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

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.

1 Introducción

Durante las últimas décadas, la rápida evolución de la tecnología ha producido una proliferación de nuevos sistemas dinámicos, generalmente hechos por el hombre y de gran complejidad. Ejemplos de ellos son las redes de computadoras, sistemas de producción automatizados, de control de tráfico aéreo; y sistemas en general de comando, de control, de comunicaciones y de información. 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 lleva a definir el término de Sistemas de Eventos Discretos.
Las herramientas matemáticas que hoy disponemos (básicamente ecuaciones diferenciales y en diferencias) 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. Nos ocuparemos, sin embargo, exclusivamente de modelización y simulación, dejando de lado estas herramientas de análisis.
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 mas 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 jerarquias de modelos.

2.1 DEVS clásico

2.1.1 Definiciones previas

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

$$time = (T,<)$$

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

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

$$w: <t_1, t_2> \rightarrow A \text{ o } w_{<t_1, t_2>}$$

El conjunto de todos los segmentos sobre A y T se denota como (A,T).
Un segmento $w: <t_1, t_2> \rightarrow R$ sobre una baase de tiempo continua es llamado continuo si es continuo en todos los puntos $t \in <t_1, t_2>$. Un segmento continuo en piezas (piecewise) es continuo en todos los puntos excepto un número finito de puntos $t' \in <t_1, t_2>$.

Un segmento de eventos representa una secuencia de eventos ordenada en el tiempo. Sea $w: <t_1, t_2> \rightarrow A \cup \{\varnothing\}$ un segmento sobre una base de tiempo continua y un conjunto arbitrario $A \cup \{\varnothing\}$. Entonces w es un segmento de eventos si hay un número finito de puntos en el tiempo $t_1, t_2, \ldots, t_{n-1} \in <t_0, t_n>$ tales que $w(ti) = ai \in A$ para i=1,...,n-1 y $w(t) = \varnothing$ para todo otro $t \in <t_0, t_n>$. ∅ 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.
$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$
donde:

X: Conjunto de valores de entrada.

Y: Conjunto de valores de salida.

S: Conjunto de estados secuencial.

$\delta_{int}: S \rightarrow S$ Función de transición interna.

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

λ: S$\rightarrow$Y Función de salida.

$ta:S \rightarrow R_0^+ \cup \infty$. 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 λ(s) y cambia el estado a δ_{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 $\leq$ ta(s), el sistema cambia al estado δ_{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´).

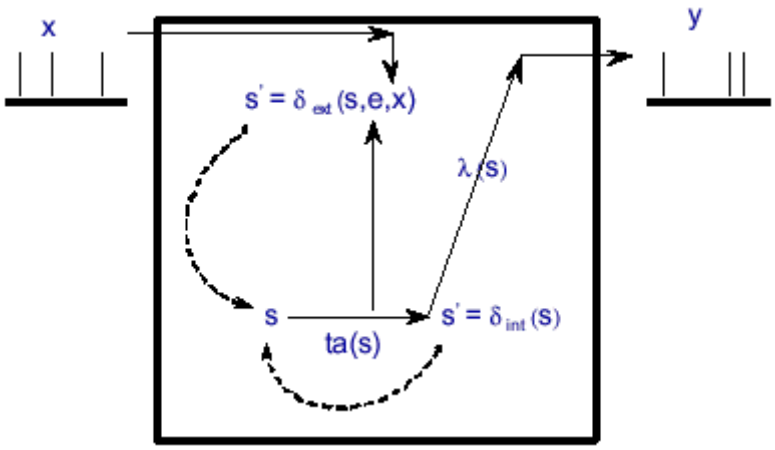


Figura 1 Funcionamiento de un modelo DEVS

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

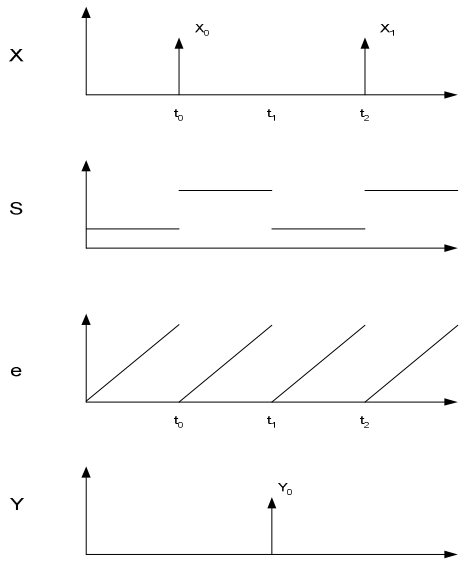


Figura 2 Trayectorias

Aquí la trayectoria de entrada es una serie de eventos que ocurren a intervalos de tiempo tales como t0 y t2. En medio de estos tiempos de arribo de eventos puede haber otros tales como t1 que son tiempos de arribo de eventos internos. Los ultimos 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:

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

donde:

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

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

S: Conjunto de estado secuencial.

$\delta_{int}: S \rightarrow S$ Función de transición interna.

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

λ: $S \rightarrow Y$ Función de salida.

$ta: S \rightarrow R_0^+ \cup \infty$. 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:

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

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 $d \in D$

$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ *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:

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

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

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

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

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

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

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

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

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

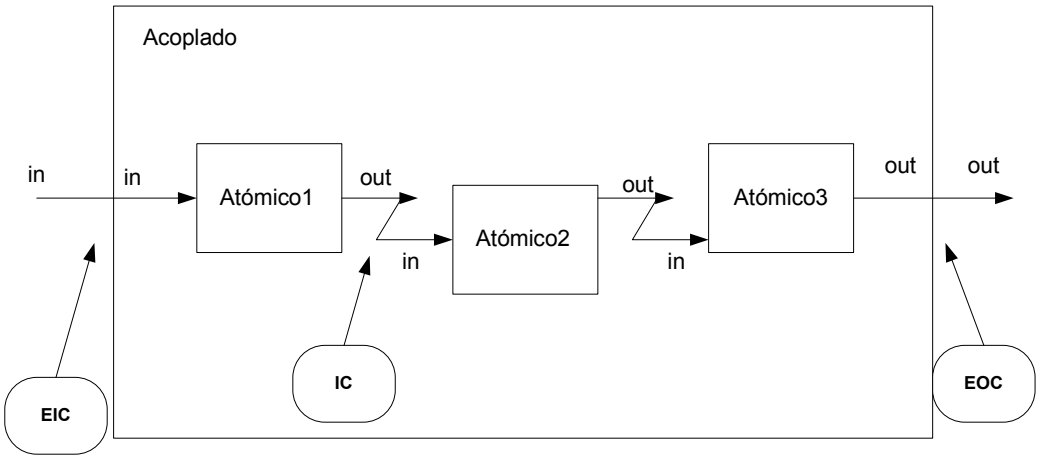


Figura 3 Modelo DEVS Acoplado

2.2 Modelos DEVS Paralelo

El formalismo DEVS Paralelo [Cho94] 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:

- Manejo de colisiones: el comportamiento en caso de colisión debe ser controlable por el modelador.
- Paralelismo: el formalismo no debe usar ninguna función de serialización que prohíba posibles concurrencias.
- 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:

$$DEVS = < X_M , Y_M , S, \delta_{ext} , \delta_{int}, \delta_{con}, \lambda, ta)$$

donde:

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

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

S es el conjunto de estados secuenciales;

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

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

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

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

$ta : S \rightarrow R_0^+ \cup \infty$ 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:

- La interfase del modelo incluye puertos y valores.
- Las funciones de transición externa y salida pueden recibir más de un evento simultáneamente, porque han sido extendidas para recibir bolsas de eventos.
- 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 $\lambda(s)$ y cambia al estado $\delta_{int}(s)$. Si uno o mas eventos externos $E = \{ x_1 .. x_n / x \in X_M \}$ ocurren antes que el tiempo ta(s) transcurra, o sea cuando el sistema esta en el estado total (s, e) con e ≤ ta(s), el nuevo estado estará dado por $\delta_{ext}(s,e,E)$. 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 $\delta_{ext}(\delta_{int}(s),e,E)$ o $\delta_{int}(\delta_{ext}(s,e,E))$. Para evitar un comportamiento fijo, se puede definir el comportamiento más adecuado con la función δ_{con}. Por lo tanto en el formalismo DEVS Paralelo, en presencia de colisiones, el nuevo estado del sistema será el definido por $\delta_{con}(s,E)$.

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 $d \in D$

$M_d = (X_d , Y_d S, \delta_{ext} , \delta_{int}, \delta_{con}, \lambda, ta)$ *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:

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

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

- *acoplamientos de salidas externas (EOC)* conectas las salidas de los componenetes a las salidas externas del modelo:

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

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

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

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

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

- 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.

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

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

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

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:

- 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.
- 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.
- 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

- Cada modelo básico de un formalismo es implementado por una clase propia.
- Cada clase de modelo atómico tiene su propia clase que la simula.
- Un modelo acoplado de cada formalismo es soportado por una clase de modelos acoplados.
- Cada clase de modelo acoplado tiene una clase simuladora llamada coordinador.
- Los simuladores utilizan un protocolo basado en mensajes que permite coordinar la simulación con otros objetos.
- 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.

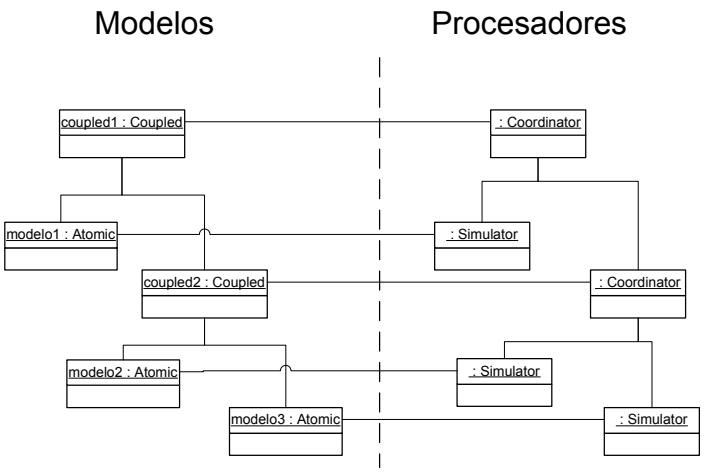


Figura 4 Relación entre modelos y simuladores

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.

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.

3.2.1 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.

*3.2.2 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.

3.2.3 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 inminente[1], 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.

3.2.4 Mensaje listo

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.

[1] Inminente: que tiene minimo tiempo al proximo evento.

3.3 Simulador

El simulador DEVS utiliza dos variables t_l (timestamp del ultimo evento) y t_n (timestamp del próximo evento agendado. Por la definición de DEVS surge que:

$$t_n = t_l + ta(s)$$

Dado t, la hora actual de la simulación, el tiempo transcurrido desde el último evento es:

$$e = t - t_l$$

y el tiempo restante al próximo evento:

$$\sigma = t_n - t = ta(s) - e$$

3.3.1 Simulador DEVS [Zei00]

(esta versión no incorpora los mensajes listo).

```
begin Devs-simulator
variables:
      parent      // parent coordinator
      tl    // time of last event
      tn    // time of next event
      DEVS  // associated model with total state (s, e)
      y     // current output value of the associated model
when a ( i , t ) message is received
      tl = t - e
      tn = tl + ta(s)
end when
when a ( *, t ) message is received
      if t <> tn then
            error: bad synchronization
      end if
      y = λ(s)
      send y-message (y, t) to parent coordinator
      s = δint(s)
      tl = t
      tn = tl + ta(s)
end when
when a (x, t) message is received
      if not ( tl <= t <= tn ) then
            error: bad synchronization
      end if
      e = t - tl
      s = δext(s, e, x)
      tl = t
      tn = tl + ta(s)
end when
end Devs-simulator
```

Coordinador

```
begin Devs-coordinator
variables:
      DEVN = (X, Y, D, {Md}, {Id}, {Zi,d}, Select) // red asociada
      parent      // parent coordinator
      tl    // time of last event
      tn    // time of next event
```

```
      event-list // lista de elementos (d, tn_d) ordenada por tn_d y
Select
      d*    // hijo inminente
end when
when a ( i , t ) message is received
      for each d in D do
            send message (i, t) to child d
      sort event-list according to tn_d and Select
      tl = max { tl_d | d ∈ D }
      tn = min { tn_d | d ∈ D }
end when
when a ( * , t ) message is received
      if t <> tn then
            error: bad synchronization
      end if
      d* = first( event-list )
      send (*, t) to d*
      sort event-list according to tn_d and Select
      tl = t
      tn = min { tn_d | d ∈ D }
end when
when a ( x , t ) message is received // input x
      if not ( tl <= t <= tn ) then
            error: bad synchronization
      end if
      // consult external input to get children influenced by the
input
      receivers = { r | r ∈ D, N ∈ I_r, Z_N,r (x) ≠ Φ }
      for each r in receivers
            send (x_r, t) with input value X_r = Z_N,r (x) to r
      sort event-list according to tn_d and Select
      tl = t
      tn = min { tn_d | d ∈ D }
when a ( y_d* , t ) message is received // output y_d* from d*
      // check external coupling to see if there is an external output
      // event
      put Y_d* of d*
      receivers = { r | r ∈ D, d* ∈ Ir, Z_d*,N(Y_d*) ≠ Φ }
      for each r in receivers
            send (x _r, t) with input value X_r = Z_d*,N(Y_d*) to r
end when
end Devs-coordinator
```

Coordinador raíz

```
begin Devs-root-coordinator
variables:
      t     // tiempo actual de simulación
      child  // hijos directos (coordinadores o simuladores)
t = t0
send initialization message (i, t) to child
t = tn of its child
loop
      send (*, t) to child
      t - tn of its child
until end of simulation
end Devs-root-coordinator
```

3.3.2 Simulador para DEVS Paralelo

En [Cho94] 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).

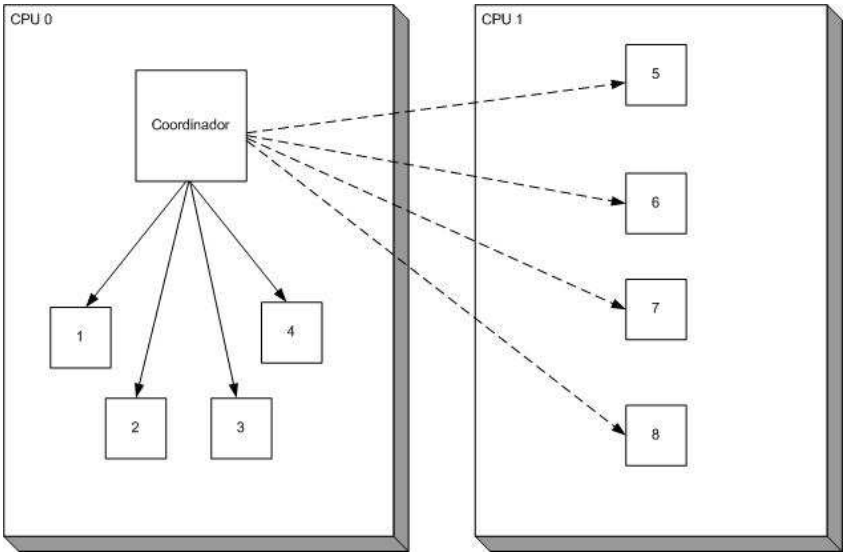


Figura 5 Un coordinador envía mensajes a sus hijos.

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).

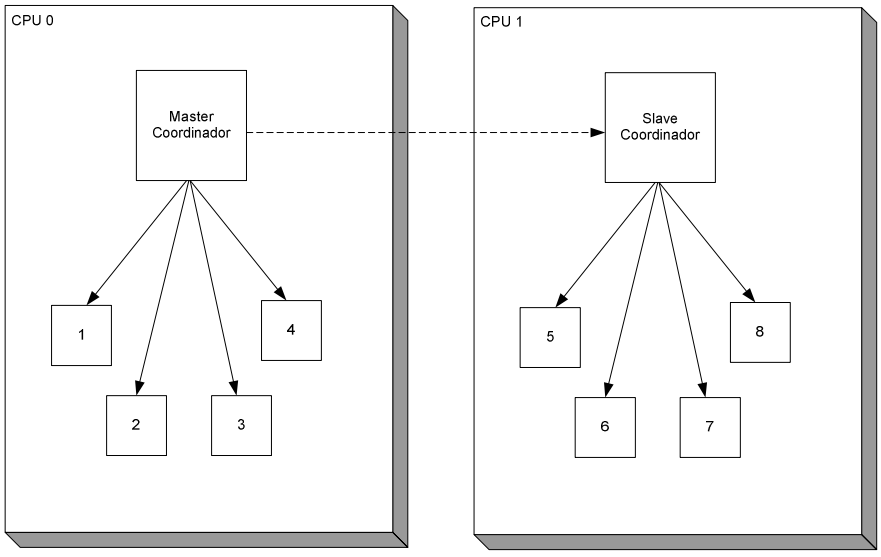


Figura 6 Un maestro envía un mensaje a través del esclavo.

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 $t = t_N$ **then**
        $y := \lambda(s)$
        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 $t_L \leq t < t_N$
        $e := t - t_L$
        $s := \delta_{ext}(s, e, bag)$
        empty *bag*
    end case
    case $t = t_N$ and *bag* is empty
        $s := \delta_{int}(s)$
    end case
    case $t = t_N$ and *bag* not is empty
        $s := \delta_{con}(s, bag)$
        empty *bag*
    end case
    case $t > t_N$ or $t < t_L$
        raise error
    end case
    $t_L := t$
    $t_N := ta (s)$
    send (*done*, t_N) to parent coordinator
end when

Función auxiliar coordinador

coordinator : M x P → 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 := 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 = 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
        sent_to_master := false
        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
                        if not sent_to_master
                                send ( y, t ) to parent coordinator
```

```
                                        sent_to_master := true
                                end if
                        end if
                end for
                if self ∈ I_i ( 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 t_L ≤ t ≤ t_N
        for all q ∈ bag
                for all receivers of q,  j ∈ I_self
                        if j is a local processor
                                q := z_self, j (q)
                                send ( q, t ) to j
                                cache j in the synchronize set
                        else
                                do nothing
                        end if
                end for
        end for
        empty bag
        for all i in the synchronize set
        send ( *, t ) to i
          end for
        wait until all ( done, t_N)'s are received
        t_L := t
        t_N := minimum of components' t_N's
        clear the synchronize set
        send ( done, t_N ) 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 clases que implementan en código C++ 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 del CD++ esta 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 esta 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].

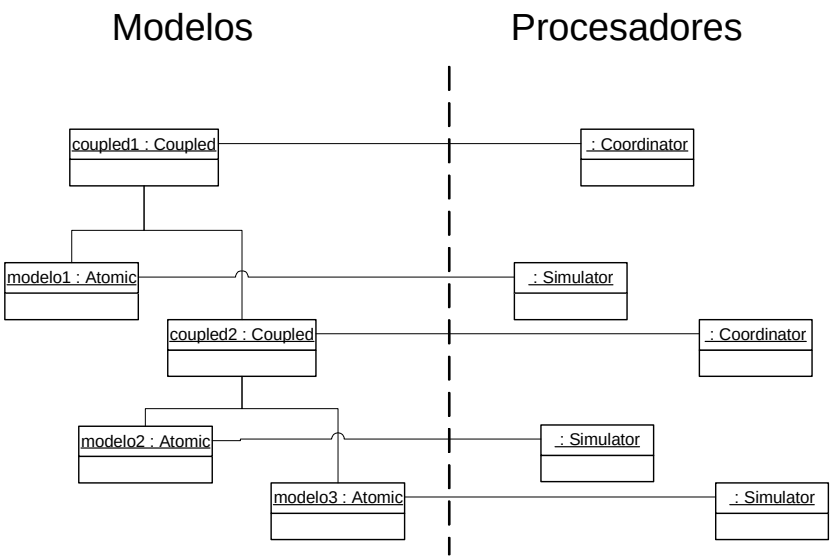


Figura 7 Relaciones entre modelos y procesadores

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.

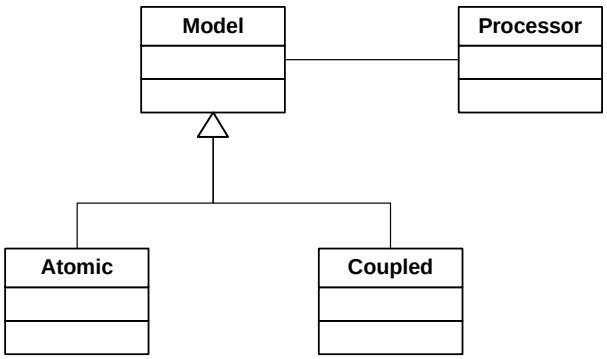


Figura 8 Jerarquía de Modelos de CD++

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

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

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

• ***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

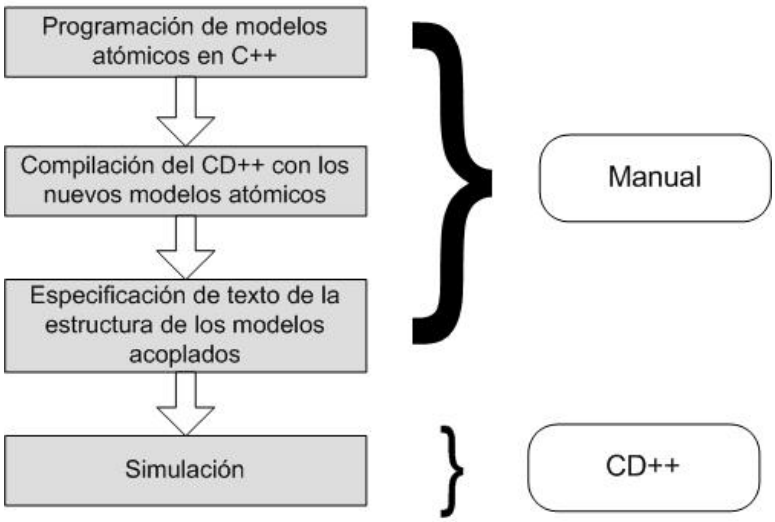
5 GGAD, una nueva representación de modelos DEVS

5.1 Motivación

Las ventajas de la simulación son múltiples: puede reducirse el tiempo de desarrollo, las decisiones pueden chequearse artificialmente, y un mismo modelo puede usarse muchas veces. Por otro lado, la simulación es de empleo mas simple que ciertas técnicas analíticas y precisa menos simplificaciones de los modelos empleados.[Wai96]

Con este criterio en mente, el objetivo es disminuir la barrera de entrada existente para el uso de técnicas de simulación DEVS.

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
• Mayor expresividad • Menos intuitiva • Herramientas disponibles (editores de texto)	• Menor expresividad • Más intuitiva • 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 mas 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 asi 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 representacion 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 grafico 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.

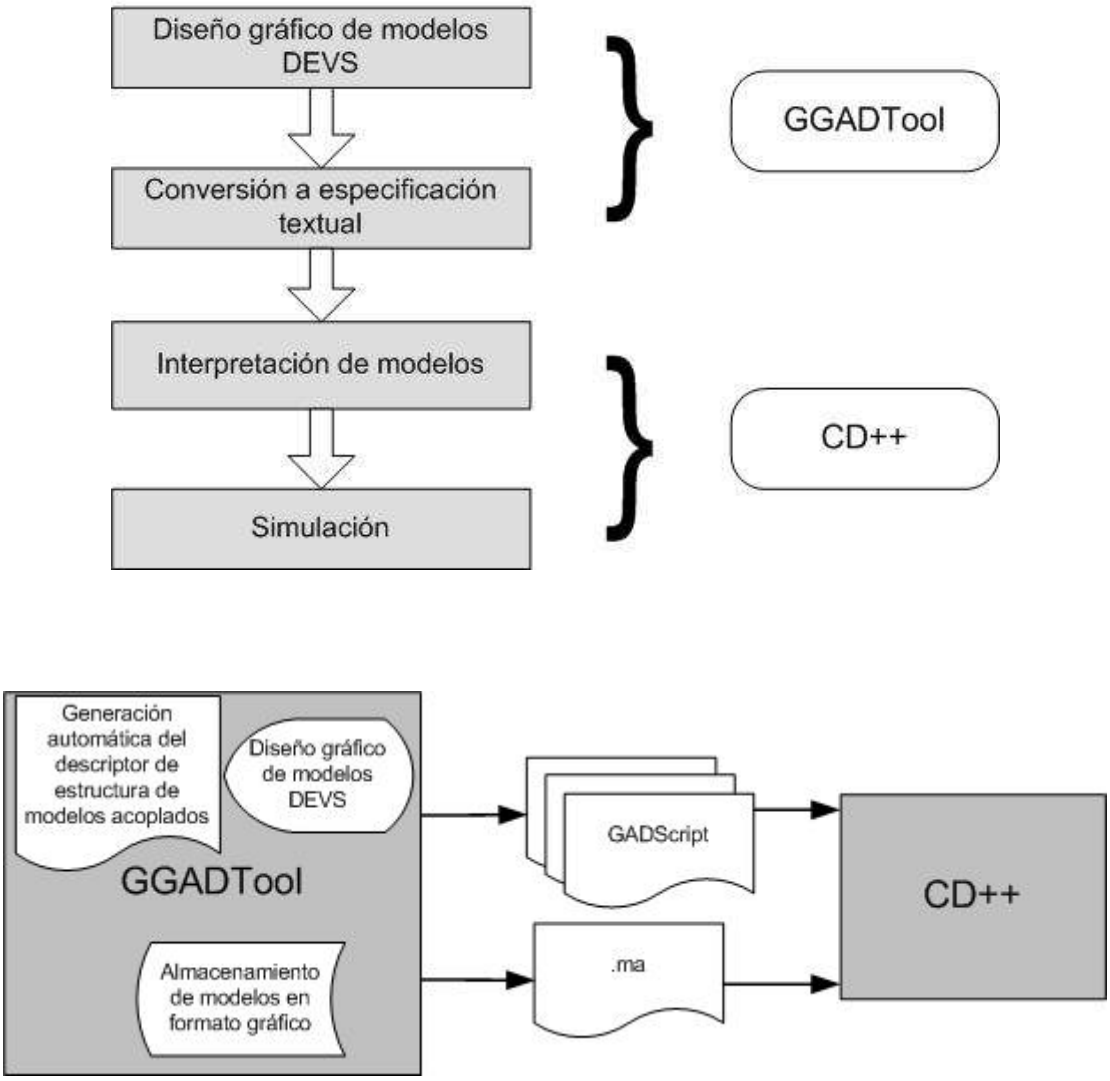


Figura 9 División de roles

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 atomicos GGAD. Los mismos estan compuestos por estados, variables, puertos y transiciones.

5.2.1 Estados

Un modelo atomico GGAD en un momento dado de la simulación esta en un determinado estado. El estado esta dividido en dos partes: un estado nominado y los valores del conjunto de variables del modelo.
Los estados nominados son representados graficamente por circulos con su nombre en el centro, y por ello son llamados "burbujas". Se utilizan en el mismo sentido que los estados en los automatas 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.

5.2.2 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.

5.2.3 Puertos

La interacción con otros modelos de la simulación y el exterior se realiza a traves 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 envio 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 mas de un mensaje, incluso varios mensajes del mismo puerto. Si el modelo tiene puertos a y b algunos ejemplos de mensajes podrian ser :

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

5.2.4 Transiciones

Los cambios de estado del modelo se producen al ejecutar transiciones. Cada estado puede estar asociado a una o varias transiciones que definen cual es el nuevo estado del modelo y como cambian los valores de las variables.
Las transiciones pueden ser internas o externa dependiendo de a que 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 estan asociadas a la aparición de mensajes en los puertos de entrada del modelo. Estan 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 ambiguedad. 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 caracteristicas 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.

5.3.1 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.

5.3.1.1 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

5.3.1.2 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

5.3.1.3 Regla var

Sintaxis

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

Declara variables pertenecientes al modelo.
variable1 variable2 ... variablen son nombres de variables.

5.3.1.4 Regla state

Sintaxis

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

Declara estados del modelo. Corresponden a las "burbujas" de la representación grafica. No confundir los estados con los estados secuenciales.

5.3.1.5 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.

5.3.1.6 Regla de estado inicial

Sintaxis:

```
inital : statename
```

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

5.3.1.7 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:

$$\lambda\ (ESTADO\_INICIAL) = \{\ (\ PUERTO\_DE\_SALIDA\ !\ Expresion\ )+\ \}$$

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`.

5.3.1.8 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 evalue 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.

5.3.1.9 Expresiones y acciones

Estas no son reglas sino partes de las mismas.

5.3.1.9.1 Expresiones

Las expresiones pueden ser:

- constantes numericas
- puertos
- variables
- funciones

5.3.1.9.2 Constantes numericas

Ejemplos:

- 5
- -10
- 0

5.3.1.9.3 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.

5.3.1.9.4 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.

5.3.1.9.5 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.

5.3.1.9.6 Acciones

Las acciones son la unica forma de cambiar el estado total del modelo, ademas 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 grafica

La representación grafica de modelos es una simplificación de la representación textual que permite una forma sencilla de diseñar modelos DEVS. No encontramos practico o util representar graficamente 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.

5.4.1 Estados

Los estados no representan el estado total del modelo sino que son un componente mas del mismo, junto con las variables.
Se representan graficamente con un círculo y el nombre del estado adentro

Ejemplo:

5.4.2 Transiciones

Hay dos tipos de transiciones: externas e internas.

5.4.2.1 Transiciones externas

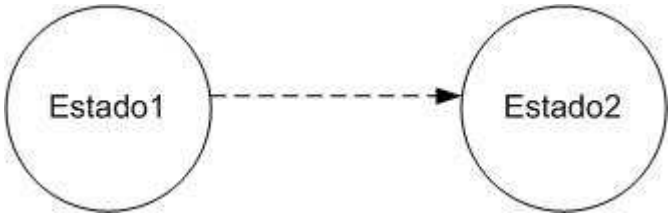
Se representan graficamente con una linea continua con una flecha en la dirección al destino de la transición.

Ejemplo:

5.4.2.2 Transiciones internas

Se representan graficamente con una linea discontiua con una flecha en la dirección al destino de la transición.

Ejemplo:

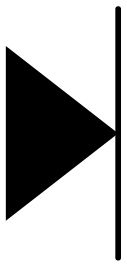


5.4.3 Puertos

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

5.4.3.1 Puertos de entrada

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



5.4.3.2 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:

$$GGAD = < X_M , Y_M , S, \delta_{ext} , \delta_{int}, \delta_{con}, \lambda, ta >$$

donde:

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

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

$S = B \times P(V)$ es el conjunto de estados sequenciales,
donde:
$B = \{ b |\ b \in Burbujas \}$ es el conjunto de estados del modelo.
$V = \{ (v,n) | v \in Variables, n \in R_0 \}$ es el conjunto de variables del modelo y sus valores.
P(V) es el conjunto partes de V.

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

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

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

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

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

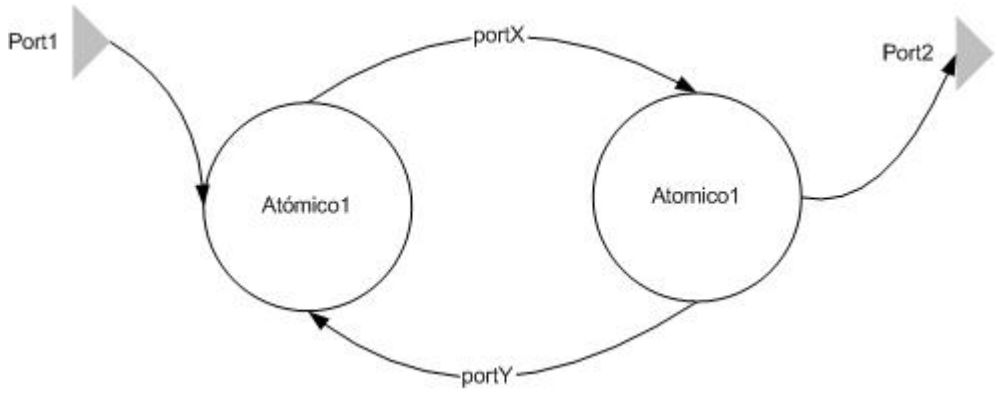
donde $Q := \{ (s, e) | s \in S , 0 \leq e \leq ta(s) \}$ es el conjunto de *estados totales*.

La función de confluencia se define como el resulto 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 el 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 del CD++ es:

```
[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
```

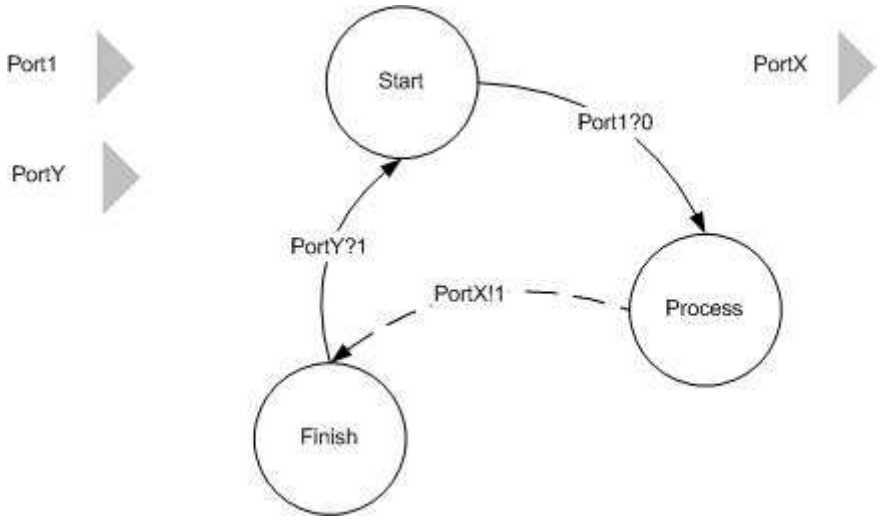
El formato de este archivo .ma del CD++ es el siguiente:
En la sección [Top] se describe el modelo acoplado raíz del sistema.

- 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 clasede modelo GGAD)
- 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
- 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

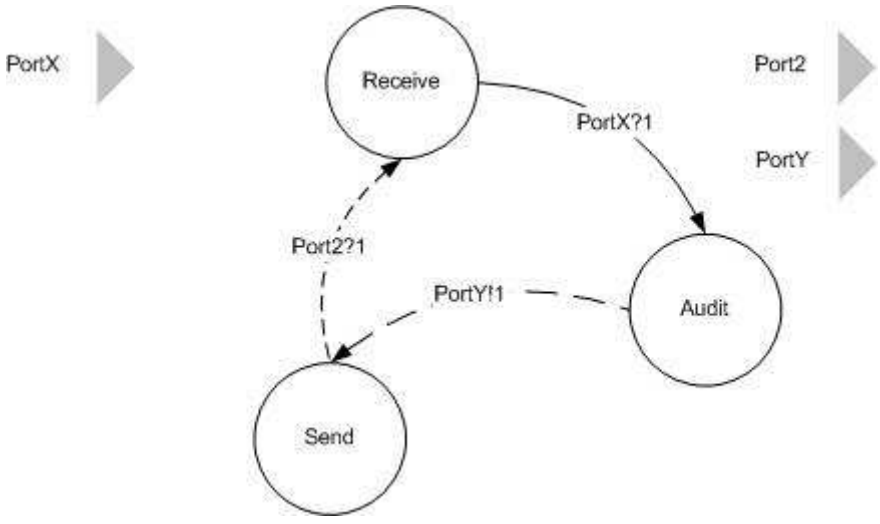


y su representación textual GADscript es::

Atomico1.cdd

```
[atomico1]
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 atomico2 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 especifico sino que utiliza el mismo simulador que un modelo DEVS Clasico.

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 el 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 teorico del punto anterior, nótese que en el listado de modelos que componen al modelo acoplado los modelos atomicos 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 puden crear modelos acoplados con componentes basados en distintos formalismos.

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 atomico. No fue necesario modificar el funcionamiento del simulador, sino que se extendio su comportamiento utilizando los mecanismos previstos para incorporar nuevos modelos. Solo se usaron las interfaces publicadas para dicho fin. El modelo atomico GGAD esta desacoplado del funcionamiento del resto del CD++ para evitar interdependencias innecesarias con otros componentes. De esta forma se evito que cambios en la estructura del CD++ impacten significativamente en el desarrollo. Fisicamente la implementación modelo atomico GGAD esta provista por una libreria.
Por convención todos los objetos tienen el prefijo ggad (o una variante, por ej: GGAD). Se han utilizado solamente tecnicas de programación validas 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 mas importantes y como se relacionan.

La clase Ggad implementa el modelo atomico y funciona como punto de acceso del simulador. Junto con GgadDynamicState (que mantiene el estado del modelo) son las unicas clases visibles por el CD++. Ggad recibe los mensajes del CD++ y delega en GgadImpl la implementación del modelo. Al estar separada de la atención de eventos y la interación con CD++ que es responsabilidad de Ggad, es posible cambiar la implementación del modelo sin afectar el resto del modulo, 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:
- Atención de eventos del simulador
- Logging de cambios de estado

Clase GgadImpl

Responsabilidades:
- Implementar el modelo.

Clase GgadParser

Responsabilidades:
- Construir un modelo a partir de una especificación textual.

Clase GgadSymbolTable

Responsabilidades:

- Implementar una tabla de simbolos para registrar los objetos.

Clase GgadDynamicState

Responsabilidades:

- Almacenar el estado dinamico del modelo.

Clase GgadDynamicStateAdapter

Responsabilidades

- 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 párametro un objeto GgadDynamicStateAdapter que permite acceder al estado actual del modelo.

Responsabilidades:

- Definir la interfaz un árbol sintactico.

Clase GgadActionNode
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

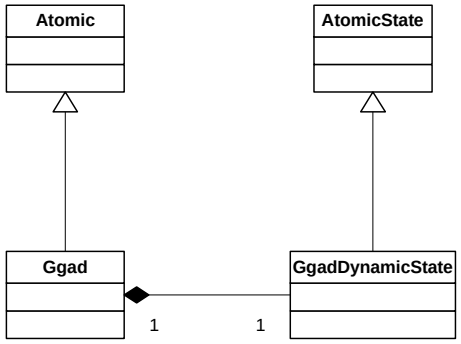


Figura 10 Clases Ggad y GgadDynamicState

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:

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

Luego hay una detalle que depende de cada evento, separado por el simbolo ':'.

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 envia 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) , ... }

7 Ejemplos

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.

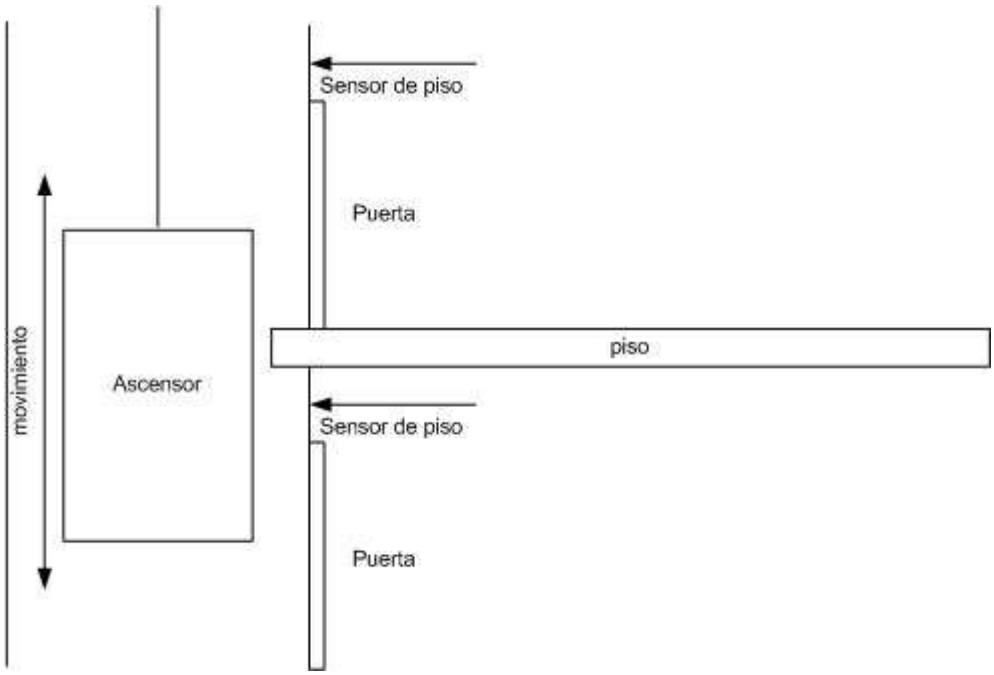
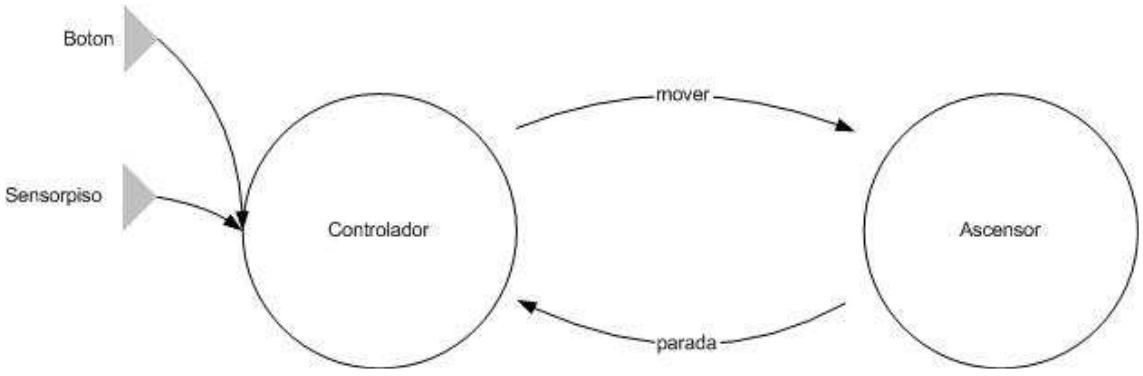


Figura 11 Diagrama conceptual del ascensor.

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”.

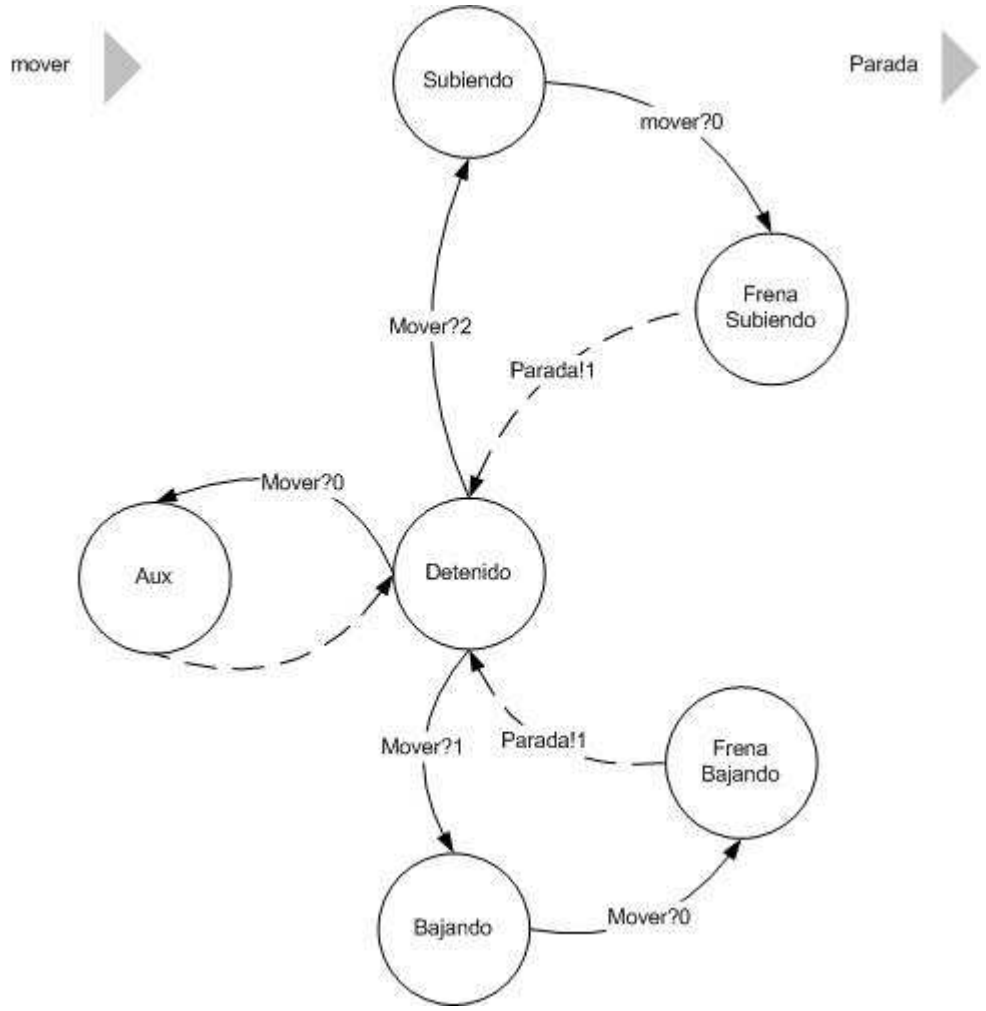
AscensorSimple.ma

```
[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:

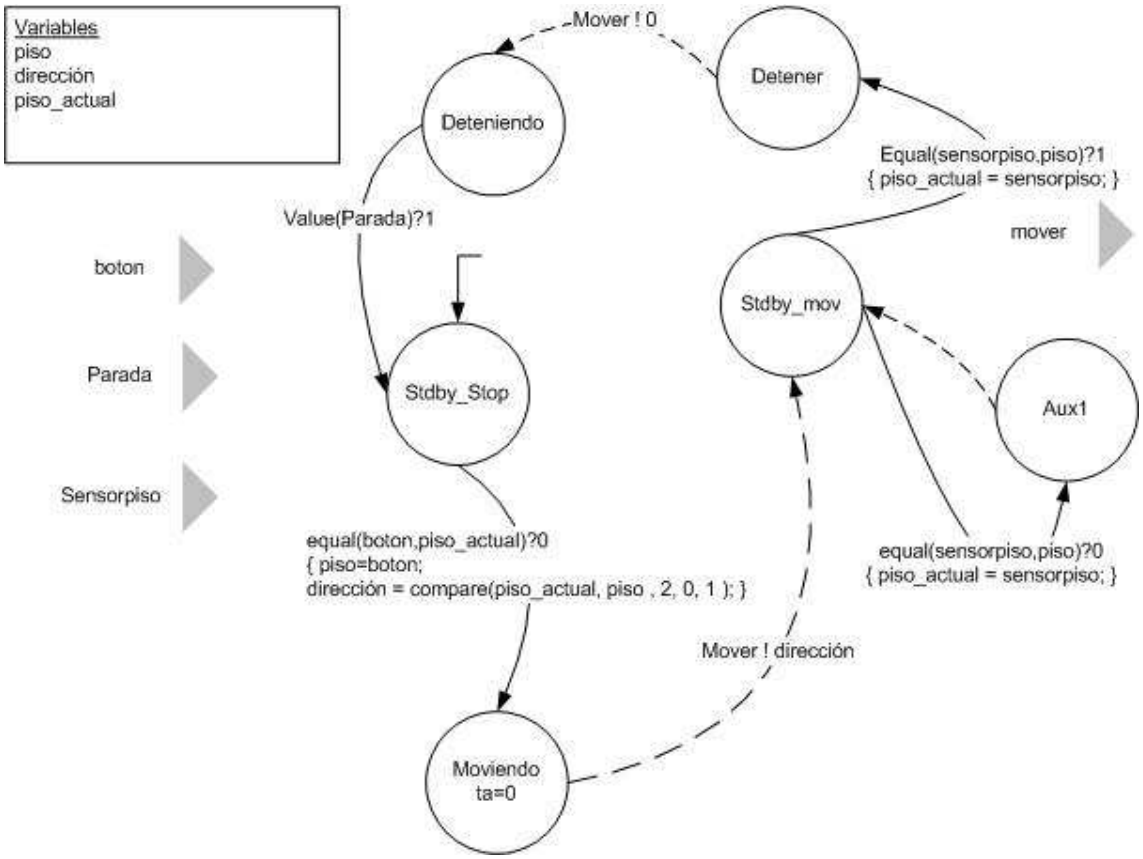
El ascensor solo acepta ordenes de ponerse en marcha cuando está detenido. Si recibe un 2 en el puerto "mover", significa que suba y si recibe un 1 es que baje. Estando en movimiento espera recibir un 0 en el puerto "mover" para iniciar el frenado. Una vez que logra detenerse emite un 1 por el puerto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

- 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.
- Parada: por este port llega un valor "1" cuando el ascensor se detiene
- SensorPiso: recibe el número de piso al que esta por llegar el ascensor

Y un puerto de salida:

- Mover: le pasa los comandos de movimiento al ascensor

También posee 3 variables:

- Piso: almacena el piso al que está yendo
- Dirección: es para calcular si tiene que ir arriba o abajo cuando es llamado
- 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 envia mensajes y la otra es receptora, a traves de un canal por el que se envian dos mensajes (M0 y M1) que representan una secuencia alternada de datos (M0M1M0...).
El emisor envia el primer mensaje y espera una confirmación del receptor de ese mensaje. Si lo recibe envia el siguiente mensaje y espera una confirmación del segundo. Si no se recibe confirmación, o se recibe confirmación del mensaje erroneo el emisor reenvia el mensaje. El nombre del protocolo proviene del hecho que ambos mantienen un estado representando el mensaje M0 o M1 y alternan entre ambos.

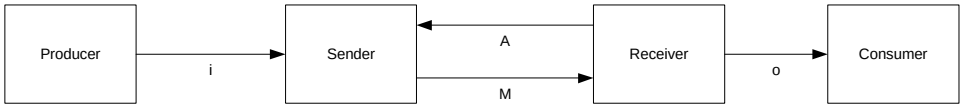


Figura 12 El emisor envia mensajes al receptor mientras que el receptor confirma los mensaje recibidos. Notese que hemos separado el productor y el consumidor del mensaje del protocolo.

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

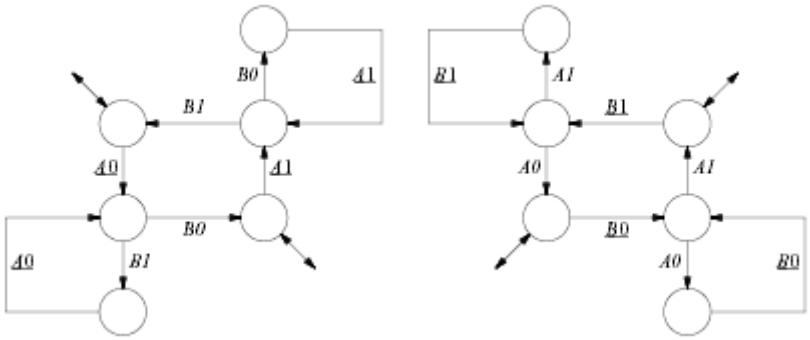


Figura 13 Protocolo de Bit Alternado original.

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 envie 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.

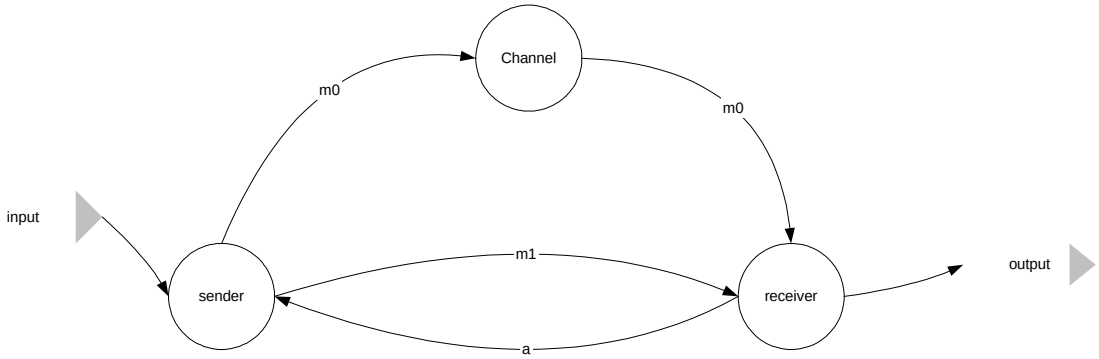


Figura 14 Modelo acoplado para Protocolo de Bit Alternado

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

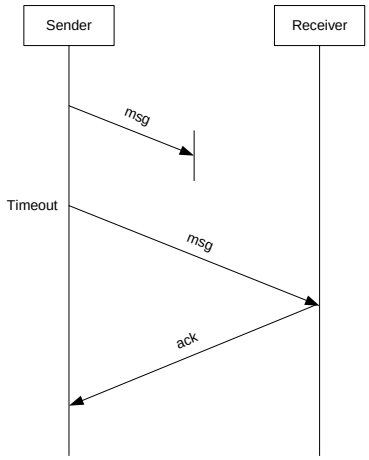


Figura 15 Secuencia de error de recepción.

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.

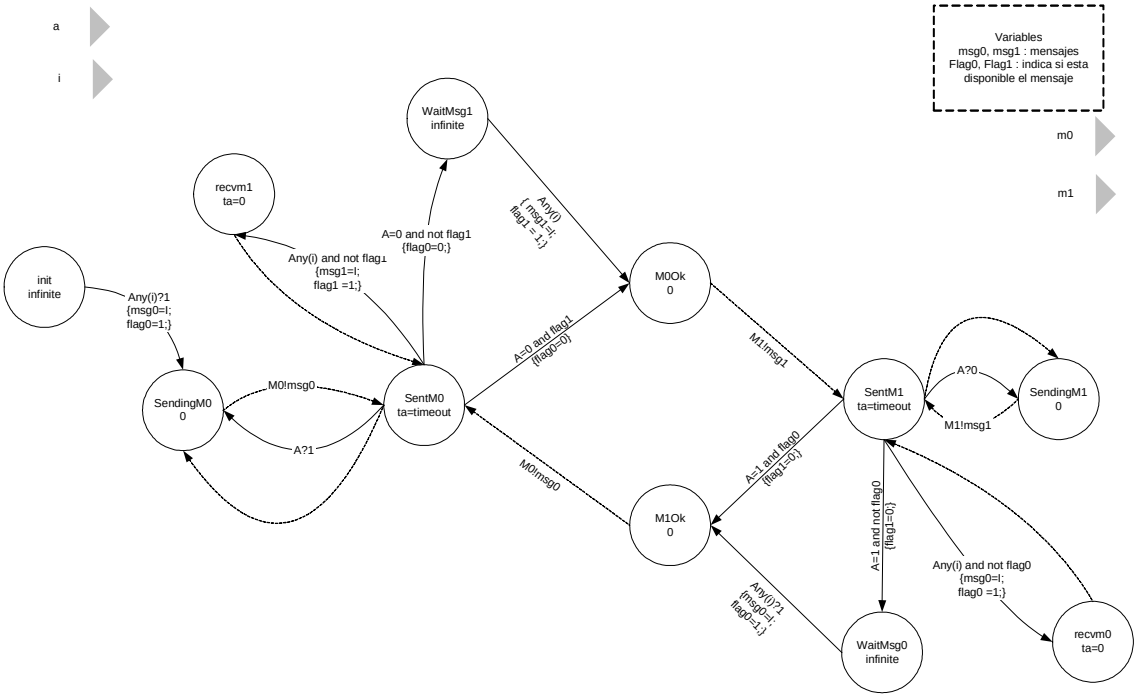


Figura 16 Modelo átomico Sender

```
[sender]
in: i a
out: m0 m1
var: msg0 msg1 flag0 flag1
state: start sentm0 m0ok m1ok sentm1 recvm0 recvm1 waitmsg1
sendingm0 WaitMsg0 sendingm1
initial:start
int: sendingm0 sentm0 m0!msg0
int: recvm1 sentm0
...
ext: sentm0 recvm1 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

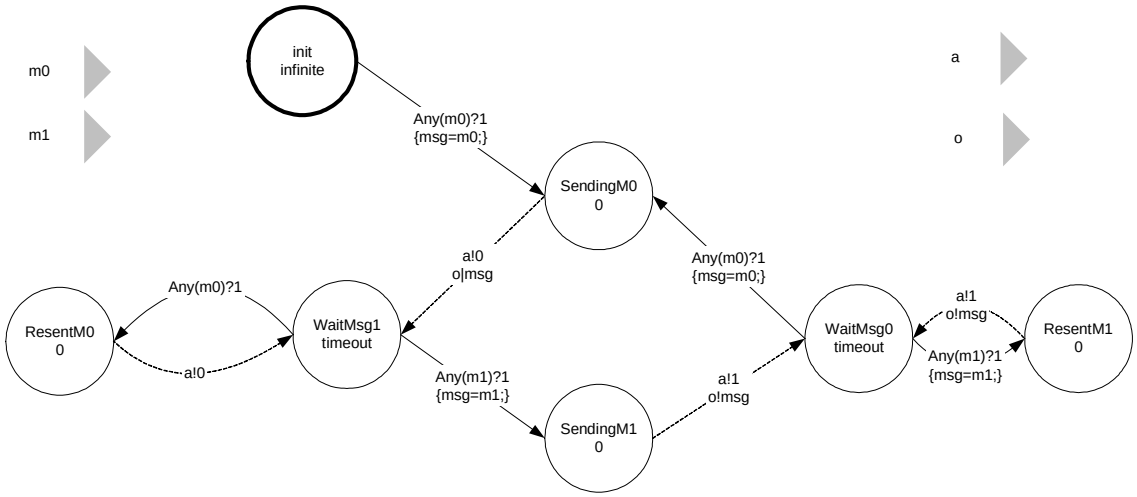


Figura 17 Modelo átomico receiver.

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.

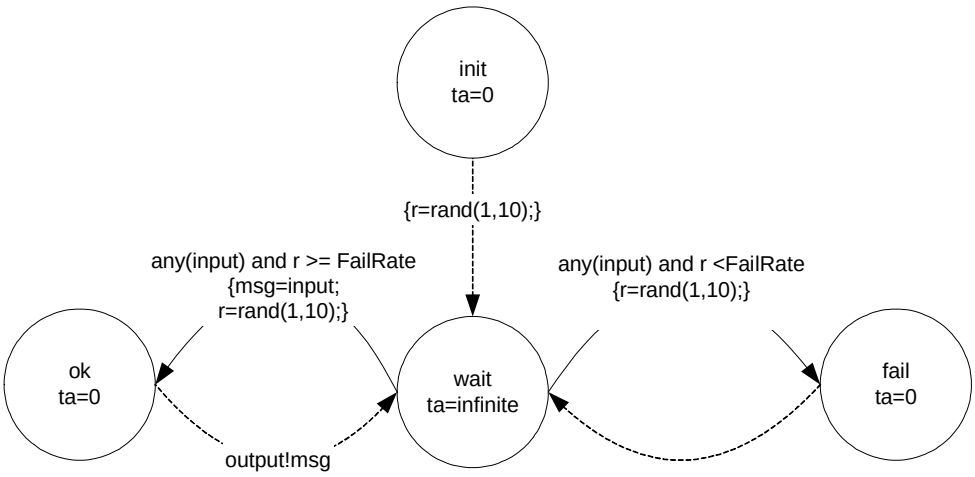


Figura 18 Modelo átomico Channel.

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 envia 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 envio 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 , m1ok (flag0=1) (flag1=0) (msg0=3) (msg1=2)
O 00:01:30:000 : m0 , 3
I 00:01:30:000 : m1ok , 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 tambien 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 tenia error.

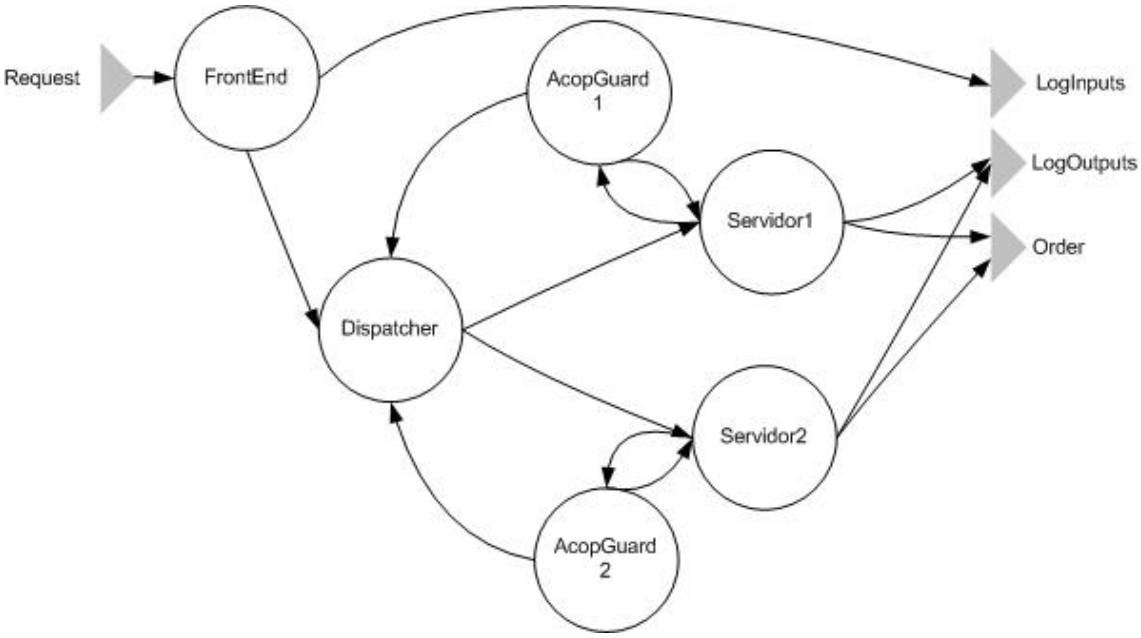
7.3 Load balancer:

7.3.1 Descripción

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.

7.3.2 Diagramas

El modelo acoplado puede verse en la siguiente figura:



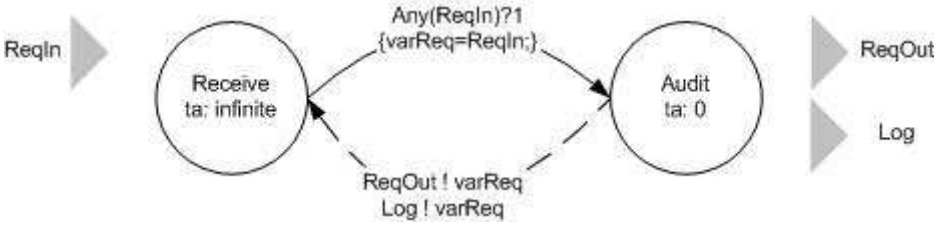
Hay 4 puertos:

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

El diseño del modelo consta de :

- Un modelo atómico llamado FrontEnd que es el que recibe los request y registra la entrada de los mismos.
- Un modelo atómico Dispatcher se encarga de enviarle el requerimiento al servidor en cuestión
- Dos modelos atómicos de Servidor que procesan el request y lo envían al port de salida del modelo acoplado
- 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.

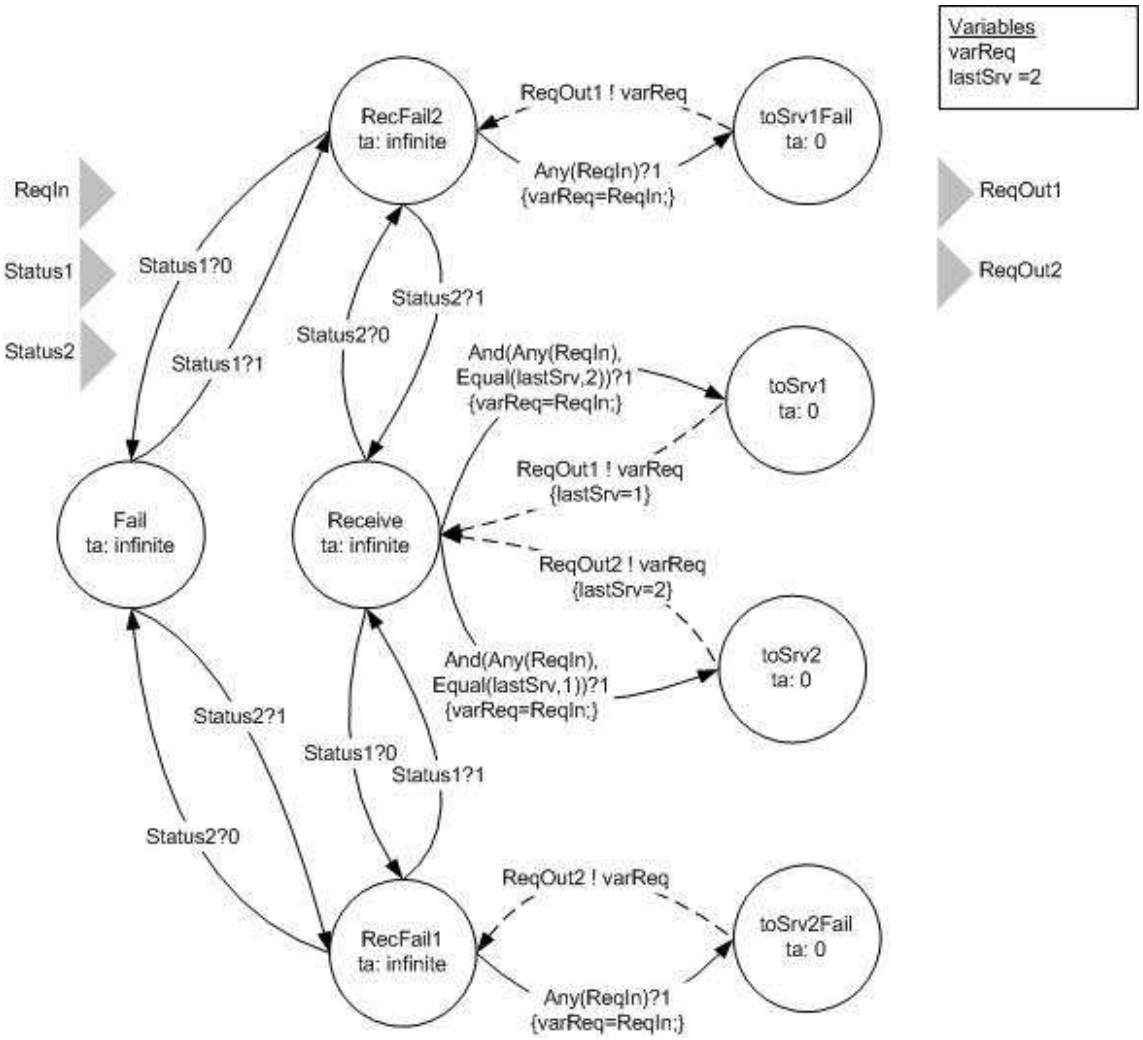
Modelo atómico FrontEnd:



Funcionamiento:

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

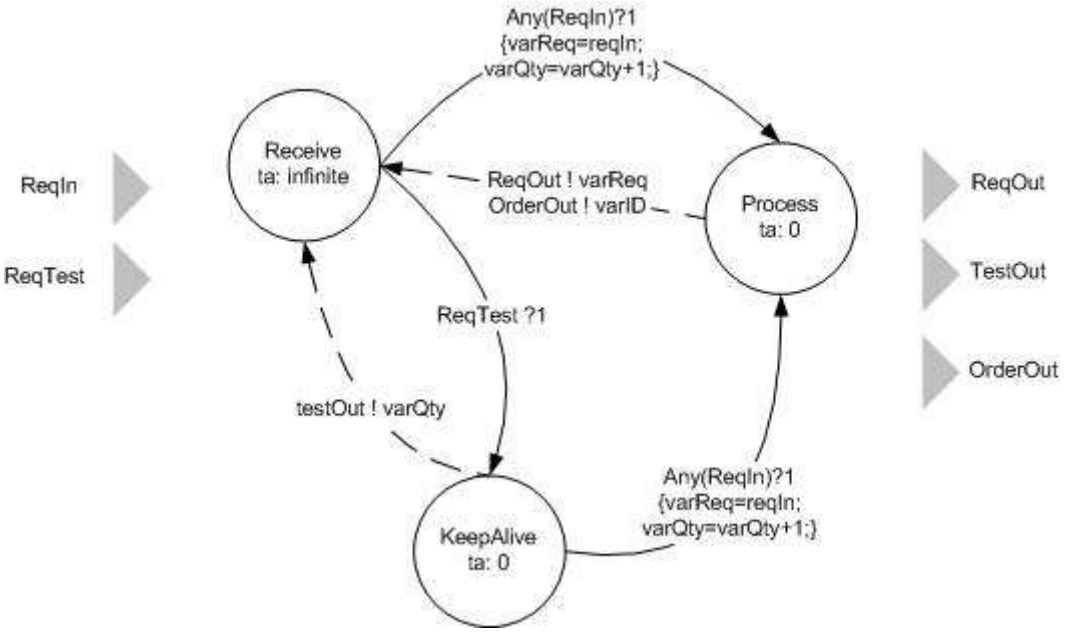
Modelo atómico Dispatcher



Funcionamiento:

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 caido está nuevamente operativo o que falló el otro servidor, en cuyo caso se va al estado Fail en el cual no procesa más requests.

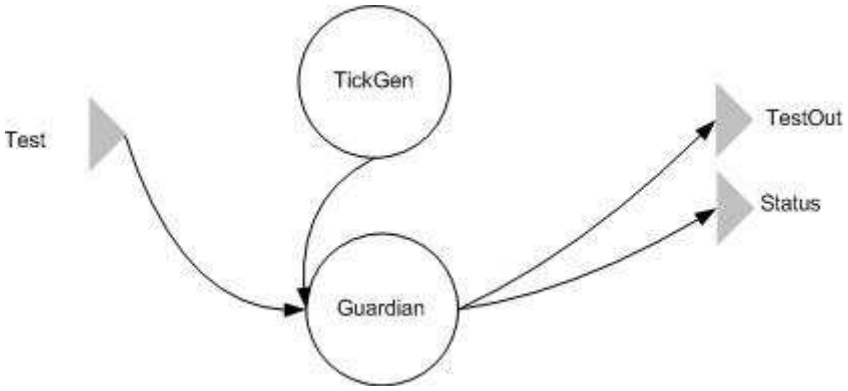
Modelo atómico Servidor (Servidor1 y Servidor2)



Funcionamiento:

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

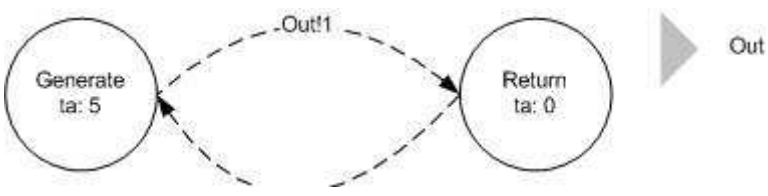
Modelo acoplado AcopGuard (Acopguard1 y Acopguard2):



Funcionamiento:

Este modelo se encarga de envíar 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

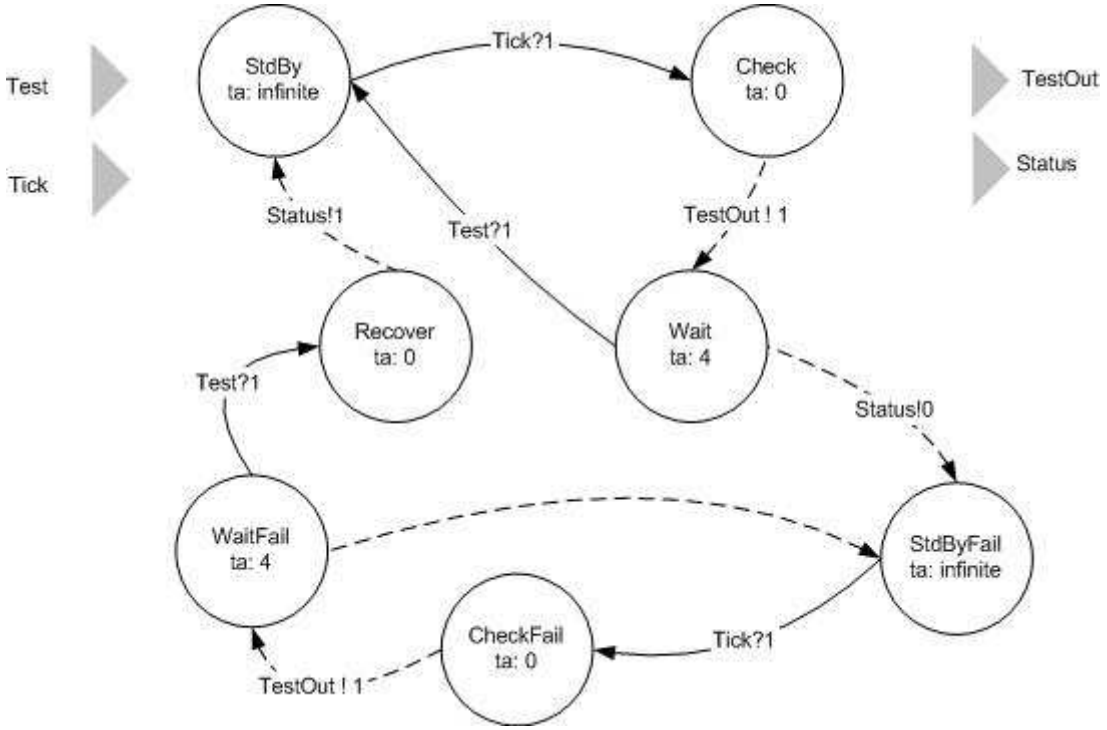
Modelo atómico TickGen (parte del acoplado Acopguard)



Funcionamiento:

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)



Funcionamiento:

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 emitira un mensaje de status OK al dispatcher y volverá al estado StdBy.

7.3.3 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
```

7.3.4 Simulación

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.

- 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

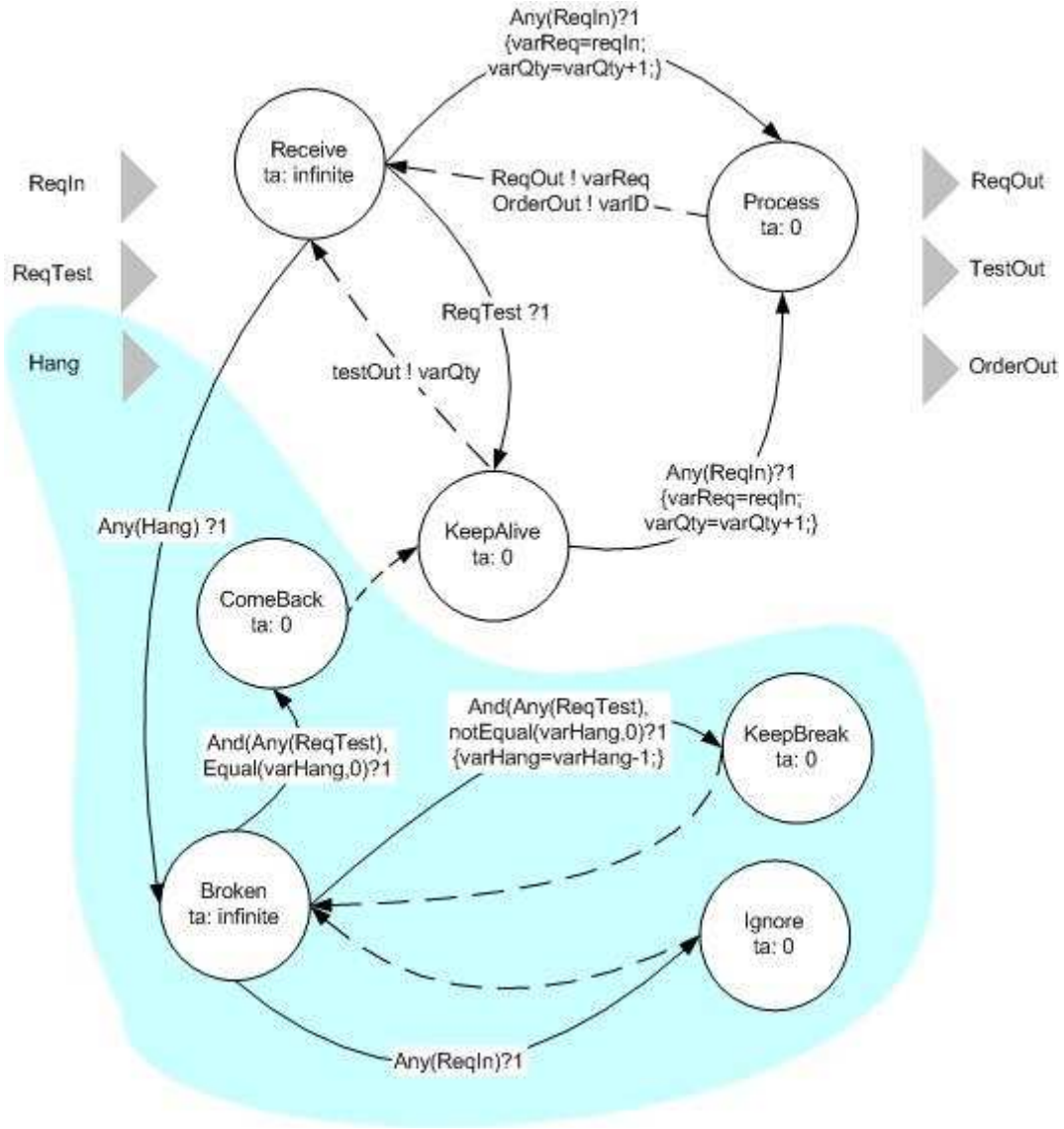
```
C 00:00:00:000 : receive ,  (varid=1) (varqty=0) (varreq=0)
? 00:00:02:000 : reqin , 1
E 00:00:02:000 : receive , process (varid=1) (varqty=1) (varreq=1)
O 00:00:02:000 : orderout , 1
O 00:00:02:000 : reqout , 1
I 00:00:02:000 : process , receive (varid=1) (varqty=1) (varreq=1)
? 00:00:05:000 : reqtest , 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

Ha sido presentada 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, se ha mostrado como pueden ser diseñados los modelos en una forma que es ejecutable por CD++ sin necesidad de programar en C++.

Entonces, con esta especificación gráfica y textual (GADscript) se ve facilitada la actividad de crear nuevos modelos y se acerca la simulación a un público más amplio

Implementamos esta especificación en la herramienta GGADTool en lo que refiere a la generación de nuevos modelos y en el CD++ respecto a la ejecución de las simulaciones.

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.

Futuros desarrollos relativos a este trabajo incluyen:

- La adaptación o extensión de esta especificación para cubrir otras variedades de modelos DEVS, como ser DEVS de tiempo real, etc.
- Las capacidades de la herramienta de diseño asi como su interacción con otros motores de simulación más allá del CD++

9 Referencias y bibliografía

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

[Zei95] ZEIGLER, BERNARD P.; KIM, T.; PRAEHOFER, H; SONG H. "DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems".

[Cho94] CHOW, ALEX C.; ZEIGLER, BERNARD P. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Winter Simulation Conference Proceedings*, Orlando, Florida, 1994. SCS.

[Wai96] WAINER, GABRIEL. “Introducción a la simulación de sistemas de eventos discretos”. Technical Report 96-005. Departamento de Computación FCEyN – UBA. 1996.

[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.

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

[Cho94] 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

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

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

[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.

[Tro01] TROCCOLI, ALEJANDRO. "Modificaciones a CD++ para simulación paralela y distribuida de modelos Cell-DEVS". Tesis de Licenciatura. Departamento de Computación FCEyN – UBA. 2001

[Hol91] HOLZMANN, GERARD J. "Design and Validation of Computer Protocols". Prentice Hall. 1991.