
I 00:00:11:000 : red , green
O 00:00:16:000 : luz , 2
I 00:00:16:000 : green , yellow
O 00:00:17:000 : luz , 0
I 00:00:17:000 : yellow , red
O 00:00:22:000 : luz , 1
I 00:00:22:000 : red , green
O 00:00:27:000 : luz , 2
I 00:00:27:000 : green , yellow
```

```
O 00:00:28:000 : luz , 0
I 00:00:28:000 : yellow , red
O 00:00:33:000 : luz , 1
I 00:00:33:000 : red , green
O 00:00:38:000 : luz , 2
I 00:00:38:000 : green , yellow
O 00:00:39:000 : luz , 0
I 00:00:39:000 : yellow , red
```

CGGAD Graphic Tool

Manual del Usuario

1 Introducción

Coupled GGAD Graphic tool es una herramienta de diseño de modelos DEVS acoplados. Entre sus funciones podemos destacar:

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

2 Definición

La herramienta de simulación CD++ realiza simulaciones de modelos acoplados DEVS. Para la definición del modelo acoplado a simular usa un lenguaje que describe la interconexión entre los modelos atómicos y/o acoplados que lo componen. Los modelos atómicos pueden ser “clásicos”, es decir que el simulador CD++ ya conoce su funcionamiento, o “GGAD”, los cuales se definen a través de un archivo de texto que describe su estructura y funcionamiento y cuyo formato se denomina GADscript.

La herramienta Coupled GGAD Graphic tool está concebida en 2 partes. En una permite la creación y edición de un modelo DEVS acoplado y en la otra de modelos DEVS atómicos “GGAD”. El modelo acoplado inicial o raíz, puede incluir otros modelos acoplados, modelos atómicos GGAD o modelos atómicos clásicos.

Un modelo acoplado tiene un nombre y un archivo de texto que describe su composición (normalmente con extensión .ma). Los modelos atómicos GGAD también poseen un nombre de modelo y un archivo que los describe, a diferencia de los modelos atómicos clásicos que simplemente son incluidos con un nombre y una lista de parámetros y sus valores asociados, dado que el simulador CD++ ya conoce su estructura y funcionamiento.

Para la simulación de un modelo acoplado creado con esta herramienta, se debe exportar el mismo en formato GADscript. Esto creará un archivo con extensión “.ma” que tendrá una sección para cada modelo integrante del acoplado raíz. En las secciones correspondientes a modelos atómicos clásicos solo habrá parámetros, en las secciones correspondientes a submodelos acoplados estará la descripción de su estructura, y en las secciones correspondientes a modelos atómicos GGAD estará el nombre del archivo que contiene su estructura y funcionamiento más los parámetros que el modelo acepte.

3 Edición de modelos acoplados con GGADTool

3.1 Inicio

Una vez iniciada la aplicación aparecerá tal como en la figura 1.

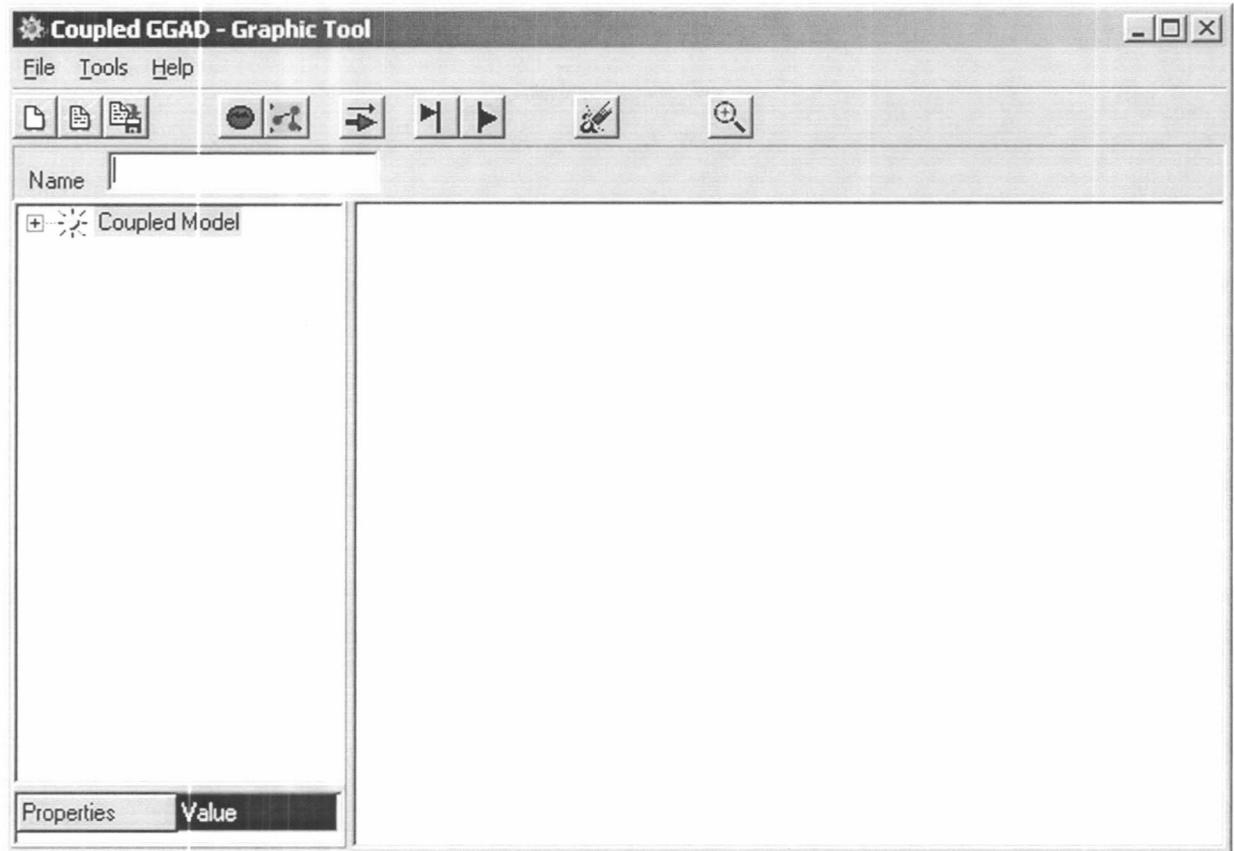


Figura 1

Aquí podemos ver la división de la ventana en 3 grupos de información:

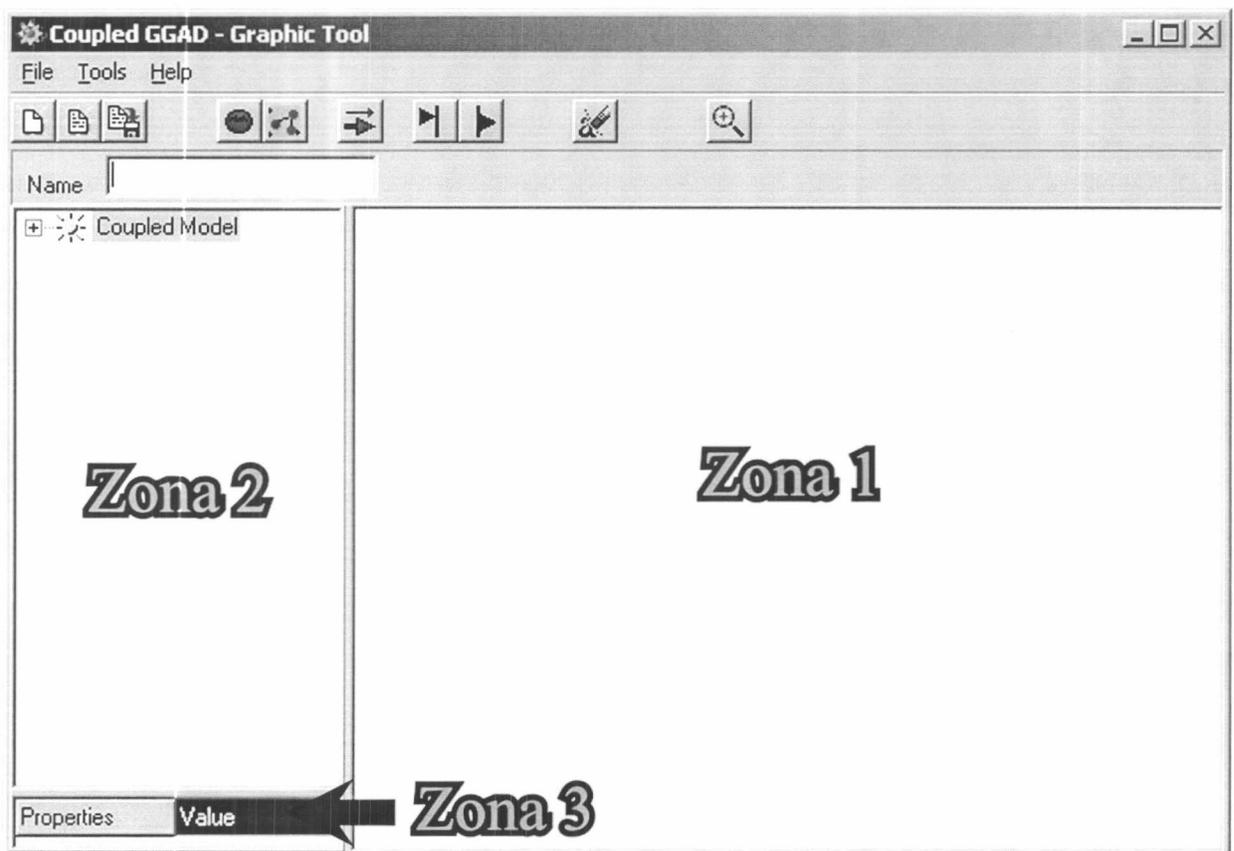


Figura 2

- ξ la zona de dibujo (1)
- ξ la zona de definición jerárquica del modelo acoplado que estamos diseñando (2)
- ξ la zona de propiedades y valores del objeto seleccionado (3)

Para comenzar a diseñar un modelo DEVS acoplado basta con escribir un nombre para el objeto a insertar, y presionar el botón correspondiente en la barra de herramientas.

Se pueden agregar modelos acoplados, modelos atómicos, ports de entrada o salida del modelo acoplado raíz, ports de entrada o salida de cada modelo atómico (si es que son modelos atómicos clásicos) y conexiones entre ports.

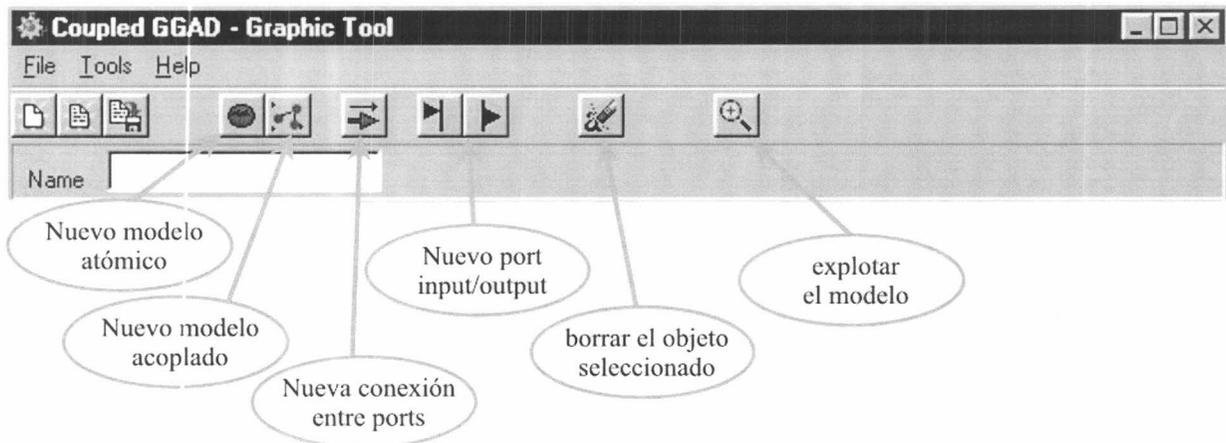


Figura 3

3.2 Agregar modelos atómicos

Para agregar modelos atómicos primero se escribe el nombre en el casillero "Name" y luego se presiona el botón correspondiente. Una vez agregado se puede cambiar el valor de sus atributos en la zona de propiedades de los objetos. Para ello, seleccione el modelo atómico recién creado con un click del mouse sobre su imagen, o en su nombre en la zona de definición jerárquica.

Atributos de un modelo atómico:

- ξ **Name** : no modificable
- ξ **Definition file**: nombre de archivo que define este modelo atómico (*.ggd). Se puede asignar en el momento de creación para que tome su estructura de un archivo ya existente, o se puede explotar el modelo atómico con el correspondiente botón de la barra de herramientas y en la nueva ventana diseñarlo gráficamente para después almacenarlo en un archivo.
- ξ **Params**: lista de parámetros para este modelo atómico. Se debe usar cuando en el modelo acoplado se incluyen modelos atómicos clásicos (predefinidos en el CD++), que por estar ya compilados, necesitan parámetros para ser instanciados. También se pueden definir parámetros para un modelo atómico GGAD.
- ξ **Class**: Nombre de la clase del modelo. En el caso de que sean modelos GGad su clase será Ggad. Cuando se agregan modelos clásicos, se debe especificar su clase en este casillero.

Properties	Value
Name	Modelo1
Definition file	modelo1.ggd
Params	⇒ ...
Class	GGad

Figura 4

3.3 Agregar ports

Para agregar ports, se escribe el nombre en el casillero "Name" y luego se presiona el botón correspondiente en la barra de herramientas, según sea de entrada o salida.

Los ports pueden ser creados en dos niveles:

- ξ Como ports del modelo raíz
- ξ Como ports de un modelo atómico clásico para poder definir las conexiones que el mismo tiene con el resto de los modelos

Si el objeto seleccionado es un modelo atómico, el port será creado para este modelo, en otro caso será un port del modelo acoplado raíz. La funcionalidad de crear ports para modelos atómicos está prevista para los modelos atómicos clásicos, si el modelo atómico seleccionado es GGAD, la forma correcta de agregarle un port es explotar el modelo y editarlo en la aplicación GGAD Graphic tool. Cabe destacar que al editar un modelo atómico con GGAD Graphic tool, se perderán todos los atributos y ports creados en el nivel del modelo acoplado.

3.4 Agregar conexiones entre ports

Las conexiones entre ports tienen como punto inicial un port de entrada del modelo raíz o un port de salida de alguno de los modelos interiores, ya sea atómico o acoplado, y pueden terminar en un port de salida del modelo raíz o un port de entrada de otro de los modelos interiores.

Para realizar una conexión, siga estos pasos:

- 1- Seleccione en la zona de definición jerárquica el port de inicio de la conexión
- 2- Presione el botón correspondiente en la barra de herramientas
- 3- Del cuadro de diálogo abierto, elija un port de finalización de la conexión y presione Ok

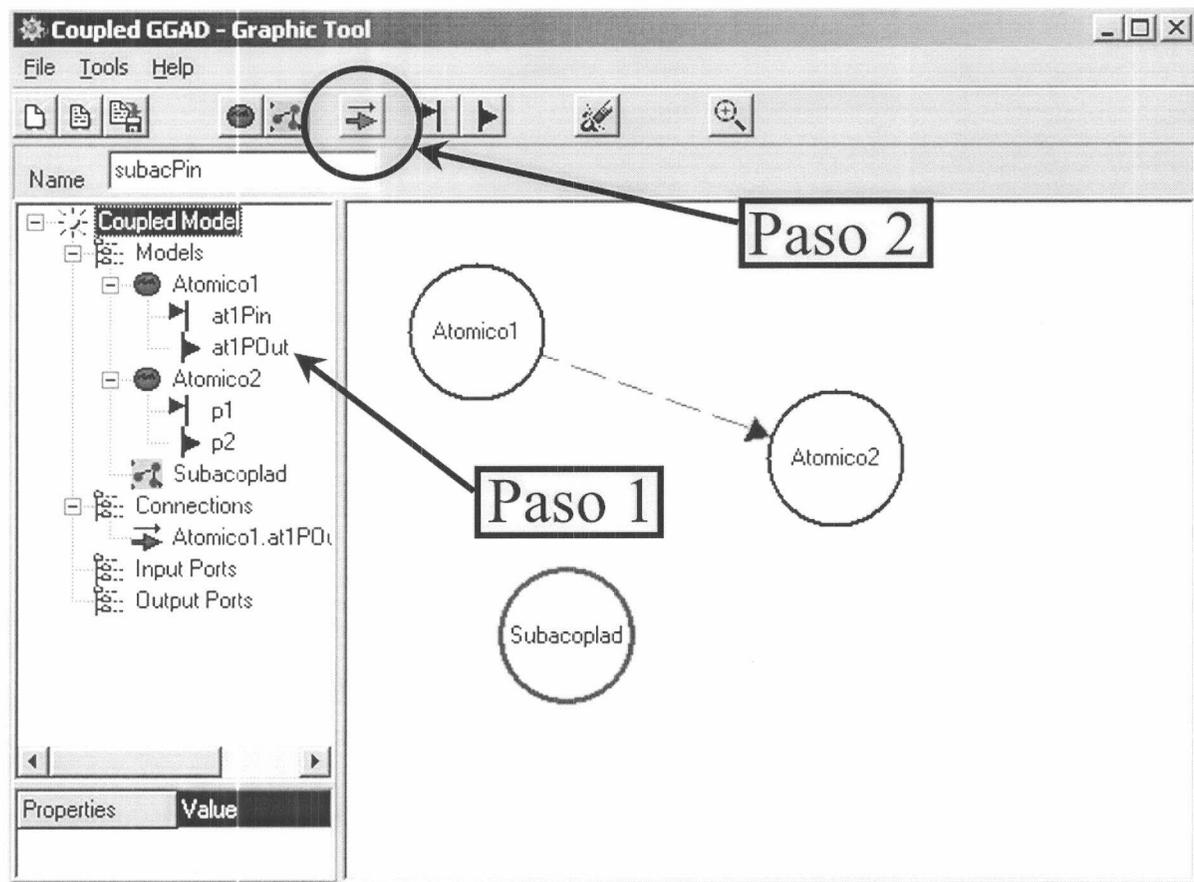


Figura 5

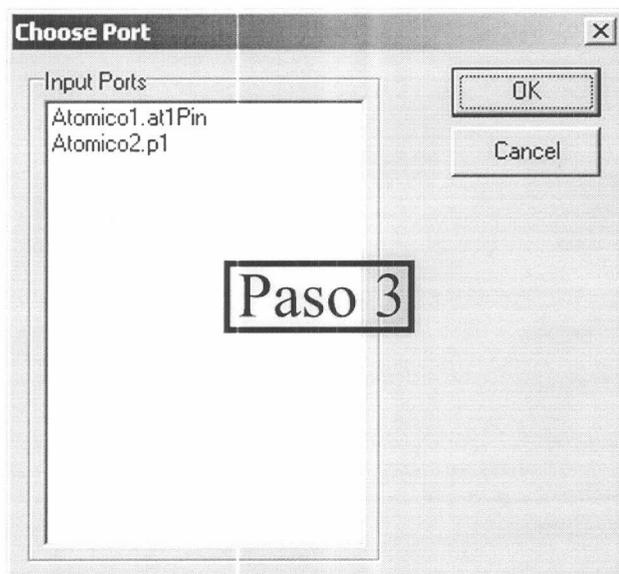


Figura 6

Nota: Una manera alternativa de realizar conexiones, es arrastrando y soltando el port inicial sobre el port final en la zona de definición jerárquica del modelo acoplado.

3.5 Borrar objetos

Para borrar un objeto, selecciónelo y luego presione el botón correspondiente en la barra de herramientas. Los modelos atómicos y las conexiones entre ports pueden ser seleccionados indistintamente en la zona gráfica

de dibujo como en la zona de la definición jerárquica del modelo, en tanto que los ports solo pueden ser seleccionados en la zona de definición jerárquica, ya que no tienen representación gráfica.

Tenga en cuenta que al borrar un objeto se borran automáticamente todos los objetos dependientes, es decir que si se borra un modelo atómico desaparecerán los ports del mismo y las conexiones en las que participe alguno de sus ports.

3.6 Explotar un modelo

Al seleccionar un modelo y presionar el botón correspondiente en la barra de herramientas:

ξ Si el objeto seleccionado es un modelo acoplado sin Definition file

Se abre otra ventana con la aplicación CGGAD Graphic tool con un modelo acoplado en blanco. No se podrá volver al modelo raíz hasta no finalizar esta aplicación. Si el modelo editado se guarda en un archivo (con extensión .cgd) al cerrar esta nueva ventana, los cambios se verán reflejados en el atributo "Definition file" del modelo en cuestión. Si el modelo tenía definidos ports antes de ser explotado, estos serán reemplazados por los definidos mientras fue editado.

ξ Si el objeto seleccionado es un modelo atómico de clase Ggad sin Definition file especificado:

Se abre otra ventana con la aplicación GGAD Graphic tool con un modelo vacío para editar. No se podrá volver al modelo raíz hasta no finalizar esta aplicación. Si el modelo editado se guarda en un archivo (con extensión .ggd) al cerrar esta nueva ventana, los cambios se verán reflejados en el atributo "Definition file" del modelo en cuestión. Si el modelo tenía definidos ports antes de ser explotado, estos serán reemplazados por los definidos mientras fue editado.

ξ Si el objeto seleccionado es un modelo con Definition file especificado:

Se abre otra ventana con la aplicación CGGAD ó GGAD Graphic tool, según si el modelo explotado es atómico o acoplado, con el modelo descrito en el archivo en cuestión. No se podrá volver al modelo raíz hasta no finalizar esta aplicación.

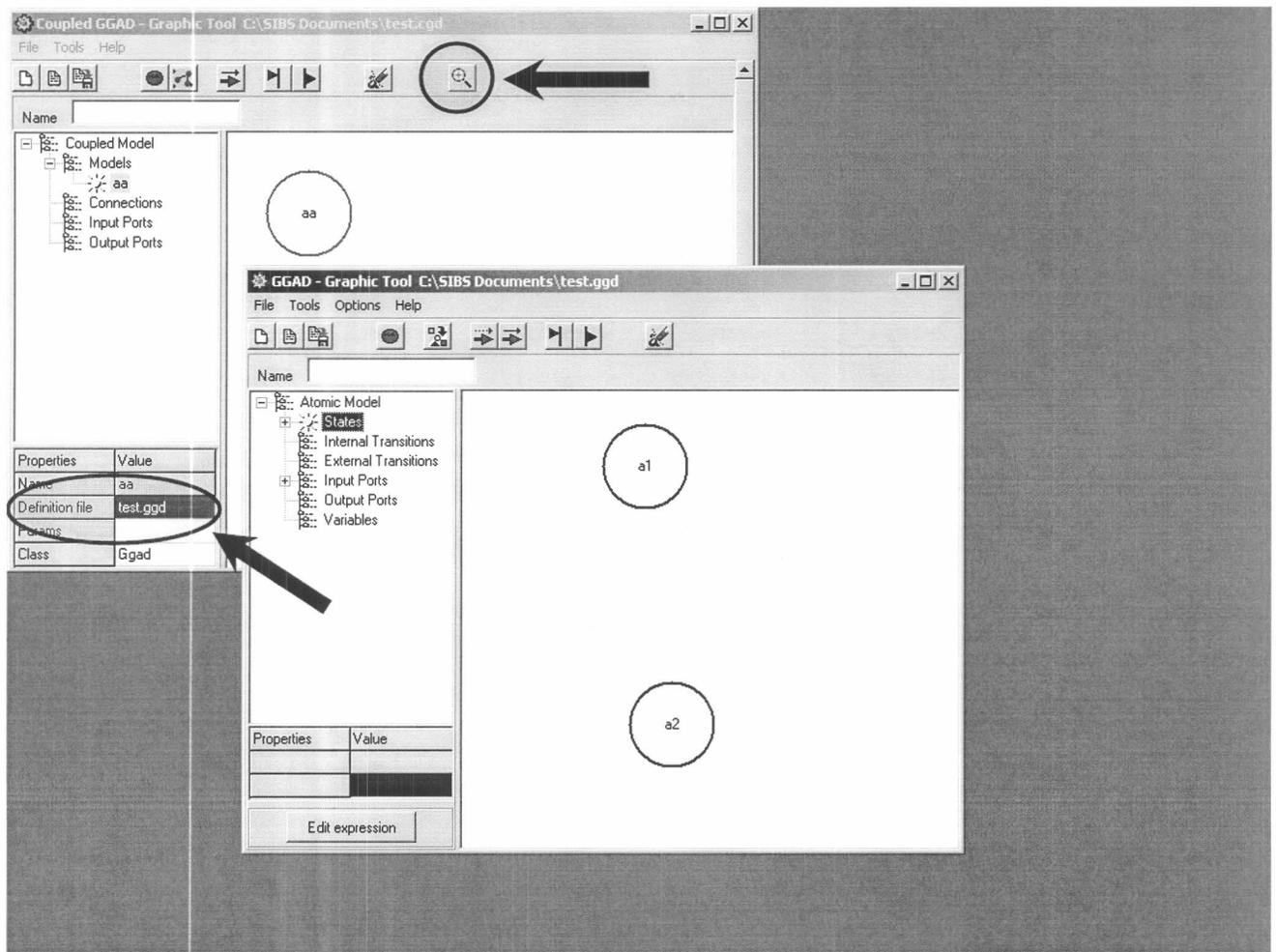


Figura 7

3.7 Guardar el modelo

Una vez terminado el modelo acoplado, el mismo puede ser guardado en un archivo “.cgd” para su posterior edición con el comando del menú File “Save”, o puede ser exportado para su uso por el simulador CD++ con extensión “.ma” con el comando del menú File “Export to CD++”. El formato de exportación (GADscript) solo almacena la información relevante para el simulador CD++, por lo que siempre se debe guardar el modelo acoplado en formato “.cgd” para su posterior edición.

Nota: Antes de poder exportar un modelo acoplado, todos sus componentes deben haber sido guardados en formato “.cgd” o “.ggd”

4 Edición de modelos atómicos GGAD

4.1 Inicio

Una vez iniciada la aplicación (por haber explotado un modelo atómico en la aplicación CGGAD), aparecerá tal como en la figura

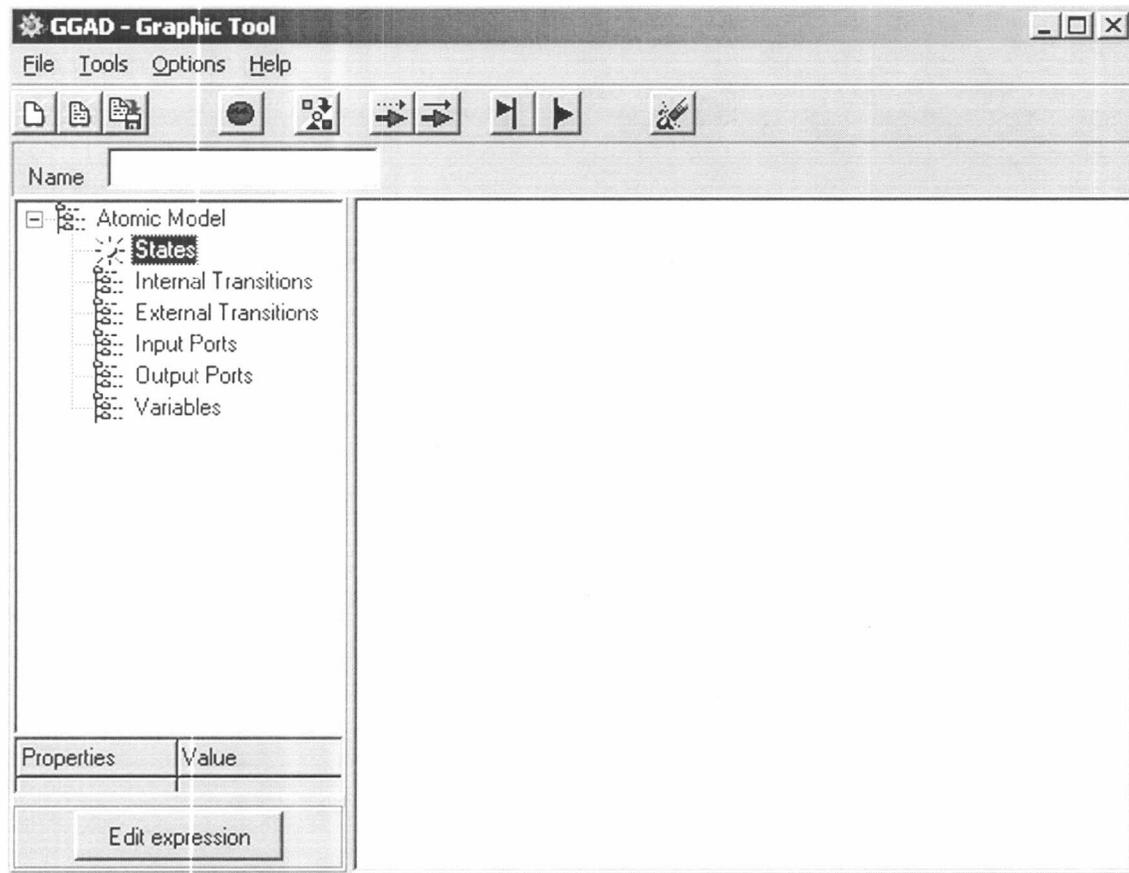


Figura 8

Aquí podemos ver la división de la ventana en 3 grupos de información:

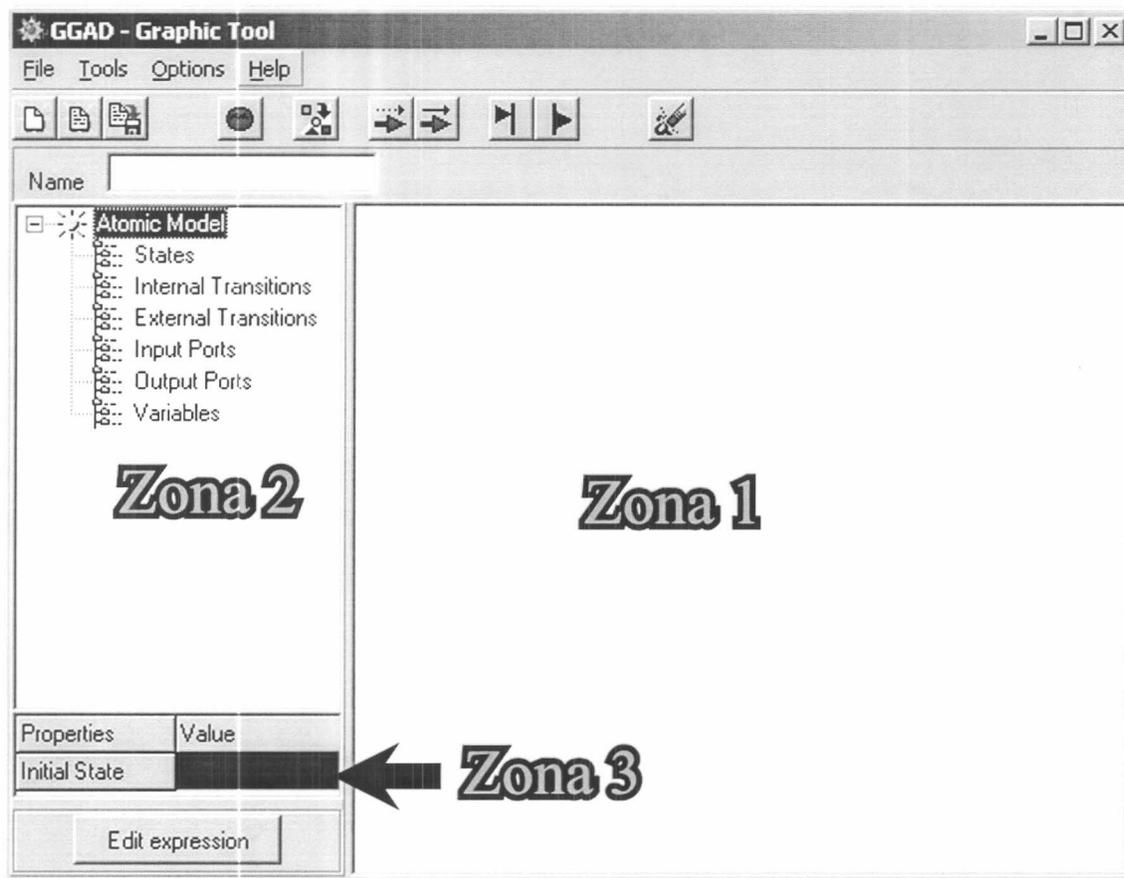


Figura 9

- ξ la zona de dibujo (1)
- ξ la zona de definición jerárquica del modelo atómico que estamos diseñando (2)
- ξ la zona de propiedades y valores del objeto seleccionado (3)

Para comenzar a diseñar un modelo DEVS atómico basta con escribir un nombre para el objeto a insertar, y presionar el botón correspondiente en la barra de herramientas.

Se pueden agregar estados, transiciones internas o externas, ports de entrada o salida, variables del modelo, o acciones a las transiciones.

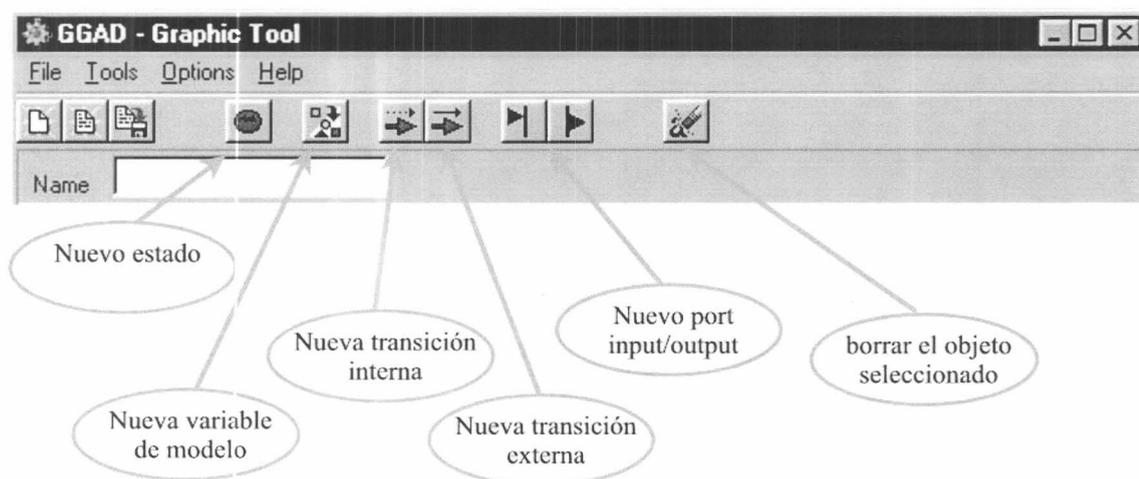


Figura 10

4.2 Agregar estados

Para agregar estados primero se escribe el nombre en el casillero "Name" y luego se presiona el botón correspondiente. Una vez agregado se puede cambiar el valor de sus atributos en la zona de propiedades de los objetos.

Atributos de un estado:

- ξ **Name** : no modificable
- ξ **Time advance**: tiempo que el simulador permanecerá en este estado antes de disparar una transición interna. Su valor puede ser expresado en unidades de tiempo con el formato 00:00:00:00 (horas:minutos:segundos:centésimas de segundo) o como "infinite" (sin las comillas) si se desea que el time advance sea infinito.

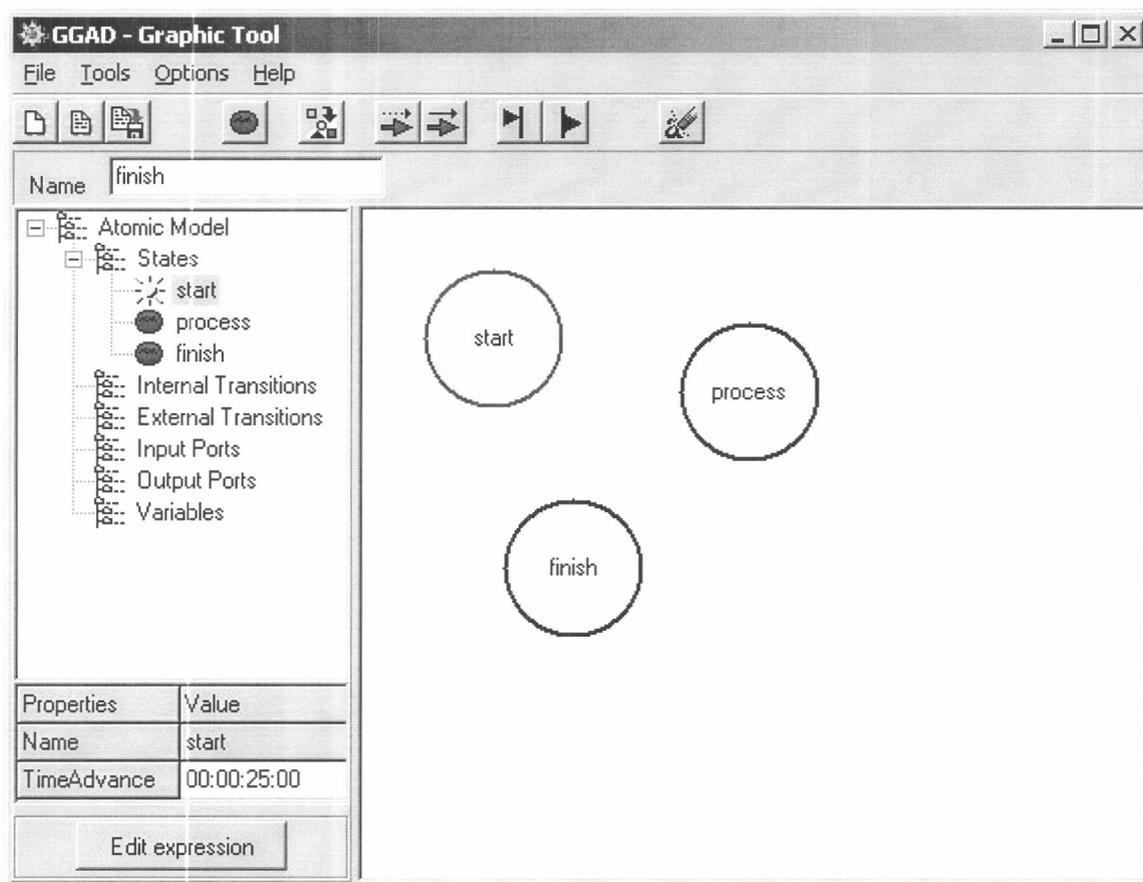


Figura 11

4.3 Agregar ports

Para agregar ports, se escribe el nombre en el casillero "Name" y luego se presiona el botón correspondiente en la barra de herramientas, según sea de entrada o salida.

4.4 Agregar transiciones internas

Las transiciones internas conectan 2 estados y pueden tener una función de salida asociada. La función de salida consiste en 0 o más pares de port/valor, es decir que ante la ocurrencia de la transición interna se puede especificar distintos valores que serán enviados por los ports de salida elegidos.

Para crear una transición interna siga estos pasos:

- ξ Seleccione en la zona gráfica el estado inicial haciendo un click sobre él.
- ξ Presionando la tecla ctrl haga un click sobre el estado final de la transición también en la zona gráfica
- ξ Si la transición a crear tiene una función de salida asociada, seleccione el port de salida correspondiente en la zona de definición jerárquica del modelo
- ξ Presione el botón correspondiente de la barra de tareas

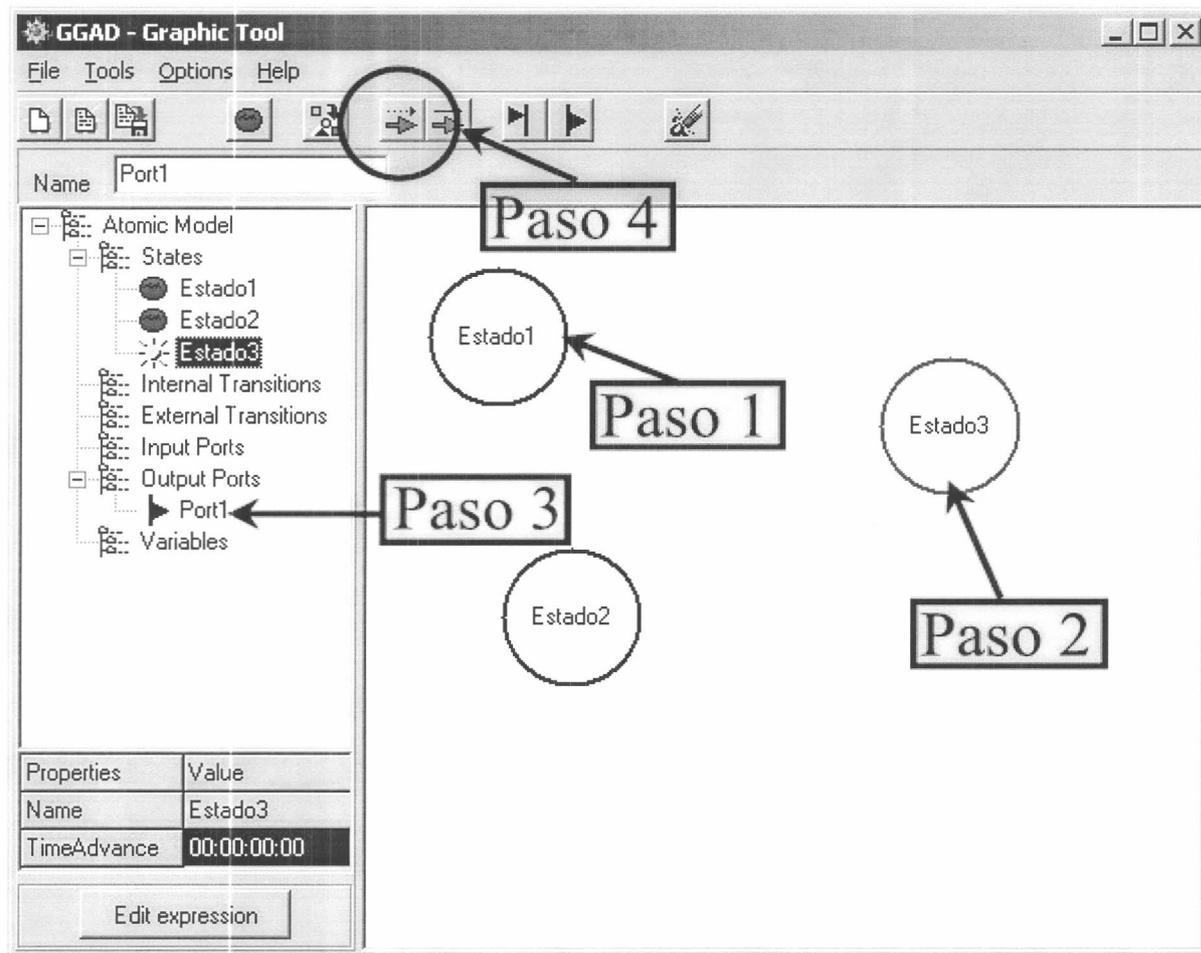


Figura 12

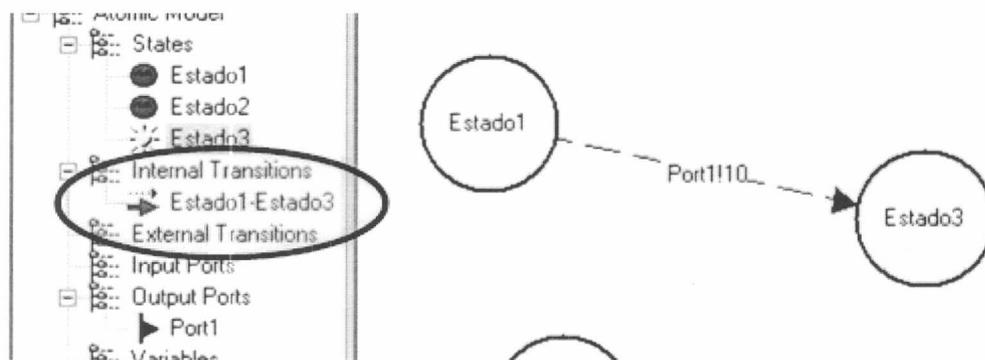


Figura 13

Si desea agregar más valores de salida en otros ports:

- ξ Seleccione la transición interna en la zona gráfica o en la zona de definición jerárquica del modelo.
- ξ En la zona de propiedades y valores del objeto seleccionado, presione el botón correspondiente a la función de salida (Output function).

- ξ En el cuadro de diálogo rellene la tabla con los pares de ports y valores que serán emitidos a través de ellos. Tenga en cuenta que solo serán agregados los pares de port/valor cuyos nombres de port existan como ports de salida del modelo. El valor a ser ingresado puede ser una expresión compuesta por llamadas a funciones Built-in cuyos parámetros sean constantes o variables del modelo.

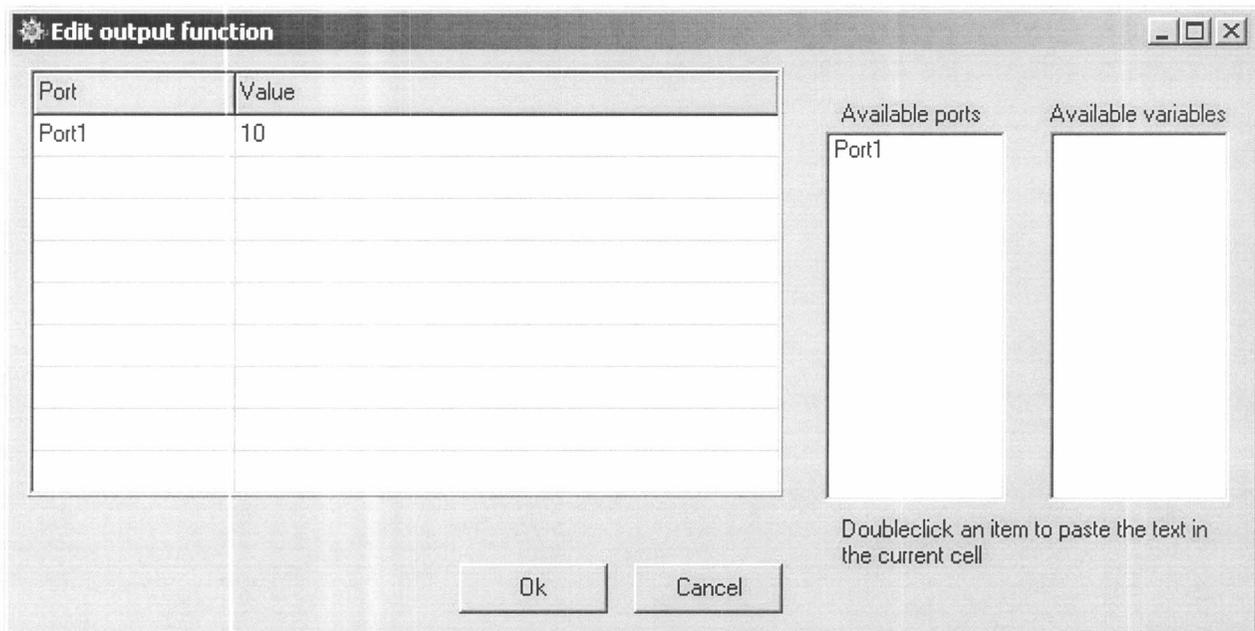


Figura 14

4.5 Agregar transiciones externas

Las transiciones externas conectan 2 estados a partir de un evento externo. Un evento externo se define como la aparición de un valor en al menos un port de entrada del modelo.

Cada transición externa posee una condición que debe cumplirse para ser activada. Esta condición se define por medio de una función cuyos parámetros pueden ser valores recibidos en los ports de entrada, variables del modelo o constantes numéricas. Las funciones a aplicar deben estar soportadas por el simulador CD++. En su versión actual soporta las siguientes funciones:

- ξ Add(n1,n2): devuelve la suma de n1 y n2
- ξ And(n1,n2): retorna verdadero si ambos parámetros son verdaderos (1)
- ξ Any(port1): retorna verdadero si el port port1 recibió un valor en este evento
- ξ Between(n1,n2,n3): retorna verdadero si n2 es mayor que n1 y menor que n3
- ξ Compare(n1,n2,n3,n4,n5): retorna n3, n4 o n5 si n1 es mayor, igual o menor que n2 respectivamente
- ξ Divide(n1,n2): retorna n1 dividido n2
- ξ Equal(n1,n2): retorna verdadero si n1 es igual a n2
- ξ Minus(n1,n2): devuelve n1-2
- ξ Multiply(n1,n2): retorna n1 multiplicado por n2
- ξ NotEqual(n1,n2): retorna verdadero si n1 es distinto de n2
- ξ Pow(n1,n2): retorna n1 a la potencia n2

ξ Value(port1): retorna el valor que recibió el port1 en este evento

Para crear la transición externa siga estos pasos:

- ξ Seleccione en la zona gráfica el estado inicial haciendo un click sobre él.
- ξ Presionando la tecla ctrl haga un click sobre el estado final de la transición, también en la zona gráfica
- ξ Seleccione el primer port de entrada (si es en función a uno o más ports)
- ξ Presione el botón correspondiente de la barra de tareas

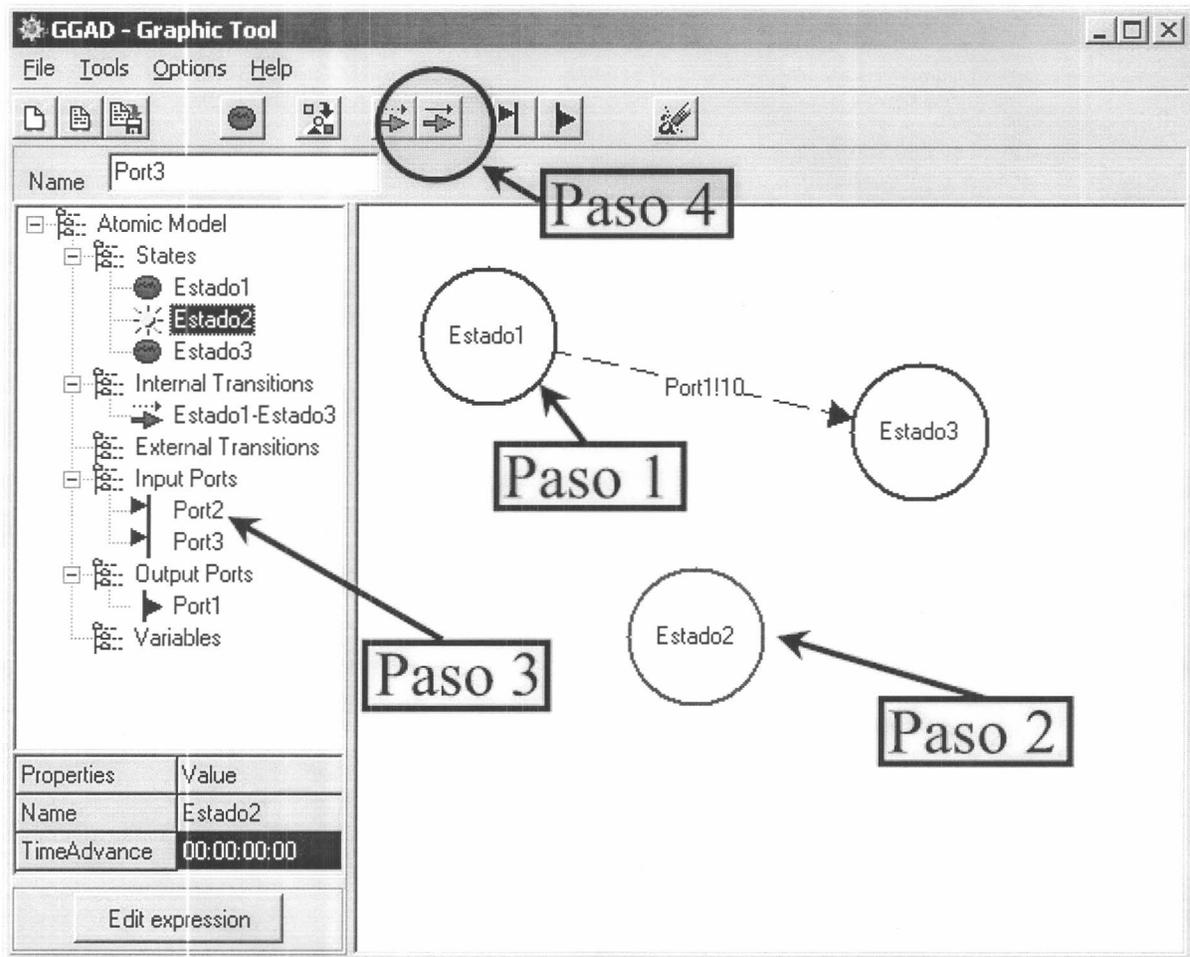


Figura 15

ξ Para modificar la condición que dispara esta transición, seleccione la transición en el diagrama jerárquico y edite la expresión. Recuerde que la expresión es una función con los parámetros correspondientes, que pueden ser constantes, variables del modelo, ports de entrada u otras funciones anidadas.

ξ Por ejemplo la expresión:

```
And ( Value ( Port1 ) ? 2 , Equal ( var1 , 5 ) )
```

Significa que la condición se cumple si en el port "Port1" hay un valor 2 y la variable "var1" es igual a 5.

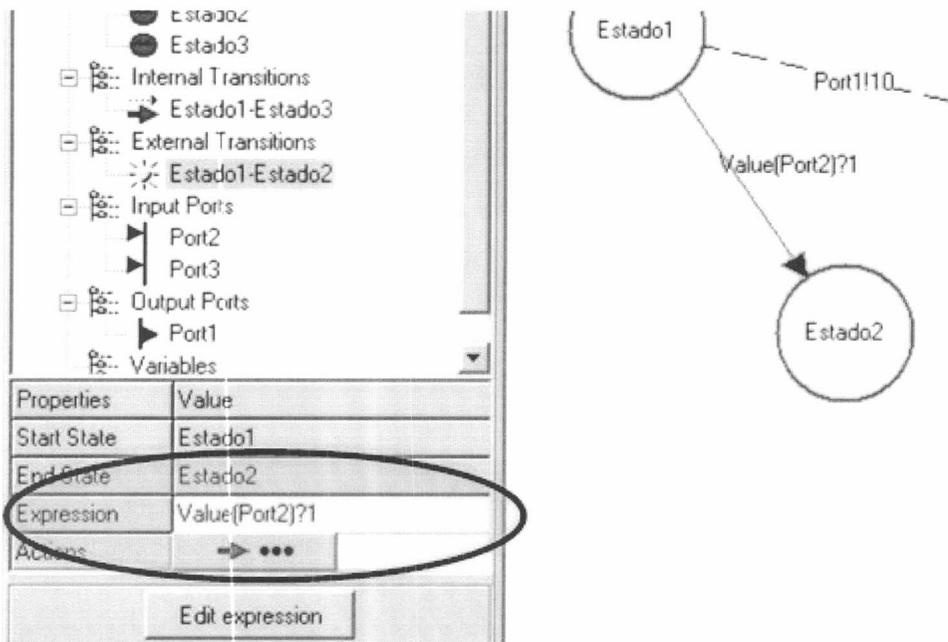


Figura 16

- ξ Una manera alternativa de modificar la condición de una transición externa es presionando el botón “Edit Expression”. De esta manera aparecerá un cuadro de diálogo como el de la figura 17 que permite editar más cómodamente la expresión. Haciendo doble click en la lista de funciones disponibles se agregará el nombre de la función Builtin a la expresión.

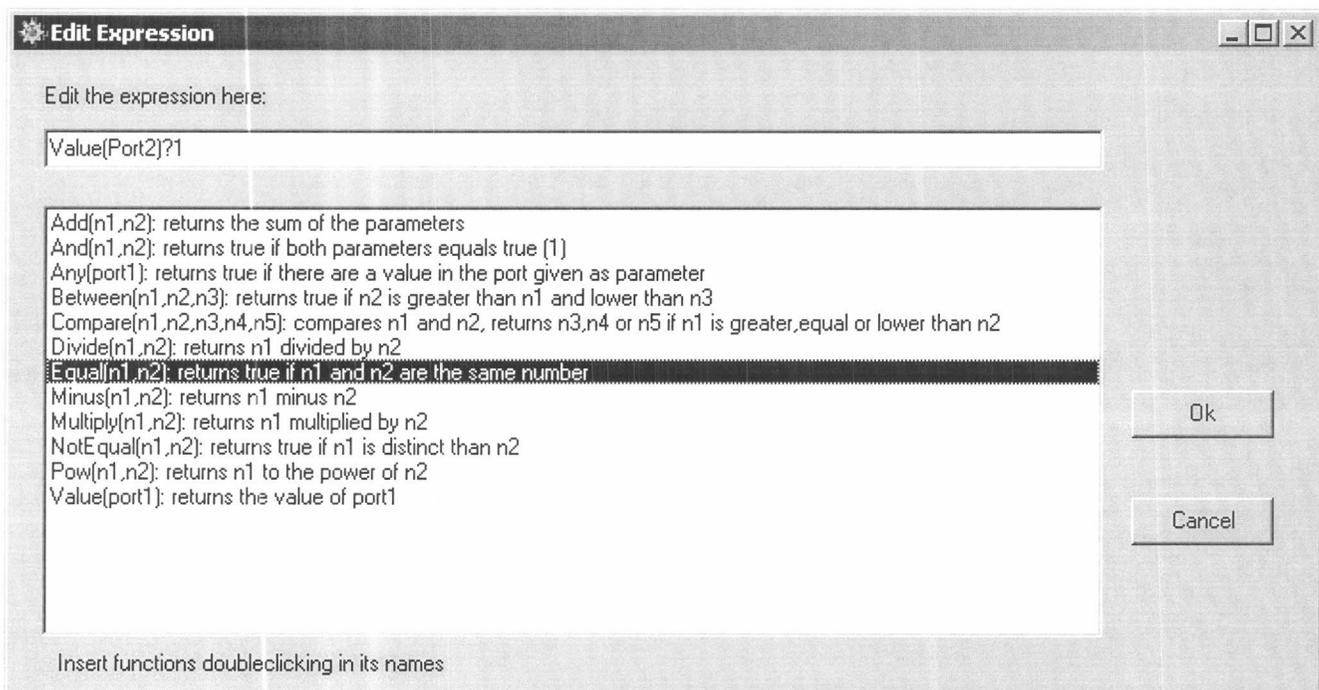


Figura 17

4.6 Agregar acciones a las transiciones

Tanto las transiciones internas como las externas, tienen la capacidad de ejecutar una lista de acciones en el momento de su activación, es decir cuando la condición para su ejecución ha sido satisfecha. Las acciones permiten asignar valores a las variables del modelo en tiempo de ejecución. El valor asignado proviene de la ejecución de una función Built-in sobre parámetros que pueden ser ports de entrada, variables del modelo y constantes.

Para agregar acciones a una transición, selecciónela y presione el botón “Actions” en la zona de propiedades del objeto seleccionado.

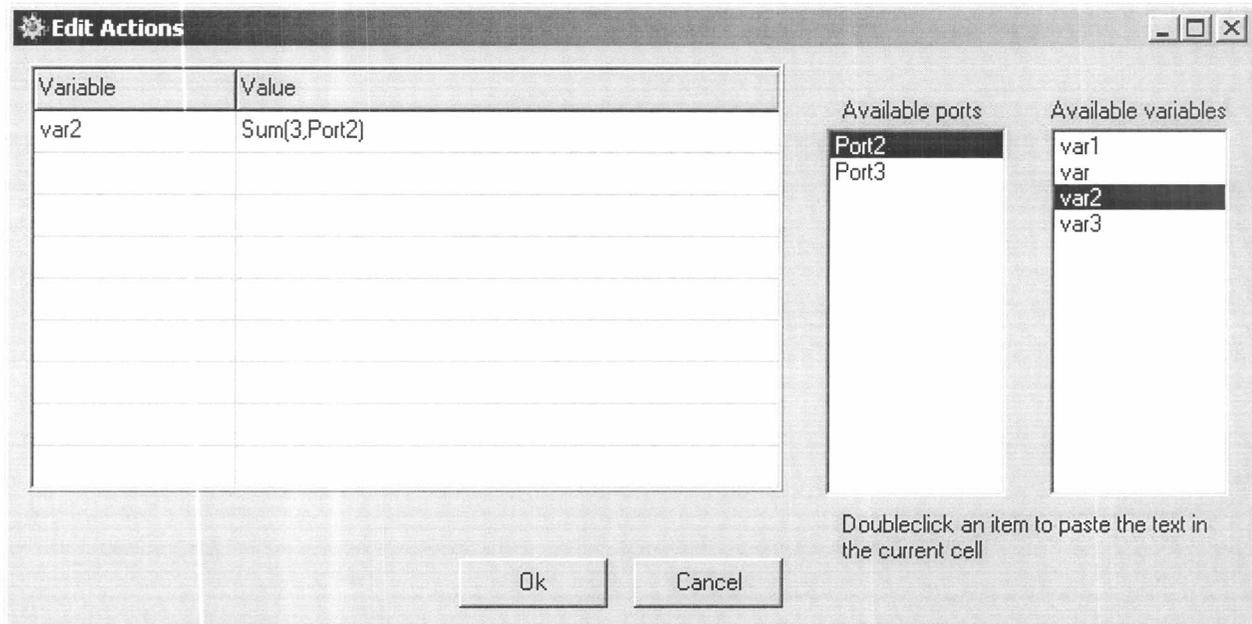


Figura 18

4.7 Borrar objetos

Para borrar un objeto, selecciónelo en la representación gráfica del modelo y luego presione el botón correspondiente en la barra de herramientas. Tenga en cuenta que al borrar un objeto se borran automáticamente todos los objetos dependientes, es decir que si se borra un estado desaparecerán las transiciones internas y externas en las que participaba.

4.8 Guardar el modelo

Una vez realizado el modelo atómico, el mismo puede ser guardado en un archivo “.ggd” para su posterior edición con el comando del menú File “Save”. Para realizar una simulación únicamente con este modelo atómico, genere un modelo acoplado que contenga únicamente a este modelo atómico y exporte el modelo acoplado. El formato de exportación solo almacena la información relevante para el simulador CD++, por lo que siempre se debe guardar el modelo atómico en formato CGGAD Graphic Tool para su posterior edición.

4.9 Otras funciones

En el menú “Options” aparecen dos opciones:

- ξ Show Transitions Names: Si está activada, en la zona del gráfico las transiciones tendrán sobreimpreso un texto con la función de salida o la condición para ejecutarse (Según sean internas o externas)
- ξ Show Transitions Tooltips: al activarla, cada vez que el mouse pase por una transición en la zona gráfica, aparecerá un “tooltip” con los atributos de la transición (incluyendo las acciones)

En el menú Tools, aparece un comando “Builtin functions”. A través de este comando se puede editar las funciones builtin disponibles para las expresiones en GGADTool. Para que tenga sentido deben estar implementadas en el simulador CD++.

5 TABLA DE CONTENIDOS

CGGAD GRAPHIC TOOL MANUAL DEL USUARIO.....	1
1 INTRODUCCIÓN	1
2 DEFINICIÓN	1
3 EDICIÓN DE MODELOS ACOPLADOS CON GGADTOOL.....	1
3.1 Inicio.....	1
3.2 Agregar modelos atómicos	3
3.3 Agregar ports.....	4
3.4 Agregar conexiones entre ports.....	4
3.5 Borrar objetos	5
3.6 Explotar un modelo	6
3.7 Guardar el modelo.....	7
4 EDICIÓN DE MODELOS ATÓMICOS GGAD.....	7
4.1 Inicio.....	7
4.2 Agregar estados.....	10
4.3 Agregar ports.....	11
4.4 Agregar transiciones internas	11
4.5 Agregar transiciones externas.....	13
4.6 Agregar acciones a las transiciones.....	15
4.7 Borrar objetos	16
4.8 Guardar el modelo.....	16
4.9 Otras funciones.....	16
5 TABLA DE CONTENIDOS	17