Modelado y simulación de Redes de Petri con el formalismo DEVS

Taihú A. N. Pire

<taihup@gmail.com>

Legajo: P-2999/8

Universidad Nacional de Rosario.

Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Tesina de Grado - Licenciatura de Ciencias de la Computación

Director : Ernesto Kofman

<kofman@cifasis-conicet.gov.ar>

Co-director : Federico Bergero

<bergero@cifasis-conicet.gov.ar>

Octubre, 2010

Índice general

Índice de figuras

Agradecimientos

Con el objeto de decorar un poco los agradecimientos hacia las personas, que de una u otra manera, formaron parte del presente trabajo, pensé en coronarlos con las distintas piezas de un juego de ajedrez:

Quiero entregar las **Torres** a Ernesto Kofman y a Federico Bergero por su acompañamiento y ayuda durante todo el trabajo, ellos fueron los pilares del mismo.

Delego los **Caballos** a mis amigos y compañeros de grado por formar parte de mi crecimiento, tanto académico como personal, durante la travesía de ser estudiante, en la que hubo saltos y caídas; caminatas y galopes.

El par de **Alfiles** se lo concedo a mi familia, ya que no solo sembraron en mi los valores y la confianza con los que piso cada día, si no también por que me impulsaron en cada centímetro del camino.

La **Dama** la deposito en las manos de mi novia, en ella se adormecieron dudas, pensamientos, lagrimas y risas. Además le agradezco especialmente, por soltar cuando quería que agarrara y apretar cuando lo fácil era soltar.

El **Rey** y los **Peones** los reservo para mi, porque si bien soy Rey por tener Dama nunca me quité el traje de Peón.

Taihú A. N. Pire

Capítulo 1

Introducción

La rápida evolución que se ha producido en el campo tecnológico en las últimas décadas, emergió junto con nuevos sistemas dinámicos como por ejemplo redes de computadora, sistema de producción automatizados, sistemas de control, etc. 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 lleva a definir el término Sistemas de Eventos Discretos (DES). Fue imprescindible también crear formalismos que permitan modelar, analizar y simular dichos sistemas dado la alta complejidad que estos tenían. Dentro de los formalismos más populares de representación se encuentran las redes de Petri (Petri Nets o PN), que proveen un lenguaje gráfico muy conveniente para modelizar y analizar distintos problemas. Por otra parte, orientado a los problemas de modelización y simulación de DES, ubicamos a DEVS (Discrete EVent System specification), el formalismo más general para el tratamiento de DES. El hecho de estar basado en teoría de sistemas, lo convierte en un formalismo universal, es decir, que en él pueden expresarse todos los modelos descritos por cualquier otro formalismo DES, en particular los reproducidos en PN.

Pese a su generalidad, DEVS no es un lenguaje gráfico, y en gran cantidad de casos es mucho más simple obtener un modelo en redes de Petri que hacerlo directamente en DEVS. Por otro lado, hay muchos modelos que no pueden representarse con redes de Petri (aproximaciones de sistemas continuos, por ejemplo, o modelos híbridos en general), y que el formalismo DEVS puede simular de manera muy eficiente.

Esto sugiere la conveniencia de utilizar ambos formalismos en conjunción. De hecho, hay un

antecedente del uso de PN en un simulador DEVS. Sin embargo, debido a las características del simulador utilizado (CD++), en dicho trabajo, los modelos tienen muchas restricciones y se deben agregar puertos y conexiones que no existen en la Red de Petri original.

En el presente trabajo se desarrolla e implementa una metodología que permite utilizar redes de Petri dentro de una herramienta de modelado y simulación de DEVS denominada *PowerDEVS*. A diferencia de lo desarrollado en CD++, esta implementación respeta la apariencia gráfica de las redes de Petri, incluyendo temporización, asignación de prioridades e interconexión con otros modelos DEVS (incluyendo aproximaciones de modelos continuos). Además de describir la implementación, se muestran ejemplos de aplicación que utilizan la librería desarrollada.

El informe esta estructurado de la siguiente manera: en el capítulo 2 se presenta las redes de Petri junto con algunas herramientas utilizadas para el modelado y simulación en dicho formalismo, en el capítulo 3 se introduce el formalismo DEVS, la herramienta PowerDEVS y se presenta un antecedente de este trabajo, el cual fue tomado como punto de partida. En el capítulo 4 desarrollamos concretamente el trabajo realizado. Luego, en el capítulo 5 exponemos distintos ejemplos y aplicaciones que manifiestan distintas características de la implementación. Finalmente, en el capítulo 6 comentamos algunas conclusiones y posibles trabajos futuros.

Capítulo 2

El Formalismo Redes de Petri

Las redes de Petri (PN) son un lenguaje de modelado matemático de notación gráfica, creado por Carl Adam Petri, que se utiliza para representar fenómenos tales como la dinámica de eventos, la evolución en paralelo, la dependencia condicional (como la sincronización), la competencia de recursos, etc. Estos fenómenos aparecen en muchos sistemas de eventos discretos, como por ejemplo sistemas de producción, protocolos de comunicaciones, computadoras y redes de computadoras, software de tiempo real, sistemas de transporte, etc.

En este capítulo haremos un análisis general de las redes de Petri, esto es, estudiaremos los componentes que las constituyen, sus posibles aplicaciones, agregados que aumentan el poder expresivo de las mismas y finalmente daremos un ejemplo práctico para afianzar los conocimientos adquiridos. Además, veremos algunas de las herramientas para la manipulación de las redes de Petri utilizadas en la actualidad.

2.1. Componentes Básicos

Desde el punto de vista de la teoría de grafos, descrita en [6], una red de Petri es (ϑ, M) compuesto por un grafo dirigido bipartito $\vartheta = (E, V)$ y una marca inicial M. El conjunto de vértices está dividido en dos subconjuntos disjuntos P (lugares) y T (transiciones). Los lugares serán representados por: $P_i, i = 1, ..., |P|$ y las transiciones por: $T_j, j = 1, ..., |T|$. Los arcos de E tienen origen en los lugares y destino en las transiciones o bien, origen en las transiciones y destino en los lugares. Al ser el grafo bipartito los arcos nunca tendrán un mismo tipo de componente en sus extremos, es decir, no conectan transiciones con transiciones ni lugares con

lugares. Gráficamente los lugares serán representados mediante círculos y las transiciones mediante rectángulos. A los arcos se les asigna un peso el cual está dado por un número entero[1]. Para terminar de completar la definición formal de una red de Petri, debemos introducir una marca inicial. La marca inicial asigna un entero no negativo M_i a cada lugar P_i. Gráficamente, M_i puntos o marcas serán colocados en el círculo que representa a P_i. El vector columna M, cuyas componentes son las M_i, recibe el nombre de marca inicial de la red de Petri. Diremos que un lugar P_i es anterior a una transición T_j, si hay un arco que va de P_i a T_j. Análogamente, diremos que un lugar P_i es posterior a la transición T_j, si hay un arco que va de T_j a P_i.

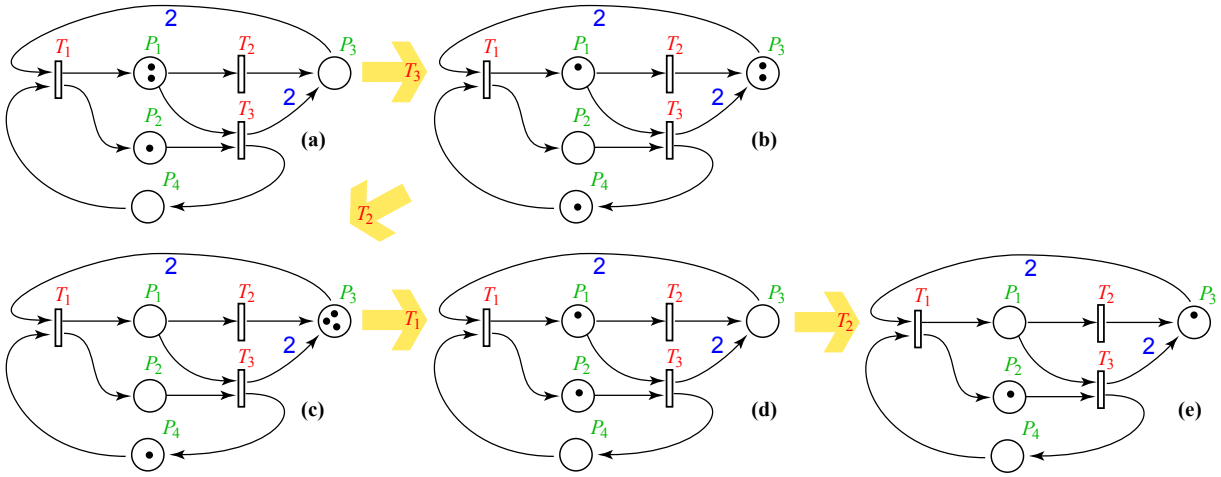

Figura 2.1: Evolución de una red de Petri

Habitualmente los lugares representan condiciones y las transiciones representan eventos. Una transición (es decir, un evento) posee un cierto número de lugares anteriores y posteriores que representan las pre-condiciones y post-condiciones de dicho evento. La presencia de una marca en un lugar (cuando el peso de todos los arcos es 1) puede interpretarse como que la condición asociada a dicho lugar se verifica. Otra interpretación posible es la siguiente: M_i marcas son colocadas en un lugar para indicar que hay M_i recursos disponibles.

Dentro del enfoque clásico de las redes de Petri, la marca de la red de Petri es identificada como su estado. Los cambios de estados ocurren según las siguientes reglas de evolución:

- una transición T_i puede dispararse o activarse (en dicho caso se dice que está habilitada) si cada lugar anterior a T_i contiene al menos tantas marcas como el peso del arco que los

[1]En ausencia de dicho número se considera que el peso es 1.

une;

- cuando una transición T_i es disparada o activada se elimina de cada lugar anterior a dicha transición tantas marcas como el peso del arco que los une. También se agrega a cada lugar posterior a T_i tantas marcas como el peso del arco que los une (ver Figura 2.1).

Observación: en lugar de asociar pesos a los arcos, podemos suponer que todos los arcos tienen peso uno; pero permitir varios arcos "paralelos" entre lugares y transiciones. En nuestro trabajo utilizaremos esta última convención (ver Figura 2.2).

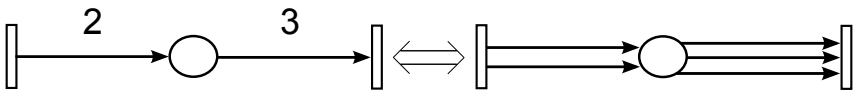

Figura 2.2: Arcos paralelos en lugar de pesos en los arcos

2.2. Semántica de las redes de Petri

Los diferentes componentes de una red de Petri tienen habitualmente las siguientes interpretaciones:

- las marcas representan recursos en el amplio sentido de la palabra. Pueden ser tanto recursos físicos, como recursos no materiales, tales como: información, mensajes, etc.;
- los lugares es donde los recursos pueden esperar o almacenarse;
- las transiciones representan acciones que consumen recursos para luego transformarlos, o bien producir nuevos recursos;
- los pesos de los arcos que van de un lugar a una transición, representan el número mínimo de recursos de la clase almacenada en dicho lugar que son necesarios para llevar a cabo la acción representada por la transición;
- los pesos de los arcos que van de una transición a un lugar representan el número de recursos de la clase que es almacenada en dicho lugar que son producidos al llevarse a cabo la acción que representa la transición;

- el número total de marcas en una red de Petri no necesariamente debe conservarse pues, por ejemplo, una transición puede representar la operación de ensamblaje de una parte compleja a partir de partes elementales, o bien inversamente, puede representar el desarme de una parte compleja en partes elementales. También los mensajes pueden combinarse para producir un nuevo mensaje (por ejemplo, sumar dos números) o un mismo mensaje puede enviarse a varios lugares.

2.3. Temporización y transiciones E/S

2.3.1. Tiempos asociados a lugares y transiciones

Un complemento que se le puede incorporar a las redes de Petri es la *temporización*. Esto puede hacerse asociándole tiempo a las transiciones y a los lugares con la siguiente semántica:

- tiempos asociados a los lugares: estos son los tiempos mínimos que las marcas deben permanecer en un lugar antes de contribuir a habilitar el disparo de una transición posterior a dicho lugar. Los mismos reciben el nombre de tiempos de espera;

- tiempos asociados a las transiciones: estos son los tiempos que separan el comienzo (el consumo de marcas de los lugares anteriores) y la finalización (la producción de marcas hacia los lugares posteriores) del disparo de una transición. Los mismos reciben el nombre de tiempos de disparo.

Los tiempos de disparo pueden utilizarse, por ejemplo, para representar tiempos de producción en el caso de los sistemas de producción (en donde las transiciones representan habitualmente a máquinas). Los tiempos de espera, pueden utilizarse para representar tiempos de transporte (es el caso en el que el lugar representa a una ruta o canal que comunica dos procesos); o bien, tiempos de almacenamiento mínimos (como por ejemplo en el caso en que una pieza debe enfriarse antes de que se pueda aplicarle el siguiente proceso).

Los tiempos de disparo y de espera pueden ser constantes en el tiempo, o pueden variar según cual sea el número de la marca en llegar a un lugar o en disparar una transición, o según el tiempo, o aleatoriamente.

2.3.2. Transiciones de entrada y salida

Las transiciones que no poseen lugares anteriores reciben el nombre de transiciones de entrada (o fuentes). Los disparos de las mismas se deben a decisiones externas (son "controladas" desde el exterior). Las transiciones que no poseen lugares posteriores se llaman transiciones de salida (o sumideros). Los disparos de las mismas nos indican cuando se producen marcas desde la red hacia el exterior.

Las mismas definiciones pueden hacerse para los lugares (los lugares de entrada deben ser provistos de marcas desde el exterior).

2.4. Ejemplo de una red de Petri

En la Figura 2.3 se ilustra un sistema *Cola-Procesador*, es decir un sistema en el cual un procesador recibe pedidos de algún usuario, los acumula en un Buffer (en caso de que se encuentre ocupado) y finalmente los procesa. El tiempo de procesamiento de cada pedido es de 2 segundos.

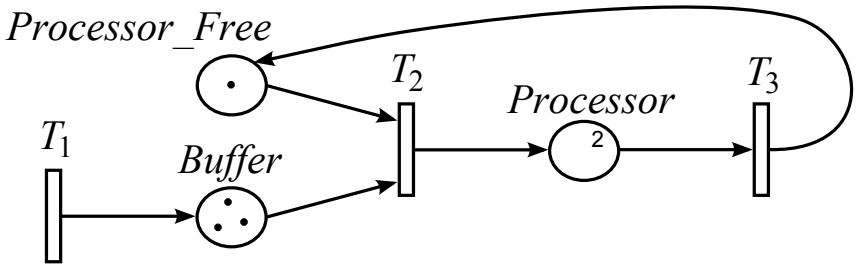

Figura 2.3: Sistema Cola-Procesador con un tiempo de dos segundos de procesamiento

En el estado inicial, el sistema tiene asignada una marca en el lugar *Processor_Free* indicando que el procesador está desocupado, y por lo tanto puede procesar un pedido, el resto de los lugares (*Buffer* y *Processor*) se encuentran sin marcas. Además, *Processor* tiene un tiempo de espera de dos unidades, representando el tiempo de procesamiento. Por otro lado, todas las transiciones son instantáneas (tienen tiempo de disparo 0).

El comportamiento del sistema puede resumirse en el siguiente ciclo:

- La transición fuente T_1 se dispara (representando el procesamiento de un pedido) y agrega una marca al Buffer;

- T_2 se activa dado que cada uno de sus lugares anteriores (*Buffer* y *Processor_Free*) contienen una marca, y envía el pedido al procesador, al cual le toma dos segundos tratarlo. Aquí, T_2 al consumir las marcas de *Buffer* y *Processor_Free* representa el comienzo del procesamiento de un pedido y el correspondiente estado "ocupado" del procesador;

- T_3 envía un evento a *Processor_Free* notificando que el procesador se encuentra libre, y puede tomar otro pedido.

El ciclo terminará cuando no haya más pedidos pendientes. Observar que Processor_Free funciona como un condicionador, es decir, mientras que no tenga una marca, el ciclo no podrá re-iniciarse.

2.5. Simuladores de redes de Petri

Dado que las PN son sencillas, fácilmente extensibles y muy útiles, en la actualidad, se ha hecho hincapié en el desarrollo de programas que permitan modelar un sistema con dicho formalismo y simularlo. Hay varias alternativas para la realización de esta tarea, a continuación expondremos algunas herramientas existentes:

- Herramientas gratuitas multiplataforma en java:

  Pipe2 [3] es a código abierto, sirve únicamente para crear y analizar redes de Petri, incluyendo redes de Petri estocásticas generalizadas (GSPN). Contiene varios módulos de análisis, incluyendo análisis avanzado de GSPN, maneja cientos de miles de estados y elimina estados que dejan de existir sobre la marcha. Además, genera grafos reutilizables y permite ver la red simulada de forma animada.

  Tapaal [4] es una herramienta para la verificación de redes de Petri con tiempos en los arcos (TAPN). Ofrece un editor gráfico para dibujar modelos TAPN, simulación para experimentar las redes diseñadas y entorno de verificación que automáticamente responde preguntas formuladas lógicamente en un subconjunto de CTL logic (esencialmente EF, EG, AF y AG). También, chequea cuando una red dada es *K-bounded*[2] para un número k dado.

[2]Una red de Petri es *K-bounded* cuando todos los lugares no pueden tener más de k marcas.

HPSim [14] es un simulador de redes de Petri que considera varias características adicionales. Una de ellas es la capacidad de los lugares. Cuando la cantidad de marcas en un lugar dado es igual a la capacidad del mismo, las transiciones que alimentan ese lugar quedan suspendidas hasta que el lugar vuelva a tener espacio disponible para recibir nuevas marcas. Otra característica adicional son los arcos inhibitorios, un arco de este tipo suspende la transición destino si el lugar de origen tiene alguna marca. En cuanto a los intervalos de ejecución de las transiciones, los mismos pueden ser: inmediatos, determinísticos, estocásticos con distribución exponencial o con distribución uniforme. Los intervalos se cuentan a partir de que la tarea está habilitada por las otras condiciones.

- Herramientas comerciales:

  Simulaworks, System Simulator [16] es un simulador de propósito general, soporta un conjunto extenso de lenguajes de simulación y técnicas de Inteligencia Artificial. Una gran cantidad de librerías están implementadas para este simulador, entre ellas una de redes de Petri.

- Librerías y Toolboxes:

  Matlab [21] posee un toolbox para diseñar, analizar y simular redes de Petri, el cual fue creado con la intención de permitir un desarrollo avanzado, incluyendo herramientas dedicadas a sistemas híbridos, ya que Matlab incorpora un completo software para el estudio de sistemas continuos y discontinuos.

  Modelica [10] cuenta con una librería de redes de Petri, que permite modelar y simular redes de Petri en las distintas herramientas de software que implementan este lenguaje. Estos modelos pueden interactuar con modelos más generales representados en Modelica. Las redes de Petri tienen la limitación de que cada lugar tenga al menos una marca.

Como podemos ver, existen multitud de herramientas (libres y comerciales) para el desarrollo y simulación de redes de Petri, sin embargo ninguna de ellas permite construir sistemas híbridos en un entorno gráfico de diagrama en bloques con la simplicidad y eficiencia que este trabajo presenta.

Capítulo 3

El Formalismo DEVS

En la década de los '70 Bernard Zeigler propuso DEVS (Discrete EVent System specification) [22], un formalismo general para representar sistemas de eventos discretos. DEVS es hoy en día una de las herramientas más utilizadas en modelado y simulación por eventos discretos, ya que ha demostrado tener no sólo una gran adaptación para la modelización de sistemas complejos, si no también simplicidad y eficiencia en la implementación de simulaciones. En este capítulo desarrollaremos algunos de los aspectos más importantes de este formalismo. Seguiremos las citas [5] y [9], a menos que se especifique lo contrario.

3.1. Conceptos previos

Para poder entender como funciona DEVS primero debemos conocer el significado de Sistema de Eventos Discretos. Los DES son aquellos cuyas variables varían o evolucionan de manera discreta en una base de tiempo en general continua. Además, necesitaremos saber como se especifica el comportamiento de un Sistema Dinámico, ya que veremos que DEVS es en realidad un caso particular de la representación de estos.

Para representar el comportamiento de los sistemas dinámicos deterministas, la Teoría de Sistemas nos proporciona la siguiente estructura:

$$S_D = (T, X, Y, \Omega, Q, \Delta, \Lambda)$$

donde:

- T es la *base de tiempo.*
- X es el conjunto de valores que puede tomar la entrada.
- Y es el conjunto de valores que puede tomar la salida.
- Ω es el conjunto de los segmentos de entrada admisibles: $\Omega : \{\omega : (t_1, t_2) \to X\}$ siendo (t_1, t_2) un intervalo en T.
- Q es el conjunto de valores de los estados.
- $\Delta : Q \times \Omega \to Q$ es la función de transición global.
- $\lambda : Q \times X \to Y$ es la función de salida.

La interpretación puede ser la siguiente: dado un segmento de trayectoria de entrada ω (dentro del conjunto de los segmentos de entrada admisibles Ω) y un valor inicial del estado (dentro del conjunto de valores de los estados Q) correspondiente al instante inicial de dicho segmento, la función de transición utiliza estos elementos como argumento y calcula el nuevo valor del estado, correspondiente al instante de tiempo del final del segmento de entrada mencionado.La salida del sistema en un instante, en tanto, es simplemente un reflejo del valor estado y de la entrada en dicho instante (a través de la función de salida por supuesto).

Esto es, supongamos que en un instante $t_0 \in T$ el estado vale $q_0 \in Q$ y en el intervalo (t_0, t_1) (con $t_1 \in T$) la entrada está dada por el segmento $\omega \in \Omega$. Luego, en el tiempo t_1 se tendrá que el estado vale $q_1 = \Delta(q_0, \omega(t_0)) \in Q$. Asimismo, la salida en el instante t_1 estará dada por $y_1 = \Lambda(q_1, \omega(t_1))$.

Ahora bien, DEVS permite representar cualquier sistema que experimente un número finito de cambios (eventos) en cualquier intervalo de tiempo, por lo tanto puede verse que DEVS es un caso específico de la representación general de sistemas dinámicos recién vista, en la cual las trayectorias de entrada estarán restringidas a *segmentos de eventos* y la función de transición tendrá una forma especial que limitará las trayectorias de salida para que tengan idéntica naturaleza. Definiremos formalmente *segmentos de eventos* de la siguiente manera:

Definición 3.1 *Segmentos de eventos*

Sea $\omega : [t_0, t_n] \rightarrow A \cup \{\phi\}$ un segmento sobre una base continua de tiempo (o sea, una función de $t \in (t_0, t_n)$, siendo este último un intervalo de los reales) y el conjunto $A \cup \{\phi\}$. Aquí ϕ es un elemento que no pertenece a A y representa la ausencia de evento.

Luego, ω es un segmento de eventos si y sólo si existe un conjunto de puntos $t_1, t_2, \ldots, t_{n-1}$ con $t_i \in (t_0, t_n)$ tales que $\omega(t_i) \in A$ para $i = 1, \ldots, n-1$ y $\omega(t) = \{\phi\}$ para todo otro punto en (t_0, t_n).

La Figura 3.1 ilustra una trayectoria de eventos, en la cual los valores a_i pertenecen al conjunto A.

Más allá de esta definición formal, un evento es la representación de un cambio instantáneo en alguna parte de un sistema. El mismo puede caracterizarse por un valor y un instante en el que ocurre. El valor puede ser un número, un vector, una palabra o, en general, un elemento cualquiera de un conjunto determinado (A en nuestra definición).

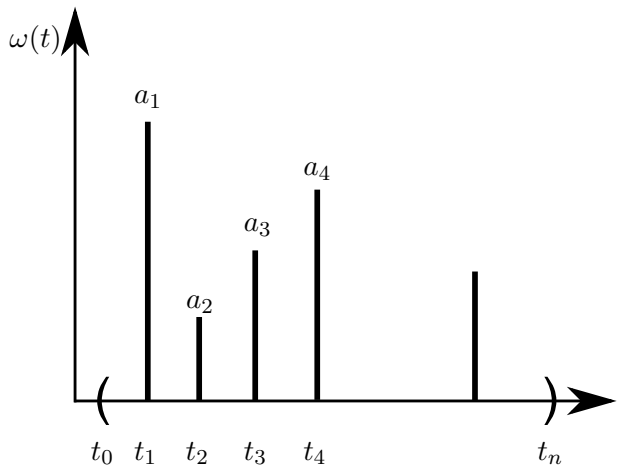

Figura 3.1: Trayectoria de Eventos

3.2. Modelos DEVS Atómicos

Un modelo DEVS procesa una trayectoria de eventos de entrada y, según esta trayectoria y sus propias condiciones iniciales, produce una trayectoria de eventos de salida. Este comportamiento entrada/salida se representa en la Figura 3.2.

Figura 3.2: Comportamiento Entrada/Salida de un modelo DEVS

Definiremos a un modelo DEVS atómico con la siguiente estructura:

$$M = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$$

donde:

- X es el conjunto de valores de eventos de entrada, es decir, el conjunto de todos los valores que un evento de entrada puede adoptar.
- Y es el conjunto de valores de eventos de salida.
- S es el conjunto de valores de estado.
- δ_{int}, δ_{ext}, λ y ta son funciones que definen la dinámica del sistema.

Cada posible estado s ($s \in S$) tiene asociado un *Avance de Tiempo* calculado por la *Función de Avance de Tiempo* (Time Advance Function) $ta(s)(ta : S \to \mathbb{R}_0^+)$. El *Avance de Tiempo* es un número real no negativo que indica cuanto tiempo el sistema permanecerá en un estado determinado en ausencia de eventos de entrada. Luego, si el estado toma el valor s_1 en el tiempo t_1, tras $ta(s_1)$ unidades de tiempo (o sea, en tiempo $ta(s_1)+t_1$) el sistema realiza una transición interna yendo a un nuevo estado s_2. El nuevo estado se calcula como $s_2 = \delta_{int}(s_1)$. La función δ_{int} ($\delta_{int} : S \to S$) se llama *Función de Transición Interna* (Internal Transition Function). Cuando el estado va de s_1 a s_2 se produce también un evento de salida con valor $y_1 = \lambda(s_1)$. La función λ ($\lambda : S \to Y$) se llama *Función de Salida* (Output Function). Así, las funciones ta, δ_{int} y λ definen el comportamiento autónomo de un modelo DEVS. Cuando llega un evento de entrada el estado cambia instantáneamente. El nuevo valor del estado depende no sólo del valor del evento de entrada sino también del valor anterior de estado y del tiempo transcurrido desde la última

transición. Si el sistema llega al estado s_3 en el instante t_3 y luego llega un evento de entrada en el instante $t_3 + e$ con un valor x_1, el nuevo estado se calcula como $s_4 = \delta_{ext}(s_3, e, x_1)$ (notar que $ta(s_3) > e$). En este caso se dice que el sistema realiza una transición externa. La función δ_{ext} ($\delta_{ext} : S \times \mathbb{R}_0^+ \times X \to S$) se llama *Función de Transición Externa* (External Transition Function). Durante una transición externa no se produce ningún evento de salida. La Figura 3.3 ilustra trayectorias típicas de un modelo DEVS. En la misma puede observarse que el sistema está originalmente en un estado s_1.

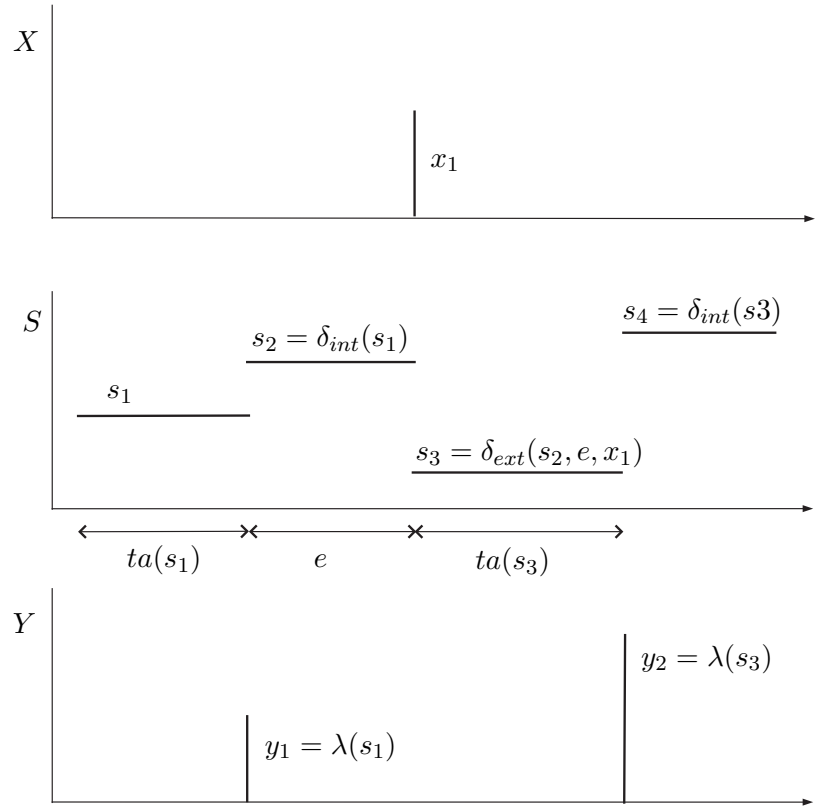

Figura 3.3: Comportamiento de un modelo DEVS

Transcurrido el tiempo $ta(s_1)$, dado que no hubo ningún evento de entrada, el sistema realiza una transición interna. Tras esta transición el estado toma el valor $s_2 = \delta_{int}(s_1)$. En el mismo instante, se produce un evento de salida $y_1 = \lambda(s_1)$. Luego, antes que transcurra el nuevo tiempo de avance $ta(s_2)$ el sistema recibe un evento de entrada con valor x_1. Entonces, se produce una transición externa y el estado adopta el nuevo valor $s_3 = \delta_{ext}(s_1, e, x_1)$ siendo e el tiempo

transcurrido desde el evento anterior (cuando el estado pasó a s_2) hasta el momento en que llega el evento. Transcurrido el tiempo $ta(s_3)$ y sin que haya habido ningún evento de entrada en el interín, el sistema realiza una nueva transición interna yendo al estado $s_4 = \delta_{int}(s_3)$ y provocando el evento de salida $y_2 = \lambda(s_3)$.

Observar que en el modelo DEVS M, el estado puede tener un avance de tiempo igual a ∞. Cuando esto ocurre, decimos que el sistema está en un estado pasivo, ya que no ocurrirá ningún cambio hasta que se reciba un nuevo evento.

3.3. Modelos DEVS Acoplados

Al momento de describir el comportamiento de un sistema complejo, se torna difícil poder plasmar todo su funcionamiento en un único modelo, para abordar esta tarea describiremos el comportamiento de varios componentes elementales y luego especificaremos como interactúan entre sí (*Acoplamiento*). En DEVS, existen dos tipos de acoplamiento: *Acoplamiento Modular* y *Acoplamiento no Modular*. En el primero, la interacción entre componentes será únicamente a través de las entradas y salidas de los mismos, mientras que en el segundo, la interacción se producirá a través de los estados.

Trabajaremos exclusivamente con el Acoplamiento Modular, puesto que en general es mucho más sencillo y adecuado, gracias a que puede aislarse y analizarse cada componente independientemente del contexto en que se encuentre. Dentro de este tipo distinguiremos dos casos: Acoplamiento mediante interfaces de traducción y acoplamiento mediante puertos, siendo esta última en realidad un caso particular de la primera.

Una de las propiedades del acoplamiento modular DEVS que no podemos dejar de tener en cuenta es la *clausura*. El cumplimiento de esta propiedad garantiza que el acoplamiento de modelos DEVS define un nuevo modelo DEVS equivalente.

Esto implica que un modelo DEVS acoplado puede utilizarse a su vez dentro de un modelo más complejo de acoplamiento, dando paso a lo que se denomina *acoplamiento jerárquico*.

La posibilidad de acoplar modelos de manera jerárquica es lo que garantiza la "reusabilidad" de los mismos. Un modelo (posiblemente acoplado) realizado como parte de un modelo más general, puede utilizarse en el contexto de otro modelo acoplado sin necesidad de realizar modificaciones.

3.3.1. Acoplamiento DEVS Básico

La Figura 3.4 muestra un modelo DEVS acoplado N. En la misma, se observan dos modelos atómicos, a y b y cinco funciones de traducción.

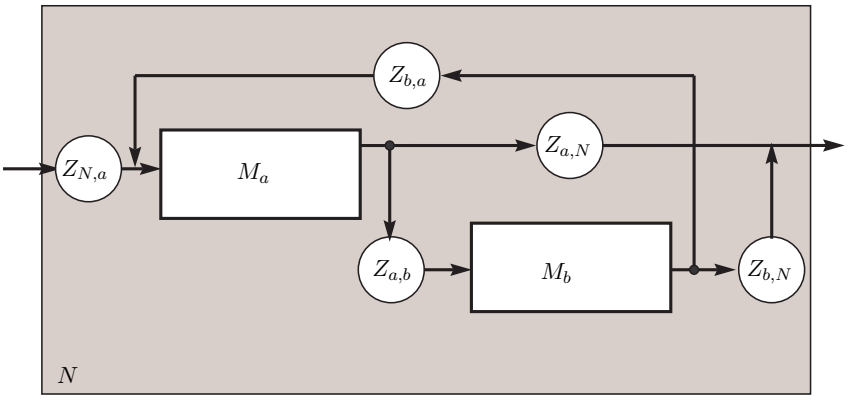

Figura 3.4: Modelo DEVS acoplado

La función de traducción $Z_{a,b}$ transforma las salidas del modelo a en entradas del modelo b. Asumiendo que:

$$M_a = (X_a, Y_a, S_a, \delta_{int_a}, \delta_{ext_a}, \lambda_a, ta_a)$$

y que:

$$M_b = (X_b, Y_b, S_b, \delta_{int_b}, \delta_{ext_b}, \lambda_b, ta_b)$$

La función de traducción $Z_{a,b}$ deberá tener dominio en Y_a e imagen en X_b.

Si además llamamos X_N al conjunto de valores de entrada en el modelo acoplado N y llamamos Y_N al conjunto de valores de salida en N, el resto de las funciones de traducción serán:

$$Z_{b,a} : Y_b \to X_a$$

$$Z_{N,a} : X_N \to X_a$$

$$Z_{a,N} : Y_a \to Y_N$$

$$Z_{b,N} : Y_b \to Y_N$$

Especificando todas estas funciones de traducción quedará entonces completamente determinado el funcionamiento del modelo N.

Formalmente, un modelo acoplado N cualquiera queda definido por la estructura:

$$N = (X_N, Y_N, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, Select)$$

donde:

- X_N es el conjunto de valores de entrada del sistema acoplado.
- Y_N es el conjunto de valores de salida del sistema acoplado.
- D es el conjunto de referencias a los componentes.
- Para cada $d \in D$, M_d es una estructura que define un modelo DEVS.
- Para cada $d \in D \cup N$, I_d es el conjunto de modelos que influyen sobre el modelo d. De acuerdo a esto, deberá ser $I_d \subseteq D \cup N$ pero con la restricción que $d \notin I_d$, esto es, un modelo no puede ser influyente de sí mismo, o sea, están prohibidas las conexiones desde un componente hacia si mismo así como una conexión directa desde la entrada hasta la salida del sistema acoplado.
- Para cada $i \in I_d$, $Z_{i,d}$ es la función de traducción donde:

  $Z_{i,d} : X_N \to X_d \quad si \quad i = N$
  $Z_{i,d} : Y_i \to Y_N \quad si \quad d = N$
  $Z_{i,d} : Y_i \to X_d \quad$ en otro caso

- $Select$ es una función, $Select : 2^D \to D$ que además verifica que $Select(E) \in E$. Esta cumple un papel de *desempate* para el caso de eventos simultáneos.

En el ejemplo de la Figura 3.4 ya dijimos como eran las funciones de traducción. Los conjuntos en tanto serán:

$$D = \{a, b\}$$
$$\{M_d\} = \{M_a, M_b\}$$
$$I_a = \{N, b\}, \qquad I_b = \{a\}, \qquad I_N = \{a, b\}$$

La interpretación funcional del acoplamiento es la siguiente: Cada modelo DEVS funciona independientemente desde el punto de vista interno, pero recibe eventos de entrada provenientes de las salidas de otros componentes (o de la entrada global del sistema). Estos componentes que pueden provocarle eventos de entrada a un determinado componente se denominan *influyentes*. Generalmente los eventos que puede recibir un componente no coinciden en el tipo con los que pueden generar sus influyentes. Para adaptar estas conexiones entonces se utilizan las *funciones de traducción*.

El único punto posible de conflicto es cuando dos o más componentes tienen prevista una transición interna para el mismo instante de tiempo. En este caso, la elección de uno u otro para realizar primero la transición en general modificará sustancialmente el funcionamiento del sistema ya que las transiciones internas provocan eventos de salida que a su vez provocan transiciones externas en los demás componentes. Evidentemente, el orden en que se produzcan dichas transiciones puede alterar el funcionamiento global del sistema.

La solución de este conflicto es efectuada a través de la función *Select*, que establece prioridades para las transiciones internas de diferentes componentes. Cuando un determinado subconjunto de componentes tiene agendada su transición interna de manera simultánea, la función *Select* aplicada a dicho subconjunto devuelve un elemento del mismo que será quien transicione en primer lugar.

3.3.2. Acoplamiento DEVS con Puertos

La introducción de puertos de entrada y salida, puede simplificar bastante la tarea de modelado, especialmente en lo que se refiere al acoplamiento de modelos.

Por un lado, la mayor parte de los dispositivos o procesos que se modelan con DEVS tienen una separación física o lógica intrínseca entre los diferentes tipos de entradas que pueden recibir o de salidas que pueden emitir. De esta forma, es natural representar que dichos eventos de entrada y de salida ocurren en diferentes puertos.

Por otro lado, es bastante engorroso trabajar con funciones de traducción ya que, salvo en el caso de dispositivos que contienen interfaces reales, su presencia es bastante artificial.

En consecuencia a esto, la mayor parte de las herramientas de software de simulación con DEVS utilizan acoplamiento mediante puertos de entrada y salida reemplazando así el uso de las funciones de traducción.

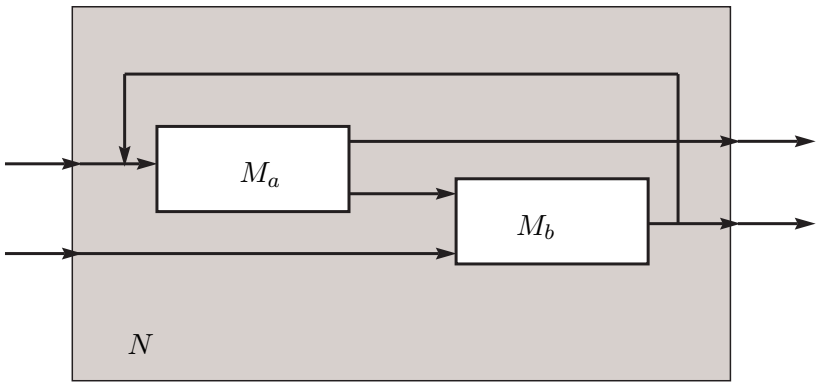

Figura 3.5: Modelo DEVS acoplado con puertos

En el formalismo DEVS, es posible introducir el concepto de puertos particularizando los conjuntos X e Y (conjuntos de valores de entrada y salida respectivamente) de los modelos atómicos. De esta manera, puede verse que el acoplamiento con puertos es un caso especial del acoplamiento visto en la sección 3.3.1, y por lo tanto también cumple la propiedad de clausura, lo que nos permitirá realizar acoplamiento jerárquico mediante puertos.

El conjunto de entrada de un modelo atómico con puertos deberá ser de la forma:

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

siendo $InPorts$ el conjunto de puertos de entrada y X_p el conjunto de entrada en el puerto p.

Similarmente, el conjunto de valores de salida será:

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

con definiciones análogas.

El acoplamiento de modelos con puertos en tanto se especifica mediante la siguiente estructura:

$$N = (X_N, Y_N, D, \{M_d\}, EIC, EOC, IC, Select)$$

donde:

- El conjunto de valores de entrada del modelo acoplado es $X_N = \{(p, v)|\ p \in Inports, v \in X_p\}$ siendo $InPorts$ el conjunto de puertos de entrada y X_p el conjunto de valores de entrada en el puerto p.

- El conjunto de valores de salida del modelo acoplado es $Y_N = \{(p, v)|\ p \in OutPorts, v \in Y_p\}$ siendo $OutPorts$ el conjunto de puertos de entrada e Y_p el conjunto de valores de salida en el puerto p.
- D es el conjunto de referencias a componentes.
- Para cada $d \in D, M_d = (X_d, Y_d, S_d, \delta_{int_d}, \delta_{ext_d}, \lambda_d, ta_d)$ donde:
  - $X_d = \{(p, v)|\ p \in InPorts_d, v \in X_{p_d}\}$
  - $Y_d = \{(p, v)|\ p \in OutPorts_d, v \in Y_{p_d}\}$
- El acoplamiento externo de entrada EIC (*external input coupling*) conecta las entradas externas con las entradas de los componentes:

$$EIC \subseteq \left\{((N, ip_N), (d, ip_d))|\ ip_N \in InPorts, d \in D, ip_d \in InPorts_d\right\}$$

- El acoplamiento externo de entrada EOC (*external output coupling*) conecta las salidas de los componentes con las salidas externas:

$$EOC \subseteq \left\{((d, op_d), (N, op_N))|\ op_N \in OutPorts, d \in D, op_d \in OutPorts_d\right\}$$

- El acoplamiento interno IC (internal coupling) conecta las salidas y las entradas de los componentes:

$$IC \subseteq \left\{((a, ip_a), (b, ip_b))|\ a, b \in D, ip_a \in InPorts_a, ip_b \in InPorts_b\right\}$$

  Nuevamente, no se permite *feedback directo* ($a \neq b$ en la definición anterior).
- *Select* se define igual que antes.

3.4. Simulación de modelos DEVS

Unas de las cualidades más importantes de un modelo DEVS son la sencillez y eficiencia con que puede ser simulado. La simulación es una herramienta muy útil a la hora de comprobar la detección temprana de errores en un modelo, nos permite ver como un sistema se comportaría de

acuerdo a una configuración dada. Al tener un seguimiento paso a paso de los distintos procesos que se van realizando, se puede no sólo detectar posibles errores si no también incorporar nuevas mejoras, logrando así un sistema más correcto y eficiente.

Ilustraremos, sin entrar en detalles, mediante los siguientes pasos la idea básica para simular un modelo DEVS acoplado.

1. Buscar el modelo atómico d^* que, de acuerdo a su tiempo de avance y tiempo transcurrido, sea el próximo en realizar una transición interna.

2. Sea tn el tiempo de la transición mencionada. Avanzar entonces el tiempo de la simulación t hasta $t = tn$ y ejecutar la función de transición interna de d^*.

3. Propagar el evento de salida producido por d^* hacia todos los modelos atómicos conectados a él ejecutando las transiciones externas correspondientes. Luego, volver al paso 1.

3.4.1. Simulación Jerárquica

Una de las formas más simples de implementar estos pasos es escribiendo un programa con una estructura jerárquica equivalente a la estructura jerárquica del modelo a simular. Esto es, asociar a un modelo atómico una rutina *Simulator* y una rutina *Coordinator* a los modelos acoplados. En la cima de esta estructura jerárquica se coloca otra rutina denominada *Root-Coordinator*, que se encarga de avanzar el tiempo global de la simulación. Los simuladores y coordinadores de capas consecutivas se comunican a través de mensajes. Los coordinadores envían mensajes a sus hijos para que ejecuten las funciones de transición. Cuando un simulador ejecuta una transición, calcula su próximo estado y cuando la transición es interna envía el valor de salida a su coordinador padre. En todos los casos, el estado del simulador coincidirá con el estado de su modelo DEVS atómico asociado. Cuando un coordinador ejecuta una transición, envía mensajes a algunos de sus hijos para que ejecuten sus funciones de transición correspondientes. Cuando un evento de salida producido por uno de sus hijos debe ser propagado fuera del modelo acoplado, el coordinador envía un mensaje a su propio coordinador padre con el valor de salida correspondiente. Cada simulador o coordinador tiene una variable local tn que indica el tiempo en el que ocurrirá su próxima transición interna. En los simuladores, esa variable se calcula utilizando la función de avance de tiempo del modelo atómico correspondiente. En los coordinadores, la misma se calcula como el mínimo tn de sus hijos. Luego, el tn del coordinador

que está por encima de todos (*Root-Coordinator*) es el tiempo en el cual ocurrirá el próximo evento considerando el sistema completo. Finalmente, el Coordinador Raíz avanza el tiempo global t hasta el valor tn y luego envía un mensaje a su hijo para que realice la siguiente transición. Este ciclo se repetirá hasta que la simulación termine.

3.4.2. Simulación Plana

Un problema de la estructura jerárquica que no podemos dejar de tener en cuenta, es que puede darse un importante tráfico de mensajes entre las capas inferiores y las capas superiores. Todos estos mensajes y su tiempo computacional correspondiente pueden evitarse con el uso de una estructura de simulación plana, en la cual todos los simuladores están a un mismo nivel jerárquico bajo un único coordinador. La forma de transformar una simulación jerárquica en una plana es muy simple en el contexto de DEVS y agrega mucha eficiencia. De hecho, la mayor parte de las herramientas de software mencionadas implementan la simulación en base a un código plano.

3.5. Herramientas de simulación basadas en DEVS

En la actualidad, existe una gran cantidad de herramientas basadas en DEVS [2, 1], esto se debe al gran potencial que este presenta para la formalización de DES. Algunas de las aplicaciones que podemos encontrar son:

ADEVS [11] es una librería de C++ para desarrollar simulación de modelos basados en en DEVS. Utiliza dos extensiones del formalismo DEVS llamadas *Parallel DEVS* y *Dynamic DEVS*, que tratan la ocurrencia de eventos simultáneos utilizando una función de transición *confluente*[1]. ADEVS no es un software de simulación completo, no posee por ejemplo una interfaz gráfica para realizar los acoplamientos de los modelos atómicos, estos deben ser acoplados mediante código. Tampoco cuenta con una forma de visualización de los resultados de la simulación.

DEVSJava [7, 22] es una herramienta para la simulación de modelos DEVS desarrollada por Bernard P. Zeigler y Hessam S. Sarjoughian. Permite crear modelos DEVS escalables,

[1] PowerDEVS utiliza una función *desempate* para tratar a los eventos simultáneos.

esto es, ofrece una vista de la estructura jerárquica del modelo DEVS incluyendo modelos atómicos, acoplados, puertos y conexiones. Además, admite la posibilidad de modificar la estructura del modelo en tiempo de ejecución (o tiempo de simulación). Incluye la posibilidad de simular los modelos en tiempo real, pero lo hace de una manera *best-shot*, ya que al estar desarrollado en JAVA y correr sobre sistemas operativos que no son de tiempo real, no puede ofrecer ninguna garantía sobre la precisión de la simulación.

CD++ [19] es una herramienta de simulación basada en el formalismo *Cell-DEVS*, que permite simular modelos complejos programando su comportamiento. Permite la construcción de sistemas en forma jerárquica y presta la posibilidad, a diferencia de los dos softwares anteriores, de utilizar el formalismo redes de Petri para la representación de sistemas donde su uso sea relevante. En la sección 3.7 veremos con más detalles esta aplicación.

Si bien existen varias herramientas basadas en DEVS, no todas cuentan con la implementación de redes de Petri. Por otro lado, las herramientas que sí implementan dicho formalismo, como CD++, presentan algunas restricciones al momento de su aplicación. En la siguiente sección se presenta la herramienta *PowerDEVS*, sobre la que se desarrolló el formalismo redes de Petri.

3.6. PowerDEVS

PowerDEVS es un entorno de simulación multi-plataforma de modelos DEVS de propósito general. Fue desarrollado y es mantenido por el Laboratorio de Sistemas Dinámicos de la Facultad de Ciencias Exactas, Ingeniería y Agrimensura de la Universidad Nacional de Rosario.

PowerDEVS permite definir modelos DEVS atómicos en el lenguaje C++, los cuales pueden ser acoplados gráficamente en un diagrama de bloques jerárquico para crear sistemas más complejos. Tanto los modelos atómicos como los acoplados pueden ser organizados en librerías para poder ser reutilizados. En esta tesina se ha incorporado una librería llamada *PetriNets* a PowerDEVS, la cual permite modelizar sistemas con redes de Petri.

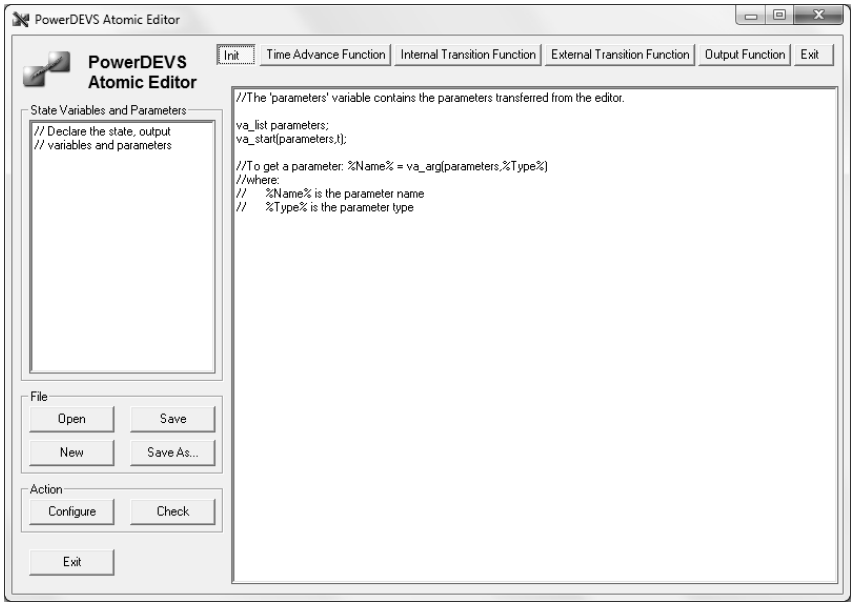

Figura 3.6: Edición de un Modelo atómico de PowerDevs

3.6.1. Introducción a PowerDEVS

Una de las característica de este software que cabe destacar es la posibilidad de realizar la simulación de un sistema en tiempo real. A diferencia de la estructura de simulación presentada en la sección 3.4.1, en PowerDEVS el esquema de mensajes está invertido en lo que refiere al tiempo de próximo evento. Aquí los modelos hijos tienen posibilidad de cambiar asincrónicamente su tiempo de avance, notificando a su padre (recursivamente). De esta forma, es posible detectar y tratar interrupciones en tiempo real a nivel de los modelos atómicos.

Otra característica importante al momento de trabajar con PowerDEVS es que ofrece un entorno de simulación DEVS apropiado para sistemas *híbridos* (sistemas continuos y discretos), dado que cuenta con librerías de métodos numéricos de integración y una interfaz gráfica de diagramas de bloques como la de *Simulink*[2], donde las definiciones de los modelos permanecen escondidas para los usuarios que no dominan DEVS. Este entorno permite que los usuarios puedan aprovechar métodos basado en DEVS como por ejemplo QSS y QSS2[3], sin tener que estudiar dicho formalismo.

[2] *Simulink* es un entorno de programación visual, que funciona sobre el entorno de programación *Matlab*.

[3] QSS y QSS2 realizan una aproximación DEVS de sistemas continuos. De esta manera, PowerDEVS puede simular sistemas híbridos de una manera muy simple y eficiente.

Ahora bien, desde el punto de vista implementación, PowerDEVS está dividido en dos módulos bien heterogéneos.

Entorno de desarrollo gráfico: Es la interfaz que ve el usuario regular de PowerDEVS. Permite describir modelos DEVS de una forma gráfica y sencilla. Este módulo es el encargado de generar el código C++ que luego será compilado y ejecutado para simular el modelo DEVS correspondiente.

Motor de simulación: Este módulo es el núcleo de toda la herramienta. El código que es generado por el entorno gráfico, luego es compilado junto con el motor de simulación y se obtiene el programa que simula el modelo DEVS. Su implementación está hecha en C++ por lo cual puede ser portado fácilmente a cualquier arquitectura y sistema operativo. Básicamente es una implementación del método de simulación de modelos DEVS descrito en [17] que veremos en la sección 3.6.3.

3.6.2. PowerDEVS y redes de Petri

Esta herramienta permite acoplar modelos gráficamente a través de una interfaz sencilla e intuitiva lo que la hace apropiada para el modelado con formalismos gráficos como las redes de Petri.

El motor de simulación de PowerDEVS, implementado completamente en C++, mapea cada modelo DEVS atómico a un clase. Al ejecutarse la simulación, habrá tantas instancias de esta clase como modelos atómicos hubiere. Esto permite referenciar a través de punteros a los modelos atómicos en tiempo de ejecución. Esta característica, como se verá más adelante, será crucial para evitar el agregado de las conexiones y puertos inexistentes que aparecen en [8].

Por otra parte, los eventos que procesan los modelos atómicos también son instancias de una clase llamada `Event`. Como ya vimos tanto los eventos de entrada como los de salida de un modelo DEVS pueden pertenecer a cualquier conjunto definido por el modelador. Para representar esto en el motor de simulación, el valor que contiene la clase `Event` es del tipo `void *`, o sea un puntero genérico. C++ hace posible que cualquier tipo sea convertido desde y hacia `void *` dejando así que los eventos tomen valores de cualquier tipo.

Además, PowerDEVS contiene una utilidad que permite especificar un orden de prioridades a los distintos modelos para determinar cuál debe ejecutarse en caso de que dos o más

puedan transicionar al mismo tiempo, llamada *Select Priority*. Está herramienta será de vital importancia para nuestro desarrollo como veremos en la sección 4.2.

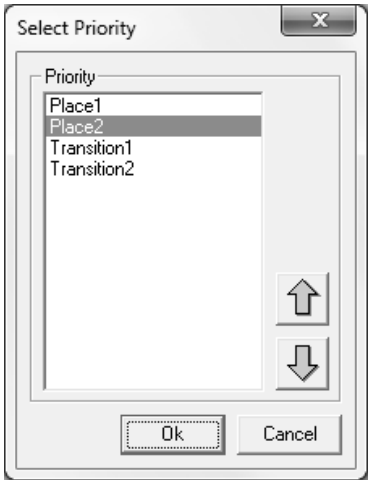

Figura 3.7: Herramienta de PowerDEVS para el manejo de prioridades

3.6.3. Pseudo-código para la simulación de DEVS

Los siguientes pseudo-códigos describen las tres clases mencionadas en la sección 3.4.1. Además, así es como está implementada la simulación en PowerDEVS.

- DEVS-simulator:

  $DEVS - simulator$

  $variables:$

  tl// $time\ of\ last\ event$

  tn// $time\ of\ next\ event$

  e// $elapsed\ time\ in\ the\ actual\ state$

  s// $actual\ state\ of\ the\ atomic$

  $y = (y.value, y.port)$// $current\ output\ of\ the\ atomic$

  $when\ recieves\ i - message(i,t)\ at\ time\ t$

  $tl = t - e$

  $tn = tl + ta(s)$

  $when\ recieve\ *-message(*,t)\ at\ time\ t$

$y = \lambda(s)$

$send\ y - message(y,t)\ to\ parent\ coordinator$

$s = \delta_{int}(s)$

$tl = t$

$tn = t + ta(s)$

$when\ recieve\ x - message(x,t)\ at\ time\ t$

$e = t - tl$

$s = \lambda_{ext}(s,e,x)$

$tl = t$

$tn = t + ta(s)$

$end\ DEVS\ simulator$

- DEVS-coordinator:

$DEVS - coordinator$

$variables:$

$tl//\ time\ of\ last\ event$

$tn//\ time\ of\ next\ event$

$y = (y.value, y.port)//\ current\ output\ of\ the\ atomic$

$D//\ list\ of\ childern$

$IC//\ list\ of\ connections\ of\ the\ form\ [(d_i, port_1), (d_j, port_2)]$

$EIC//\ list\ of\ connections\ of\ the\ form\ [(N, port_1), (d_j, port_2)]$

$EOC//\ list\ of\ connections\ of\ the\ form\ [(d_i, port_1), (N, port_2)]$

$when\ recieve\ i - message(i,t)\ at\ time\ t$

$send\ i - message(i,t)\ to\ all\ children \in D$

$when\ recieve\ *-message(*,t)\ at\ time\ t$

$send\ *-message(*,t)\ to\ d^*$

$d^* = arg[min_{d \in D}(d.tn)]$

$tl = t$

$tn = t + d^*.tn$

$when\ recieve\ x - message((x.value, x.port), t)\ as\ time$

$(v,p) = (x.value, x.value)$

$for\ each\ connection\ [(N,p),(d,q)]$

$send\ x - message((v, q), t)\ to\ child\ d$

$d^* = arg[min_{d \in D}(d.tn)]$

$tl = tn$

$tn = t + d^*.tn$

$when\ recieve\ y - message((y.value, y.port), t)\ from\ d^*$

$if\ a\ connection\ [(d^*, y.port), (N, q)]\ exists$

$send\ y - message((y.value, q), t)\ to\ parent\ coordinator$

$foreach\ connection[(d^*, p), (d, q)]$

$send\ x - message((y.value, q), t)\ to\ child\ d$

$end\ DEVS - coordinator$

- DEVS-root-coordinator:

$DEVS - root - coordinator$

$variables:$

$t//\ global\ simulation\ time$

$d//\ child\ (coordinator\ or\ simulator)$

$t = t_0$

$send\ i - message(i, t)\ to\ d$

$t = d.tn$

$loop$

$send\ * - message(*, t)\ to\ d$

$s = \delta_{int}(s)$

$t = d.tn$

$until\ end\ of\ simulation$

$end\ DEVS - root - coordinator$

En PowerDEVS cada modelo es un objeto de C++ propiamente dicho, los métodos de la clase se muestran en el Código 3.1.

En la siguiente sección se analiza el software *CD++*, en particular, los modelos utilizados para la representación de los principales componentes de las redes de Petri. Cabe destacar que dichos modelos fueron el punto de partida de nuestro trabajo en PowerDEVS. Esto es, estudiamos su estructura y funcionamiento, intentando no sólo extender nuestro software, si no también aplicar nuevas mejoras con el fin de tener un mayor rendimiento.

Código 3.1 Métodos de un modelo atómico en PowerDEVS.

```
void init(double, ...);
double ta(double t);
void dint(double);
void dext(Event , double );
Event lambda(double);
void exit();
```

3.7. CD++

Al igual que PowerDEVS, *CD++* [8] es una herramienta que permite a los usuarios implementar modelos DEVS. Los modelos atómicos pueden ser programados e incorporados a una clase básica jerárquica programada en C++. El software también habilita a los usuarios construir modelos usando una notación basada en grafos, la cual logra una visualización más abstracta del problema. Cada grafo define los cambios de estado de acuerdo a las funciones de transición externa e interna, y cada uno es traducido a una definición analítica.

Además, CD++ cuenta con una librería que permite modelar sitemas con redes de Petri. En lo que sigue, estudiaremos los dos principales componentes del formalismo (Place y Transition), es decir, expondremos la definición de cada uno de los modelos DEVS que los representan.

3.7.1. Redes de Petri en CD++

Lugar (Place)

La Figura 3.8 ilustra el modelo DEVS que describe un lugar de una red de Petri. Este tiene un sólo puerto de entrada y un sólo puerto de salida.

Figura 3.8: Modelo conceptual de un lugar

El puerto de entrada (IN) es usado para recibir marcas desde cero o más transiciones. También es usado para notificar al lugar que debe descontar marcas cuando una transición es disparada. Los mensajes recibidos por este puerto contienen información sobre el número de

marcas y la operación (adición o sustracción) que debe realizarse.

El puerto de salida (OUT) es usado para notificar a las transiciones el número de marcas que contiene, de esta manera las transiciones conectadas a él pueden determinar si están activas. Este proceso es ejecutado cada vez que el número de marcas en el lugar es modificado y cuando el modelo es inicializado al comienzo de la simulación.

La especificación formal para este modelo es:

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, D, \lambda)$$

donde:

- $X = \{IN \in \mathbb{N}^+\}$
- $Y = \{OUT \in \mathbb{N}^+\}$
- $S = \{\{tokens \in \mathbb{N}_0^+\} \cup \{id \in \mathbb{N}^+\} \cup \{phase \in \{active, passive\}\}\}$ donde $tokens$ es el número de marcas contenidos en el lugar, el id es el identificador de el lugar asignado por el simulador DEVS y $phase$ el comportamiento que debe tomar el lugar, $active$ notifica el número de marcas que contiene y $passive$ espera la próxima transición externa.
- $\delta_{ext}(s, e, x)$ {
  $retrieve\ id\ and\ number\ of\ tokens\ from\ message$
  $case\ id = 0\ /*\ generic\ message\ */$
  $increment\ tokens$
  $hold\ in\ active\ 0\ /*\ to\ advertise\ the\ number\ of\ tokens\ */$
  $case\ id\ \neq 0\ /*\ specific\ message\ */$
  $id\ matches\ id\ of\ this\ place?$
  $no:\ disregard\ the\ message$
  $yes:\ decrement\ tokens\ by\ the\ number\ of\ tokens\ specified\ if\ there\ are$
  $enough.\ Otherwise\ throw\ an\ exception.$
  $hold\ in\ active\ 0\ /*\ to\ advertise\ the\ number\ of\ tokens\ */$
  } $end\ of\ external\ transition\ function$
- $\delta_{int}(s)$ {
  $passivate\ /*\ wait\ for\ the\ next\ external\ event\ */$

}

- $\lambda(s)$ {
  combine id and tokens state variables in one message and send on the out port.
  }

Transición (Transition)

La Figura 3.9 ilustra el modelo DEVS que describe una transición de una red de Petri. Esta tiene cinco puertos de entrada y cinco puertos de salida:

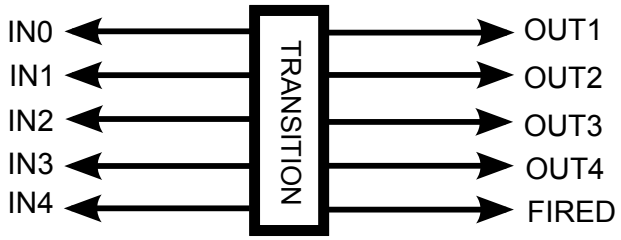

Figura 3.9: Modelo conceptual de una transición

$IN1$ es usado para notificar la cantidad de *tokens* que tienen los lugares anteriores. Los lugares conectados a este puerto están conectados con un sólo arco. En otras palabras, si la transición se dispara, sólo un *token* será removido de los lugares anteriores;

$IN2$ tiene la misma función que el puerto $IN1$ excepto que el arco representa una conexión doble;

$IN3$ tiene la misma función que el puerto $IN1$ excepto que el arco representa una conexión triple;

$IN4$ tiene la misma función que el puerto $IN1$ excepto que el arco representa una conexión cuádruple;

$IN0$ cumple la misma función que el puerto $IN1$ con la excepción de que las conexiones consisten en un arco inhibidor. Es decir, los lugares de entrada deben contener cero *tokens* para que la transición este activa y cuando esta se dispare ningún *token* es removido del lugar;

$OUT1$ es usado para transferir un *token* a los lugares posteriores que estén conectados con su puerto de entrada a este puerto. El *id* de los mensajes enviados por este puerto es siempre cero, ya que todos los lugares que reciban este evento actualizaran su número de *tokens*;

$OUT2$ cumple con el mismo propósito que el puerto $OUT1$, excepto que el arco representa una conexión doble;

$OUT3$ cumple con el mismo propósito que el puerto $OUT1$, excepto que el arco representa una conexión triple;

$OUT4$ cumple con el mismo propósito que el puerto $OUT1$, excepto que el arco representa una conexión cuádruple;

$FIRED$ o puerto de disparo, es un puerto de salida que se utiliza para remover *tokens* de los lugares anteriores, estos deben tener sus puertos de entrada conectados a este puerto.

La especificación formal del modelo es:

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, D, \lambda)$$

donde:

- $X = \{IN0 \in \mathbb{N}^+, IN1 \in \mathbb{N}^+, IN2 \in \mathbb{N}^+, IN3 \in \mathbb{N}^+, IN4 \in \mathbb{N}^+\}$
- $Y = \{OUT1 = 1, OUT2 = 2, OUT3 = 3, OUT4 = 4, FIRED \in N^+\}$
- $S = \{\{inputs \in \mathbb{N}_0^+\} \cup \{enabled \in bool\}\}$ donde *inputs* es la cantidad de lugares anteriores y *enabled* indica si la transición esta activa o no. Notar que el modelo usa una base de datos para almacenar el *id* de los lugares anteriores, la cantidad de *tokens* que estos contienen y el peso de los arcos que conectan los lugares a la transición. Notar que esta información no define el estado del modelo de la transición, por lo tanto no esta incluida en la definición más arriba de S.
- $\delta_{ext}(s, e, x)$ {
  case port
  $in0$: *set arcwidth (temp var) to* 0.
  $in1$: *set arcwidth (temp var) to* 1.

$in2:$ *set arcwidth (temp var) to*2.

$in3:$ *set arcwidth (temp var) to*3.

$in4:$ *set arcwidth (temp var) to*4.

$-$ *extract id (place of origin) and number of tokens from the message.*

First message we get from this id?

yes : *increment inputs.*

no : *continue*

$-$ *save id, arcwidth and number of tokens in database.*

$-$ *scan the entire database to determine if all input places have enough tokens to enable the Transition.*

transition is enabled?

yes : *set enabled to true*

hold in $(active,\ random\ (0 - 60sec))$

no : *set enabled to false*

passivate

} *end of external transition function*

- $\delta_{int}(s)$ {

  $inputs = 0$?

  yes : $/*$ *transition is a source* $*/$

  hold in $(active,\ random\ (0 - 60\ sec))$

  no : *passivate*

  }

- $\lambda(s)$ {

  $-$ *send* 1 *on out*1 *port.*

  $-$ *send* 2 *on out*2 *port.*

  $-$ *send* 3 *on out*3 *port.*

  $-$ *send* 4 *on out*4 *port.*

  $-$ *go through the database and send a message to every input place via the fired port.*

  } *end of output function*

Si bien CD++ es una herramienta correcta para el modelado y el análisis de las redes de Petri, desde el punto de vista práctico carece de una interfaz adecuada para el estudio de las mismas. Los puertos para cada peso en las transiciones dificultan gravemente el aspecto visual de las redes llevando un modelo de mediana complejidad ilegible. En el siguiente capítulo abordaremos este problema explotando al máximo la potencia de PowerDEVS.

Capítulo 4

Redes de Petri en PowerDEVS

Como dijimos anteriormente, nuestro objetivo es desarrollar una librería que nos permita, por un lado, analizar y simular redes de Petri bajo el formalismo DEVS, y por otro, conservar el aspecto visual original de las mismas.

Una de las principales dificultades al momento de diseñar redes de Petri en PowerDEVS, es el hecho de cuando una transición se dispara, los lugares anteriores deben decrementar su número de marcas de manera atómica. Este comportamiento atómico en DEVS no es trivial, tal como vimos en la sección 3.6, los modelos DEVS cambian su estado de un modelo a la vez, es decir, no puede haber cambio en dos modelos al mismo tiempo.

Por otro lado los modelos DEVS son modulares, es decir, un modelo DEVS no puede conocer u observar el estado de otro modelo. Esta restricción nos lleva a que toda la comunicación entre los modelos debe ser a través de eventos. Por tanto, para que el comportamiento normal de una red de Petri pueda realizarse en DEVS e implementarse en PowerDEVS, primero los lugares deben dar aviso a la transición sobre las cantidades de marcas con la que disponen. En el caso en que la transición pueda dispararse no sólo debe enviar un evento hacia los lugares posteriores, si no también a los lugares anteriores indicando la cantidad de marcas que deben ser restadas.

En este capítulo analizaremos las distintas implementaciones que fueron surgiendo a lo largo del trabajo, conoceremos las principales características, ventajas y desventajas de cada una de ellas. También desarrollaremos el comportamiento interno de cada modelo explicando cada componente de:

$$M = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$$

4.1. Primer enfoque

Esta primera implementación está basada en la especificación de CD++ vista en la sección 3.7. Aquí, se intenta plasmar un esquema base en PowerDEVS, que nos de acceso a una visión general del problema. El apéndice A muestra la implementación del presente enfoque.

Características

- Los lugares no tienen asociado un tiempo (tiempo de espera), únicamente están temporizadas las transiciones (tiempo de disparo). La ausencia de temporización en los lugares forma parte de la idea de hacer una implementación simple;

- Los lugares pueden tener varios puertos de entrada pero sólo uno de salida. De los puertos de entrada el primero (puerto de entrada 0) es utilizado para *conexiones espúreas*[1] y el resto para conexiones pertenecientes a la red. La conexión del puerto de salida puede ser ramificada, y cada uno de sus extremos pueden llegar a una misma transición o distintas;

- Cada mensaje de resta elimina soló una marca de un lugar, visto de otro modo, para que una transición pueda borrar n marcas de un lugar, deberá enviar n mensajes por sus respectivos puertos espúreas;

- Las transiciones poseen tantos puertos de salida como puertos de entrada, más un puerto adicional (puerto de salida 0) para enviar eventos a los lugares posteriores;

- Se toma como convención que el lugar conectado al puerto de entrada i ($i \geq 0$) de una transición tendrá asociado la conexión espúrea del puerto de salida $i + 1$;

- Los mensajes que se transmiten dentro de la red, están definidos por la estructura del Código 4.1, donde sus campos son:

    - un arreglo `y[10]` el cual esta ubicado primero para respetar la convención utilizada en toda la librería, que permite enviar información;

[1] Las *conexiones espúreas* son el medio que utilizan las transiciones para avisarle a los lugares anteriores que deben restar sus marcas. Las mismas deberían permanecer ocultas al usuario.

- un identificador `src` que indica el emisor del mensaje[2];
- un entero `tokens` que representa la cantidad de marcas que contiene un lugar.

Código 4.1 Estructura de los mensajes de una red de Petri.

```
typedef struct sms{
        double y[10];
        Simulator *src;
        int tokens;
        }Message;
```

La definición formal DEVS del lugar es:

Lugar

- $X = \{(p, Message) |\ p \in InPorts\}$
- $Y = \{(p, Message) |\ p \in OutPorts\}$
- $S = \{\{tokens \in \mathbb{N}_0^+\} \times \{Sigma \in \mathbb{R}^+\}\}$ donde el entero *tokens* representa la cantidad de marcas "listas" que contiene el lugar, mientras que *Sigma* mantiene el tiempo de avance del atómico, es decir, el tiempo en que el modelo permanecerá en un estado determinado[3].
- δ_{int} Como los lugares no están temporizados, ante una transición interna siempre se vuelve a un estado pasivo ($Sigma = \infty$).
- δ_{ext} Cuando llega un evento por el puerto 0, se decrementa *tokens* en uno, en cualquier otro caso se aumenta en una unidad la variable *tokens*. En ambas situaciones asigna $Sigma = 0$ para avisar a las transiciones posteriores que la cantidad de marcas del lugar ha cambiado.
- λ Asigna a los campos `y[0]` y `tokens` la cantidad de *tokens* del lugar, y al campo `src` el puntero del lugar (`this`[4]).

[2]En PowerDEVS, los modelos se comunican por medio de *eventos*, estos se encuentran representados por una estructura con los campos *value* y *port* indicando el valor enviado y el puerto destino respectivamente. Es en el espacio *value* donde se ubica el mensaje.

[3]Utilizar una variable *Sigma* igual al tiempo de avance es muy común en DEVS (de hecho, es lo que haremos en todos los modelos de aquí en adelante). De esta forma, para ignorar un evento de entrada se debe dejar el resto del estado como estaba, restándole e (*tiempo transcurrido*) al valor que tenía *Sigma*.

[4]En C++ se define `this` dentro de un objeto como un puntero al objeto en que está contenido.

- *ta* Retorna la variable $Sigma$.

La definición formal DEVS de la transición es:

Transición

- $X = \{(p, Message) |\ p \in InPorts\}$
- $Y = \{(p, Message) |\ p \in OutPorts\}$
- $S = \{\{lastack \in \mathbb{N}^+\} \times \{enabled \in bool\} \times \{working \in bool\} \times \{data \in Database\} \times \{Sigma \in \mathbb{R}^+\} \times \{SigmaFlag \in \{0, 1\}\}\}$ donde *enabled* representa si la transición esta activa (esperando el tiempo de disparo). *working* si se encuentra enviando mensajes de sustracción. *lastack* indica el puerto de salida por el cual se deben enviar dichos mensajes. $SigmaFlag$ es una bandera que representa cuando el sistema se encuentra en estado pasivo. La base de datos *data* lleva la cantidad de lugares que están asociados a la transición, como también provee una función para determinar si la transición se encuentra activa, entre otras. Al igual que en el modelo de lugar, $Sigma$ conserva el tiempo de avance del atómico.
- δ_{int} Si todavía queda algún evento por emitir, decrementa en una unidad a *lastack* y asigna $Sigma = 0$ para generar nuevamente una transición interna. En caso contrario vuelve a un estado pasivo.
- δ_{ext} Al recibir un evento actualiza la base de datos *data* con el par (*id*,*tokens*) y verifica si la transición se encuentre activa. Si es la primera vez que se activa ($enabled = false$) asigna $Sigma = time$ ($time$ es el tiempo de disparo), y cambia su estado a $enabled = true$. En el caso en que ya se encuentre trabajando (*working*) o que este habilitada pero esperando el tiempo de disparo (*enabled*), ignora el evento.
- λ Si la transición esta en un estado tal que $enabled = true$ o $working = true$ envía un mensaje de sustracción por el número de puerto que figura en *lastack*. Cuando haya terminado de enviar todos los mensajes de resta, manda el mensaje de suma a los modelos posteriores por el puerto 0.
- *ta* Retorna la variable $Sigma$.

Observación: Por cuestiones de simplicidad, en la implementación del componente transición, se realizan cambios de estado durante el método `Output` (ver Código A.12 de la página 82).

Para entender mejor la idea, expondremos las cualidades de este enfoque en el modelo de un elevador [20] (ver Figura 4.1).

El sistema comienza con tres marcas en el lugar Place1 y con cero en el lugar Place2. Las marcas indican el número de movimientos que el elevador puede hacer hacia arriba o hacia bajo dependiendo del lugar en que se encuentren. En el estado inicial el sistema se encuentra en la Planta Baja y puede subir tres pisos. Las transiciones up y down representan subir y bajar un piso respectivamente. Observar además que mientras el ascensor este situado en Planta Baja no podrá descender (ausencia de marcas en Place1), y cuando se encuentre en el Tercer piso no podrá ascender (ausencia de marcas en Place2).

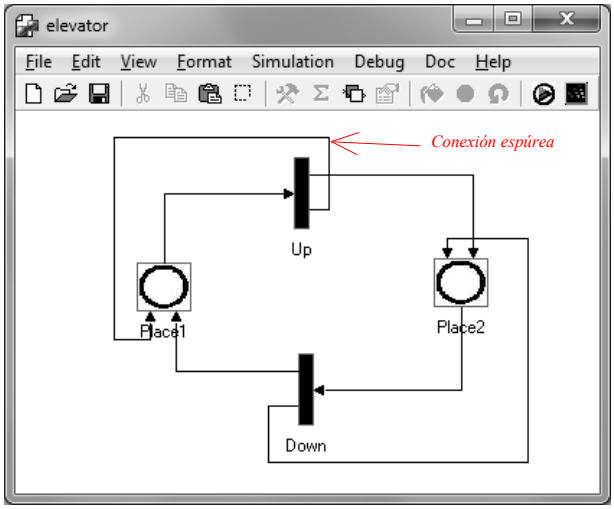

Figura 4.1: Modelo simple de un elevador

Por otro lado, como ya se mencionó en la sección 3.6.2, PowerDEVS permite asignarle prioridades a los modelos para determinar cuál debe ejecutarse en caso de que dos o más puedan transicionar al mismo tiempo. Ahora bien, supongamos el escenario de la Figura 4.2 en donde dos transiciones tengan un mismo lugar anterior en común, que ambas estén activas, pero las marcas del lugar sólo pueden satisfacer a una de ellas. ¿Qué debería suceder?, debería ejecutarse sólo una de ellas (la de mayor prioridad). Sin embargo, en nuestra implementación se ejecutarían ambas, la de mayor prioridad seguida de la de menor prioridad, dejando el lugar con un número negativo de marcas.

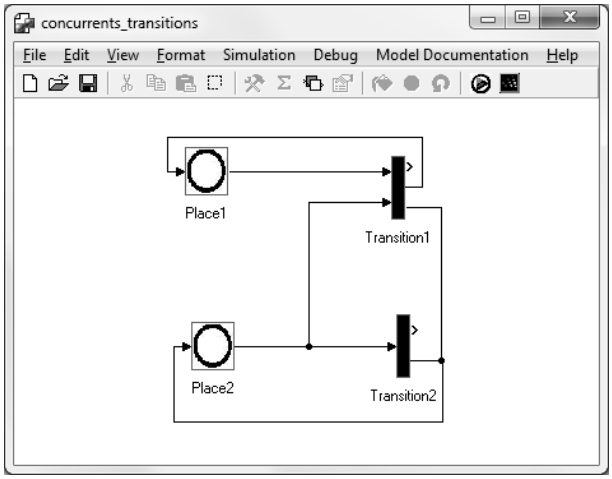

Figura 4.2: Dos transiciones que se activan concurrentemente con un lugar anterior en común

Ventajas

- Fácil de implementar.

Desventajas

- Las conexiones espúreas tornan ilegible el sistema cuando la red es medianamente grande.
- La convención, existente entre los puertos de entrada y los puertos de salida de una transición, hace que el usuario tenga que preocuparse más en el orden de las conexiones, que en el sistema en sí.
- La ausencia de tiempos de espera en los lugares restringe la capacidad para representar modelos de mayor complejidad.
- En el caso en que dos transiciones quieran dispararse simultáneamente, y estas tengan un lugar anterior en común, no funciona. Sin embargo es un error admisible para un primer intento.

4.2. Segundo enfoque: Hand-Shake

Ahora que ya tenemos una idea de como se pueden diseñar las redes de Petri en PowerDEVS, refinaremos nuestro primer enfoque a modo de aproximarnos un poco más a una correcta im-

plementación. Nos inspiraremos en el protocolo *Hand-Shake* utilizado para el establecimiento de una conexión entre dos computadoras. Además, sacaremos provecho de los campos de los mensajes para facilitar la implementación.

Características

- Tanto las transiciones como los lugares se encuentran temporizados.
- Las transiciones tienen una única conexión espúrea, la cual se conecta con todos los lugares anteriores, esto es, ahora los mensajes de resta tendrán en el campo `src` el lugar que debe tomarlo (el resto de los lugares omitirán el mensaje). Además, para evitar múltiples arcos desde las transiciones a los lugares, el atributo `tokens` contiene la cantidad de marcas que deben sustraerse, de esta forma sólo debe enviarse un mensaje por cada lugar.
- La convención entre los puertos de entrada y los puertos de salida de una transición precisada en el enfoque anterior, es reemplazada por la utilización del identificador `src` para la clasificación de los mensajes de resta.
- Utiliza un protocolo *Hand-Shake* para abordar el problema, de que dos o más transiciones estén activas concurrentemente y sólo pueda dispararse una, planteado en el primer enfoque. Este método se basa en que una transición al activarse, primero envíe un evento *Request* a sus lugares anteriores y estos acusen *Ok*, estableciendo una especie de "enlace" entre ambos. A su vez, **todos los lugares deben tener mayor prioridad que las transiciones**, esto es para que cuando se ejecute la transición de mayor prioridad y se consuman las marcas, los mismos envíen inmediatamente el número de marcas *actualizado* al resto de las transiciones (de menor prioridad) y estas, se desactiven (caso en que la cantidad de marcas actual sea menor que la precisada por cada transición) o se ejecuten (caso en que una o más transiciones permanezcan activadas). Observar que en esta última situación si dos o más transiciones continúan activadas, transicionará nuevamente la de mayor prioridad.
- A diferencia del primer enfoque, los mensajes contienen un nuevo campo (`src2`) para el envío del identificador de las transiciones cuando las mismas hacen un Request.

Código 4.2 Estructura de los mensajes de una red de Petri (Hand-Shake)

```
typedef struct sms{
        double y[10];
        Simulator *src;
        Simulator *src2;
        int tokens;
        }Message;
```

Ventajas

- Las redes de Petri tienen un aspecto más natural, dado que sólo se precisa una conexión espúrea para restar marcas de los lugares anteriores y no una por cada lugar anterior (como sucedía en la primera implementación).
- La inclusión de tiempos de espera en los lugares aumenta el poder expresivo de las redes.

Desventajas

- Todavía no se tiene un aspecto totalmente natural, se conservan conexiones espúreas.
- Si bien resuelve el problema de que una transición consuma tokens inexistentes, acarrea uno nuevo: se debe tener en cuenta que todos los lugares tengan más prioridad que las transiciones.

  En la Figura 4.3 se observan dos transiciones que comparten un mismo lugar en común. El sistema esta configurado con una marca en el lugar Place1 con tiempo de espera uno, y con una marca en Place2 con tiempo de espera cero. Ambas transiciones tienen tiempo de disparo cero. En este ejemplo las prioridades de los modelos juegan un papel importante, dado que de ellas depende el correcto o incorrecto funcionamiento del sistema.

  Si tomamos el orden de prioridades [Place1 > Place2 > Transition1 > Transition2], la simulación respeta la semántica, la ejecución del sistema es la siguiente:

  - Transition1 envía un Request a Place1 y a Place2
  - Place1 responde Ok
  - Place2 responde Ok

- Inicia Transition1
- Place1 notifica que su cantidad de marcas es 0
- Place2 notifica que su cantidad de marcas es 0
- Transition2 se desactiva

Observar que aquí la comunicación entre Transition1 y los lugares es atómica.

En cambio, si tomamos el orden [Transition1 > Place2 > Transition2 > Place1], la ejecución del sistema es la siguiente:

- Transition1 envía un Request a Place2 y a Place1
- Place2 responde Ok
- Transition2 envía un Request a Place2
- Place2 responde Ok
- Inicia Transition2
- Place1 contesta Ok
- Inicia Transition1
- Place2 notifica que su cantidad de marcas es -1

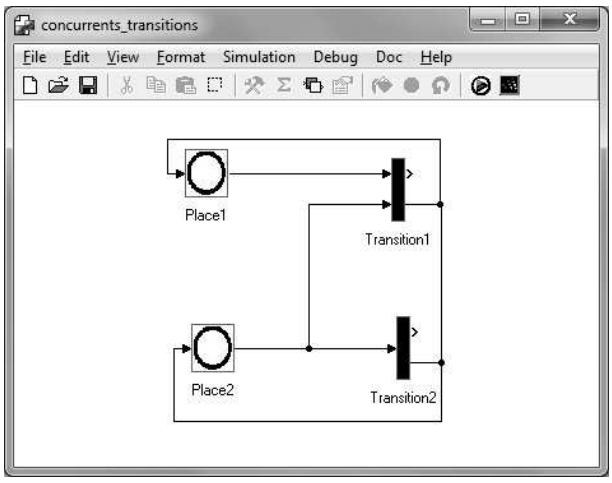

Figura 4.3: Dos transiciones que se activan concurrentemente con un lugar anterior en común.

4.3. Tercer enfoque: Hand-Shake con conexiones virtuales

Continuando con la búsqueda de un modelo ideal, proponemos un pequeño cambio, la sustitución de conexiones espúreas por *conexiones virtuales*. Decimos pequeño, porque si bien el aspecto visual es distinto, el funcionamiento en sí no se ve alterado. El apéndice B presenta la implementación de este enfoque.

Características

- Las conexiones espúreas son reemplazadas por conexiones virtuales. Estas últimas cumplen la misma función que las primeras con la diferencia de que son invisibles para el usuario. La idea es que un lugar al enviar un evento a una transición posterior, guarda el puntero `this` en el campo `src` del mensaje. La transición al recibir el evento, llama al método `externalinput` del lugar (a través del puntero almacenado en el mensaje), ejecutando una transición externa y en consecuencia el envío de un evento sin que exista una conexión.

- Un punto importante, relacionado con el ítem anterior, es la sencillez en que están implementadas las conexiones virtuales entre las transiciones y los lugares. En lo que sigue se detalla la parte del código más relevante.

Código 4.3 Lineas que componen a las conexiones virtuales en la subrutina *Internal Transition* ($\delta_{int}()$) de las Transiciones.

```
(message -> src) -> externalinput(t,(int)message);
```

En el Código 4.3 se llama al método `externalinput` con los argumentos `t` (tiempo de simulación) y `message` (mensaje de resta).

Código 4.4 Lineas que componen a las conexiones virtuales en la subrutina *External Transition* ($\delta_{ext}()$) de los Lugares.

```
if (x.port == -1)
 tokens = tokens - mens -> tokens;
```

En el Código 4.4 primero se verifica si el evento llega por el puerto `-1`, de ser así, se resta a la variable de estado `tokens` el número de marcas indicado en el campo `tokens` del mensaje[5].

Lugar

- $X = \{(p, Message) |\ p \in InPorts\}$
- $Y = \{(p, Message) |\ p \in OutPorts\}$
- $S = \{\{tokens \in \mathbb{N}_0^+\} \times \{Sigma \in \mathbb{R}^+\} \times \{list \in LinkedList\} \times \{auxSigma \in \mathbb{R}^+\} \times \{TimeFlag \in \{0,1,2,3,4\}\} \times \{auxTimeFlag \in \{0,1,2,3,4\}\} \times \{transition \in \mathbb{N}^+\}\}$. $TimeFlag$ indica en que estado está el lugar, indicando 0 el estado pasivo, 1 que hay marcas cumpliendo el tiempo de espera, 2 alguna transición se encuentra borrando marcas, 3 realizando el Hand-Shake y 4 hay marcas que cumplieron su tiempo de espera. Las variables $auxTimeFlag$ y $auxSigma$ son utilizadas para guardar un estado (al que se necesitará regresar) de $TimeFlag$ y $Sigma$ respectivamente. $transition$ almacena el puntero de la transición que esta realizando el Hand-Shake. $list$ es una lista enlazada donde se guardan las marcas con sus tiempos de espera, la misma contiene métodos para actualizar el tiempo restante y obtener las marcas que están listos para ser removidas.
- δ_{int} En el caso en que se haya finalizado el Hand-Shake, ejecuta la subrutina externalinput de la transición guardada en $transition$ con los parámetros t (tiempo de simulación) y el mensaje previamente creado en δ_{ext}. Si se cumple el tiempo de espera de alguna marca, se actualiza el valor de la variable $tokens$ y se pone el $Sigma = 0$ dando aviso de este nuevo estado. Cuando termine de enviar el evento, corrobora si todavía hay marcas en la lista. En caso afirmativo asigna al tiempo de avance el tiempo en que la próxima marca va a estar activa ($Sigma = min(list)$; $TimeFlag = 1$), de otra manera vuelve a un estado pasivo($Sigma = \infty$; $TimeFlag = 0$). Finalmente, cuando una transición consuma alguna marca, se vuelve al estado guardado en $auxSigma$ y $auxTimeFlag$.
- δ_{ext} Cuando llegue un evento por el puerto -1, verifica si en el campo `src` del mensaje contiene el puntero `this`. De ser así, decrementa la variable $tokens$ con el número de marcas que figura dentro del mensaje, guarda el estado de $Sigma$ y $TimeFlag$ en las

[5]En PowerDEVS el puerto `-1` es utilizado para conexiones virtuales.

respectivas variables auxiliares y selecciona $Sigma = 0$ para forzar una transición interna. Si el puntero almacenado en `src` no referencia al lugar en cuestión, verifica si $TimeFlag$ es 0 o 1. En el caso afirmativo, hace un backup del estado de $TimeFlag$ y de $Sigma$, además guarda el puntero contenido en el mensaje dentro de la variable *transition* y selecciona $Sigma = 0$. Desde luego, si no cumple ninguna de las condiciones anteriores (el lugar ya esta trabajando con otra transición) se ignora el evento.

Por otro lado, cuando reciba un evento por cualquier otro puerto, si se encuentra en un estado pasivo, seleccionamos $Sigma = time$ y $TimeFlag = 1$, caso contrario ignoramos el evento.

No importa el motivo por el cual se ejecuta la transición externa siempre se actualiza el tiempo de las marcas contenidas en *list*.

- λ Envía un evento que tiene como valor un mensaje con el campo `y[0] =` *tokens* (para que pueda ser utilizado por cualquier bloque de la librería), `tokens =` *tokens*, `src = this` y `src2 = this`. El mensaje resultante es emitido por el puerto 0.

  Observación: Cuando precisamos realizar una transición interna sin que se transmita un evento a los bloques posteriores, tal es el caso en que $Sigma = 0$ y $TimeFLag = 1$, durante la función *Output* se debe enviar un evento nulo (`return Event();`), el cual no es emitido por ningún puerto y por lo tanto no es recibido por ningún modelo.

- *ta* Retorna la variable $Sigma$.

Transición

- $X = \{(p, Message) |\ p \in InPorts\}$
- $Y = \{(p, Message) |\ p \in OutPorts\}$
- $S = \{\{numacks \in \mathbb{N}_0^+\} \times \{data \in Database\} \times \{Sigma \in \mathbb{R}^+\} \times \{predecessor \in \mathbb{R}^+\} \times \{HandShake \in \{0, 1, 2, 3\}\}\}$. *numacks* acumula la cantidad de OKs recibidos. *precessor* guarda un puntero de un lugar anterior a la transición. $HandShake$ indica el estado del Hand-Shake, indicando 0 el Hand-Shake debe hacerse, 1 realizando el Hand-Shake, 2 se finalizó el Hand-Shake y 3 nunca debe realizarse el Hand-Shake (transición fuente).

- δ_{int} Se puede dar una transición interna cuando nos encontramos en uno de los siguientes estados: $enabled = true$ en donde realizamos un Request a cada lugar anterior y pasamos a un estado pasivo (esperando los respectivos OKs), $Hand - Shake = 2$ donde terminó de ejecutar el Hand-Shake y se procede a restar las marcas necesarias, y finalmente en el estado con $Hand - Shake = 3$ donde asigna $Sigma = time$.

- δ_{ext} Cuando arribe un evento por el puerto -1 incrementa en uno a $numacks$, y verifica si ha recibido todos lo Acks entonces cambia su estado a $Hand - Shake = 2$ y fuerza una transición interna, caso contrario, vuelve a un estado pasivo. Para cualquier otro evento actualiza la Base de Datos y comprueba si la transición está activa. En el caso de que sea la primera vez que se activa, comienza el tiempo de disparo y cambia su estado a *enabled*. En el caso en que se encuentre realizando el Hand-Shake entonces lo reinicia, si este ya termino selecciona $Sigma = 0$. Para los estados en que no se active, diferenciamos lo casos de $HandShake = 1$ volviendo a un estado pasivo, $HandShake = 2$ coloca $Sigma = 0$. Finalmente, para cualquier otro caso selecciona $enabled = false$ e ignora el evento.

- λ Prepara los mensajes poniendo el id de la transición en el campo `src` y el valor 0 en el campo `tokens`.

- ta Retorna la variable $Sigma$.

Desde este nuevo enfoque, el ejemplo del elevador y el problema de las transiciones paralelas experimentados en la sección 4.1, quedan conforme las Figuras 4.4 y 4.5 respectivamente.

Ventajas

- Se logra la apariencia deseada, es decir, las redes de Petri preservan su aspecto original (ausencia de conexiones espúreas).

Desventajas

- Conserva el problema de las transiciones concurrentes, presentado en las anteriores implementaciones.

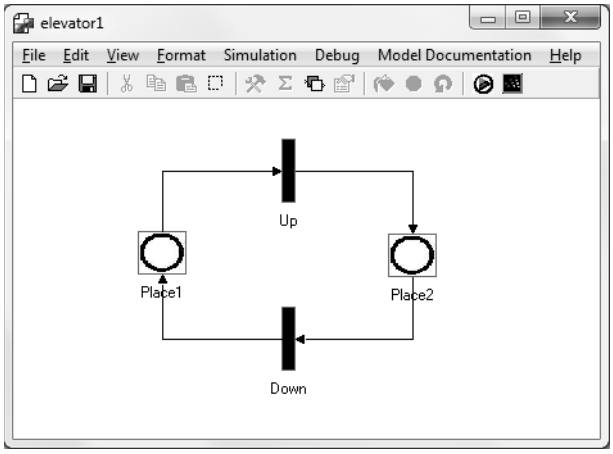

Figura 4.4: Modelo simple de un elevador sin conexiones espúreas

4.4. Enfoque Final

En esta última versión logramos un correcto funcionamiento, alcanzando todos los objetivos propuestos con una implementación sencilla, no obstante creemos que no podría haber sido posible obtener un buen resultado sin haber tenido que trazar el camino mencionado en este informe. Los apéndices C y D ilustran, en conjunto, la implementación final.

Características

- Expandimos la flexibilidad de las redes de Petri agregando un componente especial, *Transición Híbrida* (Event Triggered Transition). Esta transición tiene como objetivo acoplar redes de Petri con otro tipo de sistemas. Por lo tanto, podremos modelar cada parte de un sistema (subsistemas) con el lenguaje de especificación más adecuado, logrando un modelo óptimo y claro. En el Capítulo 5 se muestra un ejemplo híbrido donde dicha transición es utilizada.

- Reemplazamos el Hand-Shake por una nueva implementación, que si bien es un poco "sucia" como veremos en breve, deja un código mucho más legible y no requiere asignarle mayor prioridad a lo lugares.

- Los mensajes vuelven a tomar la forma del primer enfoque.

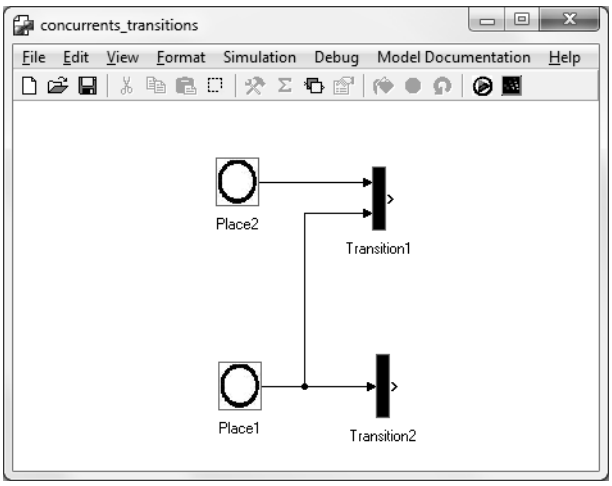

Figura 4.5: Dos transiciones que se activan concurrentemente con un lugar anterior en común.

El Hand-Shake es sustituido por la incorporación de los métodos `GetTokens()` y `SetTokens()` a la interfaz de los lugares, pues, a diferencia del Hand-Shake, permiten consultar y modificar las marcas de un lugar de manera atómica (a través del puntero contenido en el mensaje, por ejemplo `msg.src->GetTokens()`) perdiendo la necesidad de que los lugares tengan mayor prioridad que las transiciones. Dichos métodos, destruyen la modularidad de DEVS (un modelo no puede ver ni modificar directamente el estado de otro). El Código 4.5 exhibe la interfaz de los lugares.

Código 4.5 Interfaz del modelo Place expandida con las subrutinas para la consulta y actualización de tokens.

```
void init(double, ...);
double ta(double t);
void dint(double);
void dext(Event , double );
Event lambda(double);
void exit();
/* Métodos adicionales */
int GetTokens(){return tokens;};
void SetTokens(int m) {tokens = m;};
```

Observar que los métodos `GetTokens()` y `SetTokens()` están implementados dentro de la interfaz de la clase *Place*.

Lugar

- $X = \{(p, Message) |\ p \in InPorts\}$
- $Y = \{(p, Message) |\ p \in OutPorts\}$
- $S = \{\{tokens \in \mathbb{N}_0^+\} \times \{list \in List(\mathbb{R}^+)\} \times \{Sigma \in \mathbb{R}^+\} \times \{transition \in \mathbb{R}^+\} \times \{TimeFlag \in \{0, 1, 4\}\}\}$. Las variables ya mencionadas mantienen el significado anteriormente declarado.
- δ_{int} Si una marca cumple el tiempo de espera, es agregada a *tokens*, se pone $Sigma = 0$ para notificar a las transiciones posteriores que hay una nueva marca lista y se pasa a un estado en donde hay nuevas marcas activas, por lo tanto se cambia a $TimeFlag = 4$. Si $TimeFlag$ ya se encuentra en este último estado (denotando que termino de enviar un evento) corrobora si todavía hay marcas en la lista. En caso afirmativo asigna al tiempo de avance el tiempo en que la próxima marca va a estar activa ($Sigma = min(list)$; $TimeFlag = 1$), de otra manera vuelve a un estado pasivo ($Sigma = \infty$; $TimeFlag = 0$).
- δ_{ext} Se actualiza la lista teniendo en cuenta el tiempo transcurrido. Se agrega una nueva marca a la lista con tiempo de espera igual al del lugar. Luego se asigna a $Sigma$ el tiempo en el cual se va a activar la próxima marca.
- λ Envía un evento que tiene como valor un mensaje con la variable `y[0] = tokens`, para que pueda ser utilizado por cualquier bloque de la librería, el campo `tokens` igual a la cantidad de marcas listas en el lugar (variable de estado *tokens*) y el campo `src = this`, es decir se guarda el puntero del atómico que está emitiendo el evento.
- ta Retorna la variable $Sigma$.

Transición

- $X = \{(p, Message) |\ p \in InPorts\}$
- $Y = \{(p, Message) |\ p \in OutPorts\}$

- $S = \{\{data \in \mathbb{N} \to (\mathbb{N} \times \mathbb{N})\} \times \{Sigma \in \mathbb{R}^+\} \times \{predecessor \in \mathbb{R}^+\} \times \{enabled \in bool\} \times \{source \in bool\} \times \{working \in bool\} \times \{portsinit \in bool\}\}$ donde la variable booleana *source* es true cuando se trata de una transición fuente, *portsinit* se mantiene false hasta que la transición conozca cuantos lugares anteriores tiene asociados.

- δ_{int} Si la transición está activa ($enabled = true$), la transición debe verificar nuevamente (mediante la subrutina `GetTokens()`) si todos los lugares anteriores contienen la cantidad de marcas suficientes para que se dispare. Esto debe realizarse ya que otra transición pudo haber realizado un disparo en ese instante quitando alguna marca necesaria. Si la verificación se satisface, se quita de cada lugar las marcas utilizadas por la transición por medio de la subrutina `SetTokens()`. Tanto la verificación de cuantas marcas tiene un lugar, como la acción de quitarlas se realiza de manera no modular, accediendo directamente al estado de los lugares a través del puntero incluido en el mensaje (y guardado en la dataBase). Previamente al disparo de la transición, se agenda un evento para notificar a los lugares posteriores que deben agregar una marca y se selecciona $working = true$. Una vez que haya emitido el aviso de suma, notifica a los lugares anteriores que les ha quitado marcas (a través de una transición externa simulada `externalinput`). En el caso en que los lugares anteriores no contengan la cantidad de marcas necesarias, pasa a un estado pasivo ($enabled = false; Sigma = \infty$).

  Ahora bien, si no entra en ninguno de los casos anteriores, analiza si se trata de una transición fuente en donde asignará el tiempo de disparó a *Sigma*. De lo contrario, vuelve al estado pasivo.

- δ_{ext} Cuando la transición recibe un mensaje, inmediatamente actualiza la base de datos y consulta en la misma si se conoce los lugares anteriores con sus pesos. De ser así, selecciona $portsinit = true$. Luego, si *portsinit* se satisface, debe ver si todos los lugares anteriores tienen la cantidad de marcas necesarias para activarse. Si es así, y la transición no estaba activa, la transición se activa y espera su tiempo de disparo ($enabled = true; Sigma = time$), si no, sigue esperando su tiempo de disparo tomando en cuenta el tiempo ya transcurrido. El caso por default es que se mantenga en un estado pasivo y desactivada.

- λ Envía un evento que tiene como valor un mensaje con la variable `y[0] = 0`, `tokens = 0` y el campo `src = this`. Todos los bloques (de la librería) conectados a esta transición

verán un evento con valor 0 cuando la transición se dispare.

- *ta* Retorna la variable *Sigma*.

Transición Híbrida

- $X = \{(p, Void) | \ p \in InPorts\}$
- $Y = \{(p, Message) | \ p \in OutPorts\}$
- $S = \{Sigma \in \mathbb{R}^+\}$
- δ_{int} Sitúa a la transición en un estado pasivo ($Sigma = \infty$).
- δ_{ext} Al recibir la transición un evento (de cualquier tipo) genera una transición interna asignando el valor 0 a *Sigma*.
- λ Envía un mensaje con los campos `y[0] = 0`, `tokens = 0` y `src = this`.
- *ta* Retorna la variable *Sigma*.

4.5. La Librería

Ahora, ya cumplidos nuestros objetivos de modelar cada componente de una red de Petri, queda únicamente crear una librería que nos permita construir sistemas de una manera práctica y eficiente. Esto es tarea fácil en PowerDEVS, ya que provee una opción para crear una librería dado un conjunto de modelos previamente definidos, en la Figura 4.6 se muestra la librería resultante.

Aspectos a tener en cuenta al momento de manipular la librería:

- Tanto las transiciones como los lugares pueden ser utilizados como fuentes y sumideros. En este último caso, ambos componentes presentan una "anormalidad" gráfica (un puerto de salida en desuso), que puede prestar a confusión. La Figura 4.7 ilustra el formato de las fuentes y sumideros de cada modelo.
- El modelo atómico *Place* tiene los parámetros *tokens* (número de marcas iniciales), *time* (tiempo de espera) y *inputs* (cantidad de puertos de entrada) que pueden ser configurados. La Figura 4.8 ilustra el menú de configuración del componente.

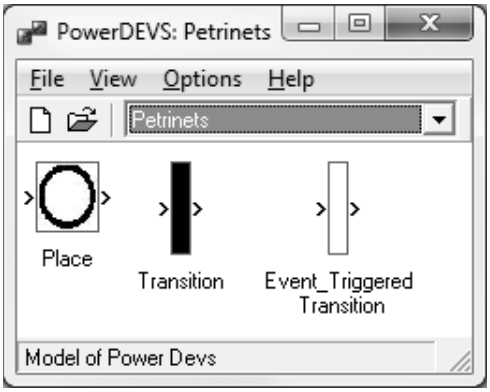

Figura 4.6: Librería para redes de Petri de PowerDEVS

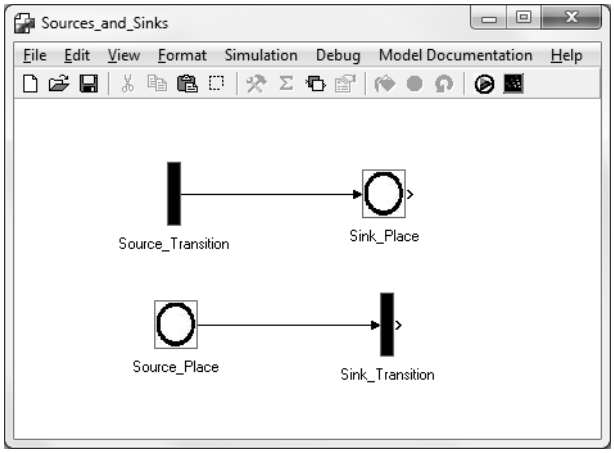

Figura 4.7: Formato de sumideros y fuentes

- El modelo atómico *Transition* tiene los parámetros *time* (tiempo de disparo) y *inputs* (cantidad de puertos de entrada) que pueden ser configurados. La Figura 4.9 ilustra el menú de configuración del componente.
- El modelo atómico *Event_Triggered Transition* no posee campos configurables.

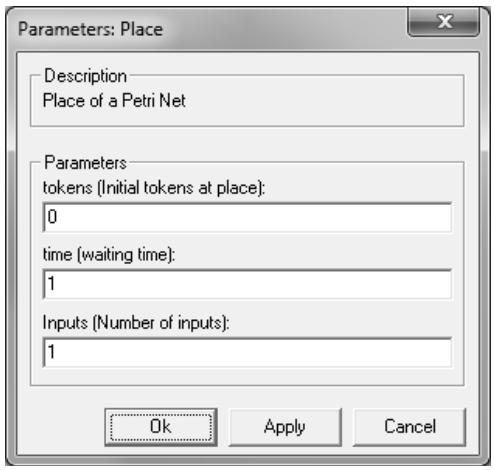

Figura 4.8: Parámetros del modelo atómico *Place*

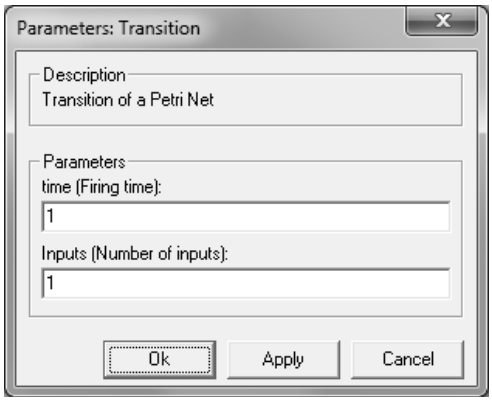

Figura 4.9: Parámetros del modelo atómico *Transition*

Capítulo 5

Ejemplos y Aplicaciones

En este capítulo se analizan distintos ejemplos que muestran el alcance de las redes de Petri y su práctica en el mundo real. El objetivo es verificar el correcto funcionamiento de la librería y mostrar su interacción con el resto de los bloques de PowerDEVS para construir modelos híbridos complejos. Los ejemplos forman parte de la distribución de PowerDEVS[1].

5.1. Cola-Procesador

En la sección 2.4 se presentó un sistema Cola-Procesador el cual recibe pedidos, los almacena y los procesa. La Figura 5.1 muestra el modelo de la Red de Petri correspondiente usando la librería de PowerDEVS desarrollada[2].

En la Figura 5.2 se puede observar la evolución de la cantidad de trabajos en cola (línea sólida ascendente) y, graficados como eventos (línea de puntos), los avisos del procesador al terminar cada trabajo y tomar otro.

[1]Con el fin de brindar una mejor explicación, no se tendrá en cuenta las prioridades asignadas a los modelos de PowerDEVS.

[2]Las transiciones son instantáneas ($time = 0$), a excepción de la Transition1 que genera los trabajos. Esta tiene un tiempo de disparo de 1,3 unidades de tiempo.

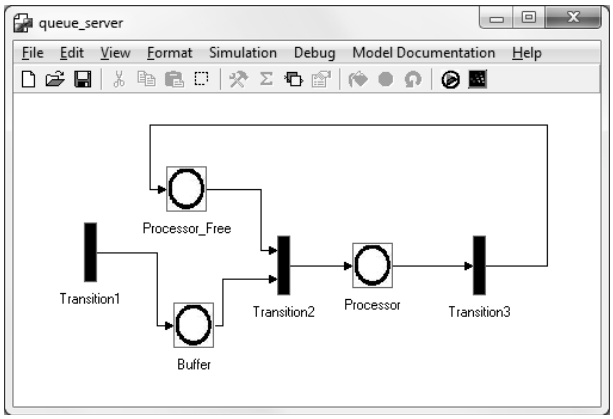

Figura 5.1: Sistema Cola-Procesador con un tiempo de dos segundos de procesamiento.

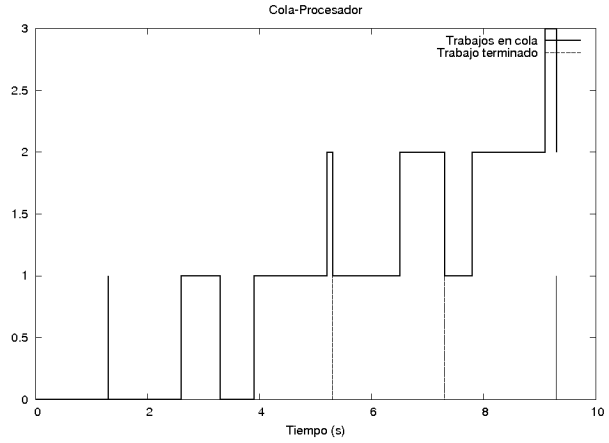

Figura 5.2: Evolución del sistema Cola-Procesador

5.2. Filósofos comensales

El problema de los filósofos comensales [15, 20] (dining philosophers) es un problema clásico de las ciencias de la computación propuesto por Edsger Dijkstra para representar el problema de la sincronización de procesos en un sistema operativo. Aquí, únicamente haremos un modelo del problema (es decir, no una solución) mediante el uso de la librería implementada.

Enunciado del problema

Cuatro filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos

son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda agarrar el otro tenedor, para luego empezar a comer.

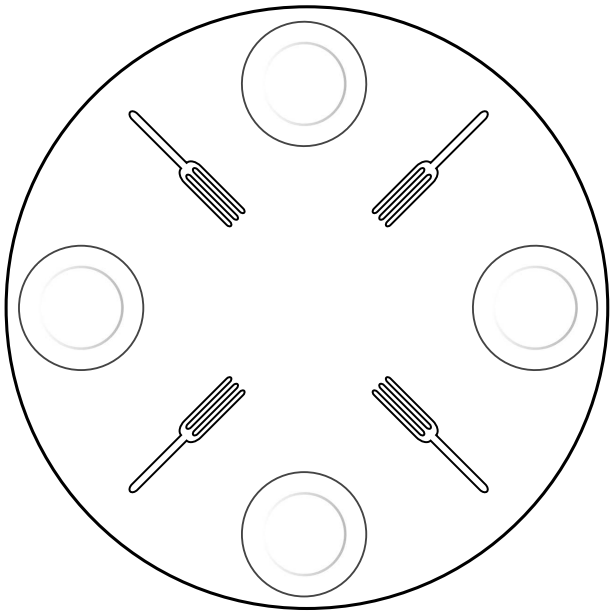

Figura 5.3: Esquema del problema de los Filósofos comensales

Si dos filósofos adyacentes intentan tomar el mismo tenedor al mismo tiempo, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo o deadlock.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de inanición.

Modelo de los Filósofos comensales

En la Figura 5.4 se ilustra el modelo del problema, donde:

- T: Pensando (Thinking)
- TE: Termina de pensar, Comienza a comer (Finishing thinking, Start eating)
- E: Comiendo (Eating)
- ET: Termina de comer, Comienza a pensar (Finishing eating, Start thinking)
- F: Tenedor (Fork)

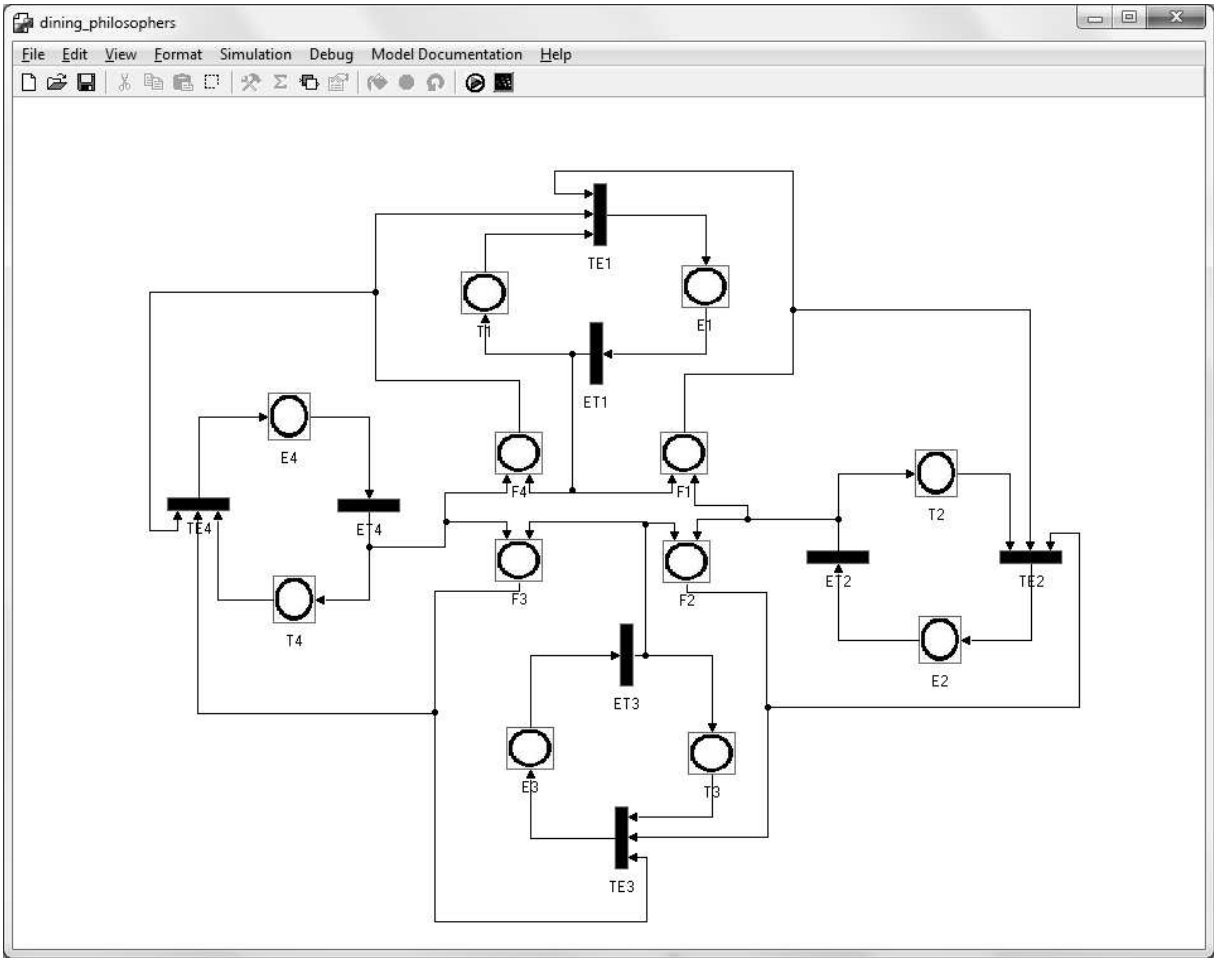

Figura 5.4: Filósofos comensales en PowerDEVS

En el estado inicial el sistema contiene un token en cada estado F (todos los tenedores están libres) y un token en cada estado TE (todos los filósofos se encuentran pensando).

Al comenzar la simulación, se puede ejecutar cualquiera de las transiciones TE, es decir, cualquiera de los filósofos puede dejar de pensar y comenzar a comer. Dado que para la finalidad del análisis no importa que transición TE se dispare, supongamos que se activa TE1. Al transicionar TE1 consume las marcas de los lugares F4, F1 y T1, y aumenta una marca al lugar E1. En este nuevo estado del sistema las únicas transiciones que pueden ejecutarse son ET1 y TE3, esto es, o el filósofo 1 termina de comer y comienza a pensar o el filósofo 3 termina de pensar y comienza a comer. En el caso en que se disparara TE3, todos los tenedores estarían en uso y por lo tanto los filósofos 1 y 2 se verían obligados a tener que esperar que los otros dos terminen de comer. En el caso opuesto, la transición que se activa es ET1, el sistema volvería al estado inicial.

5.3. Protocolo Stop and wait ARQ

Stop and wait [12, 18] (Parada y espera) es un protocolo ARQ[3] para el control de errores en la comunicación entre dos hosts (a través de la capa de enlace del modelo OSI) basado en el envío de tramas o paquetes, de modo que una vez que se envía un paquete no se envía el siguiente hasta que no se reciba el correspondiente ACK (confirmación de la recepción), en el caso de no recibir en un periodo de tiempo estipulado el acuse, se reenvía el paquete en cuestión.

Durante el protocolo, pueden surgir dos clases de problemas: la pérdida de un paquete o la pérdida de un acuse de recibo. Tanto la conducta normal del mismo como la posible presencia de errores durante la transmisión de datos, deben estar representadas en el modelo. La Figura 5.5 ilustra dicho comportamiento, donde:

- Send_Pkt: Recibe el ACK del paquete enviado anteriormente, envía el siguiente paquete
- Pkt: Enviando paquete
- Loss_Pk: Pérdida del paquete
- Get_Pkt: Paquete recibido correctamente, envía próximo paquete

[3]El ARQ (Automatic Repeat-reQuest) es un protocolo utilizado para el control de errores en la transmisión de datos, garantizando la integridad de los mismos. Éste suele utilizarse en sistemas que no actúan en tiempo real ya que el tiempo que se pierde en el reenvío puede ser considerable y ser más útil emitir mal en el momento que correctamente un tiempo después.

- Ack: Enviando ACK
- Loss_Ack: Pérdida del ACK
- Wait_Ack: Esperando próximo ACK
- Timer: Termina de esperar el ACK, reenvía el paquete
- Wait_Pkt: Esperando próximo paquete
- Reject: Paquete recibido correctamente, envía ACK

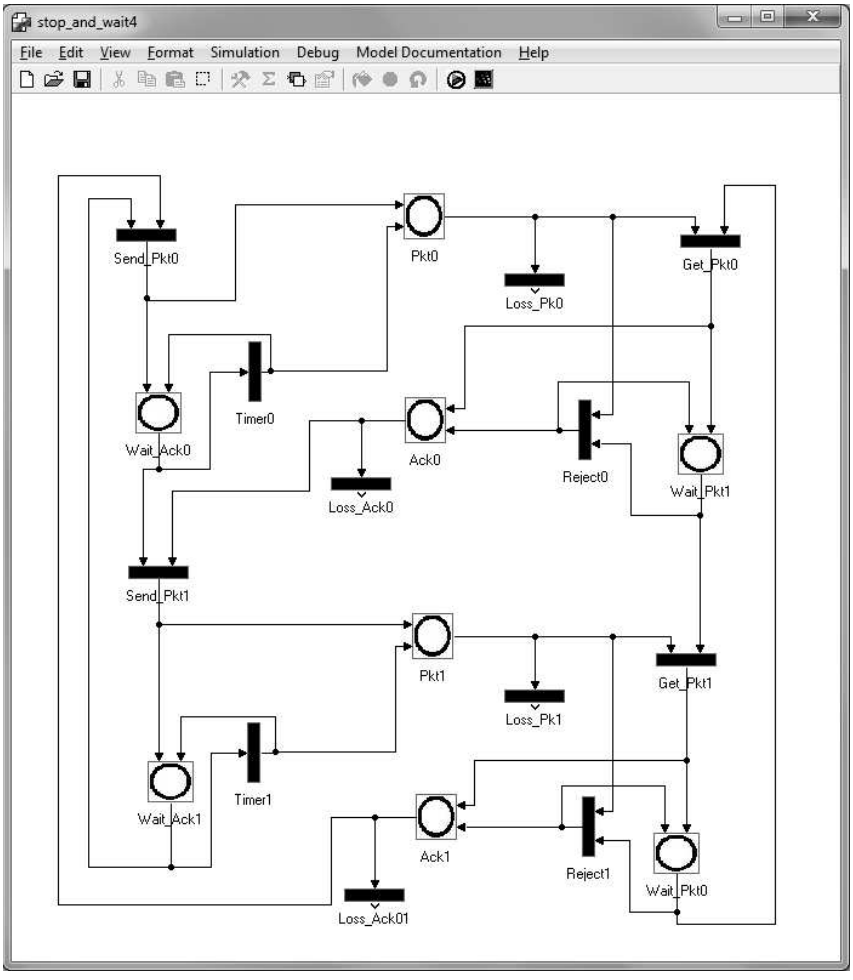

Figura 5.5: Protocolo Stop and Wait ARQ en PowerDEVS.

Un estado inicial posible del sistema es el cual los lugares Wait_Ack1 y Ack1 contienen una marca y el resto de los lugares no poseen ninguna.

5.4. Robot (Ejemplo Híbrido)

La idea es que hay una mesa con varios paquetes de dos clases en la posición $(0, 0)$ y un robot debe llevar los objetos a dos mesas distintas (según el tipo de paquete). Los paquetes del primer tipo deben ir a la posición $(1, 1)$ y los del segundo tipo a la posición $(1, -1)$.

La lógica es la siguiente:

- el robot toma un objeto de la primer clase y lo lleva hasta la mesa correspondiente;
- una vez que llega a la mesa espera 1 segundo y vuelve al origen;
- tras un segundo en el origen, si hay un paquete de la otra clase, lo toma. Si no, espera 1 segundo más y si no llegó ningún paquete de dicha clase toma uno de la otra clase.

Este control de alto nivel (ver Figura 5.6) entonces recibe un evento que indica que el robot llegó a destino, y emite tres eventos distintos: uno indica que el robot debe ir a $(1, 1)$, otro que indica que el robot debe ir a $(1, -1)$ y el último que indica que el robot debe ir a $(0, 0)$.

Luego, hay un sistema que adapta las referencias. Según la orden del sistema de control de alto nivel, produce las referencias para las posiciones x e y (o sea, produce los valores 0, 1 o -1 en cada variable).

Finalmente está el modelo del robot y su control. El robot es un modelo que rota y avanza hacia adelante. Las entradas son la velocidad de avance y la velocidad de rotación y los estados del robot son la posición en "x", la posición en "y" y el ángulo de rotación. El control del robot está diseñado con una técnica que se llama "dynamic feedback linearization", y está implementado con "Quantized State Control". En el modelo controlado, hay dos señales de referencia (que valor se quiere tener en x y que valor se quiere tener en y) y el control automáticamente ajusta las velocidades de avance y rotación para que luego de un tiempo el robot llegue a la posición deseada. La Figura 5.7 ilustra el modelo.

En el ejemplo de la Figura 5.8, inicialmente hay cuatro paquetes de un tipo y dos del otro en el origen, y el robot se encuentra en la mesa.

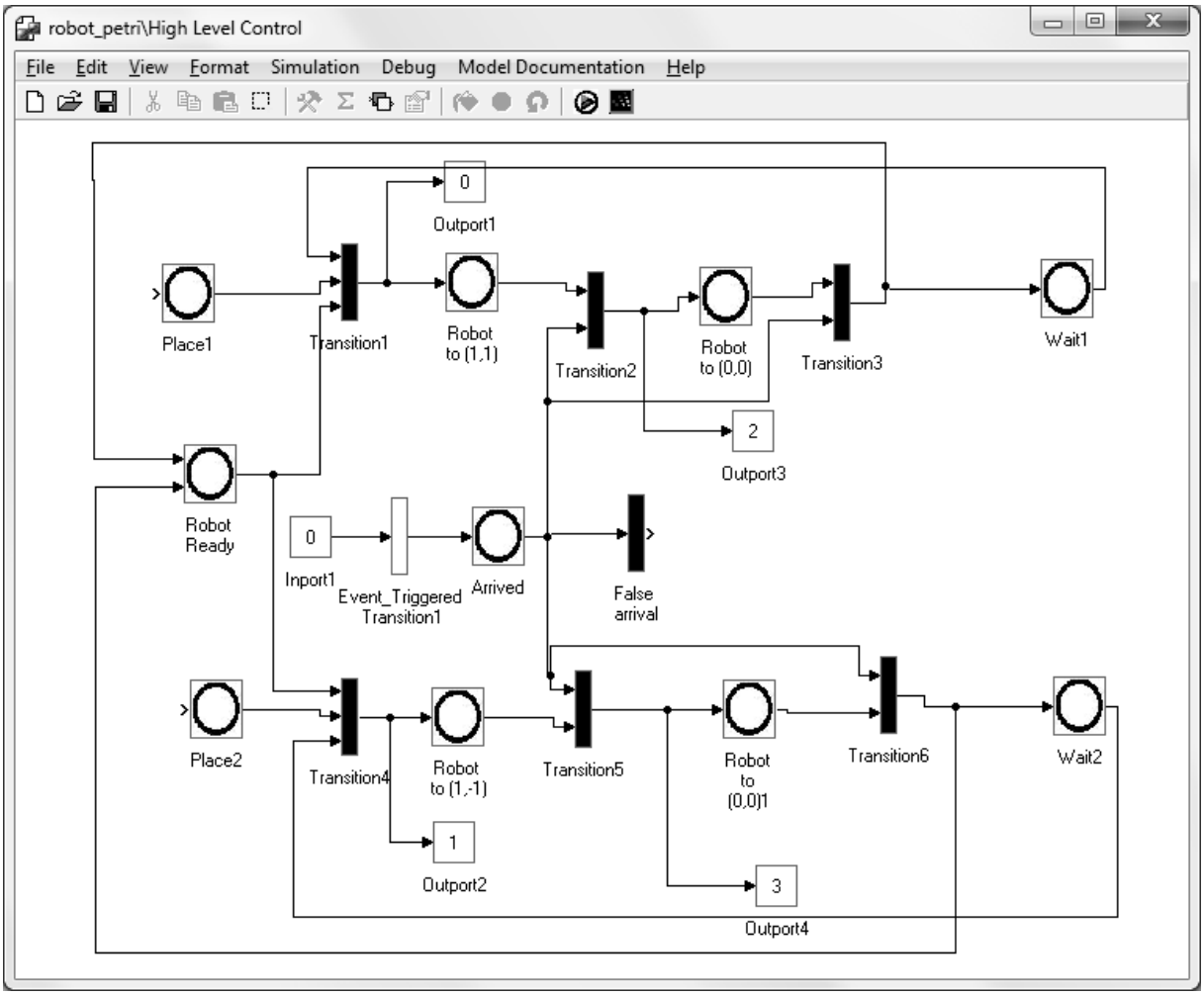

Figura 5.6: Control de alto nivel implementado en Petri.

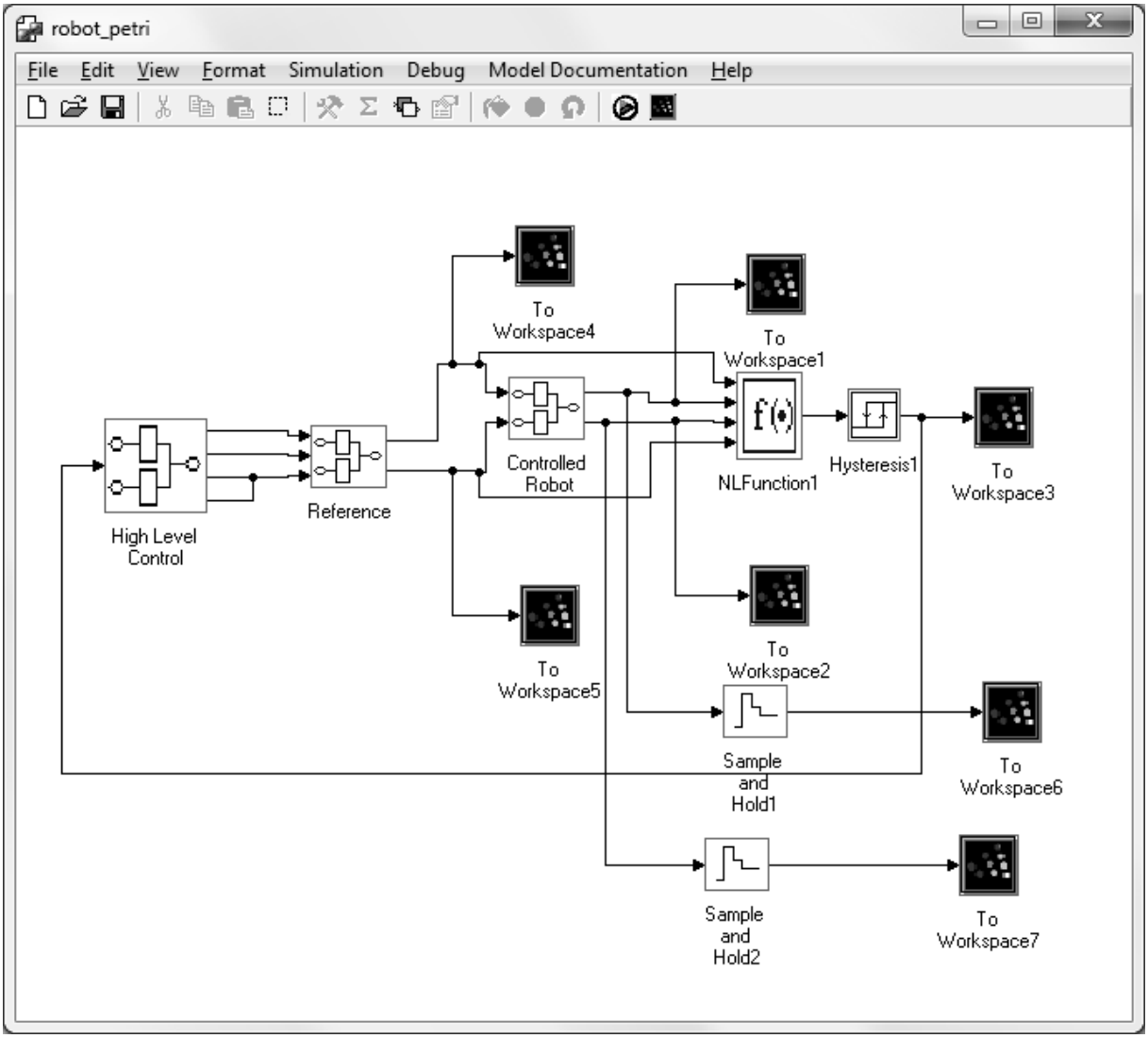

Figura 5.7: Modelo Híbrido del Robot con su control.

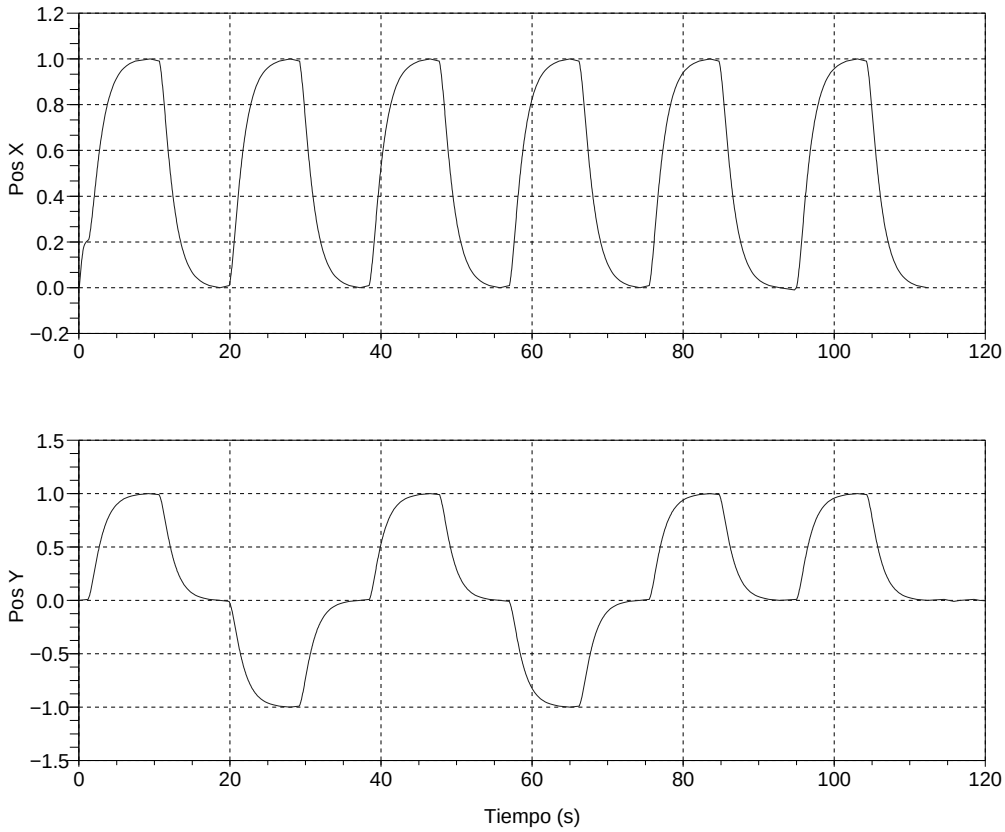

Figura 5.8: Simulación del Robot controlado por una red de Petri.

Capítulo 6

Conclusiones

En el presente trabajo se desarrolló una nueva librería para la simulación de redes de Petri dentro de un simulador de sistemas de eventos discretos (DEVS), llamado PowerDEVS. Esta librería puede ser utilizada para la realización de modelos híbridos, aumentando el poder expresivo de PowerDEVS.

Se analizaron las distintas implementaciones que marcaron el trayecto a la librería final. En cada versión se detalló sus principales características, ventajas y desventajas, como también el comportamiento de los distintos modelos que la conforman. Además, se plantearon las dificultades obtenidas en cada una de ellas con sus posibles soluciones.

Mediante varios ejemplos de aplicación, se verificó el funcionamiento de la librería y su aplicabilidad. Entre los ejemplos se encuentra el problema de los filósofos comensales, el protocolo de comunicación Parada y Espera y un ejemplo híbrido donde se implementó un control supervisor a través de redes de Petri y una simulación del sistema continuo (robot y control) por medio de aproximaciones numéricas DEVS.

La forma habitual de mostrar la evolución de una Red de Petri, es *animando* la red, incluyendo al cantidad de marcas en cada lugar y los disparos de cada transición. Una de las tareas pendientes por realizar, es encontrar una manera más amigable para visualizar el resultado de la simulación de una Red de Petri ya que en esta implementación se realiza a través de gráficas y archivos de texto.

La nueva librería desarrollada (junto con más ejemplos de aplicación) está incluida en la distribución estándar de la herramienta PowerDEVS. Su uso y distribución es libre, y está licenciado

bajo la licencia GPL.

Los resultados de esta tesina se encuentran publicados en [13].

Para más información `https://sourceforge.net/projects/powerdevs/`.

Apéndice A

Código: Primer Enfoque

Lugar

Código A.1 Lugar: State variables and parameters.

```
//Declare the state, output variables and parameters
double Sigma;
int tokens, nin, m;
double time, inf;
typedef struct sms{
double y[10];
Simulator *src;
int tokens;} Message;
Message *message;
```

Código A.2 Lugar: Init.

```
va_list parameters;
va_start(parameters,t);
m = (int) va_arg(parameters,double);
nin = (int) va_arg(parameters,double);
inf = 1e20;
tokens = m;
if (m > 0)
 Sigma = 0;
else
 Sigma = inf;
message = new Message;
```

Código A.3 Lugar: Time advance.

```
return Sigma;
```

Código A.4 Lugar: Internal transition.

```
Sigma = inf;
```

Código A.5 Lugar: External transition.

```
Message* mens = (Message*)x.value;
if (x.port == 0)
 tokens--;
else
 tokens++;

Sigma = 0;
```

Código A.6 Lugar: Output.

```
message -> src = this;
message -> tokens = tokens;
return Event(message,0);
```

Transición

Código A.7 Transición: State variables and parameters.

```
// Declare the state, output variables and parameters
double Sigma;
double time, inf;
bool enabled, working;
int nin, nout, lastack;
DataBase *data;
Message *message;
int SigmaFlag;
```

Código A.8 Transición: Init.

```
va_list parameters;
va_start(parameters,t);
time = va_arg(parameters,double);
nin = (int) va_arg(parameters,double);
nout = (int) va_arg(parameters,double);
inf = 1e20;
SigmaFlag = 1;
enabled = false;  //the transition initializes unabled
message = new Message;
data = new DataBase(nin);
if (nin== 0){  //source transition
 Sigma = time;
 working = true;
 lastack = 1;
}
else{
 working = false;
 Sigma = inf;
}
```

Código A.9 Transición: Time advance.

```
return Sigma;
```

Código A.10 Transición: Internal transition.

```
if (SigmaFlag == 1)
 Sigma = inf;
else
 Sigma = 0;
```

Código A.11 Transición: External transition.

```
Message* mens = (Message*)x.value;
bool active;
data -> Update(x.port,mens); // update the DataBase
active = data -> Check();
if (active){ // is the transition active?
 if (enabled){ // the transition is already activated
  Sigma = Sigma -e;
 else if (working) // the transition has to send more message yet
  Sigma = 0;
 else{ // the transition is activate now
  Sigma = time;
  enabled = true;
 }
 SigmaFlag = 0;
}
else if (working)
 Sigma = 0;
else{  // Finish
 enabled = false;
 Sigma = inf;
}
```

Código A.12 Transición: Output.

```
PlaceWeight *datos;
if (enabled){
 enabled = false;
 working = true;
 data -> Clean();
 datos = data -> GetPlace();
 message -> src = datos -> src;
 message -> tokens =  datos -> peso;
 lastack = nout-1;
 return Event(message,lastack);
}
else if (working && lastack >1){
 datos = data -> GetPlace();     // get the place for
 message -> src = datos -> src;  // the tokens extraction process
 message -> tokens =  datos -> peso;
 lastack--;
 return Event(message,lastack); // extract one token of the place
}                               // connected to the lastack
else if (working){
 working = false;
 SigmaFlag = 1;
 lastack--;
 message-> src = this;
 message -> tokens =  0;
 return Event(message,0); // here we send the forward message
}
```

Apéndice B

Código: Tercer Enfoque

Lugar

Código B.1 Lugar: State variables and parameters.

```
// Declare the state, output variables and parameters
double Sigma, auxSigma;
int tokens, nin, m;
double time, inf;
Message *message;
LinkedList *list;
int TimeFlag, auxTimeFlag;
Simulator *transition;
```

Código B.2 Lugar: Time advance.

```
return Sigma;
```

Código B.3 Lugar: Init.

```
va_list parameters;     // The 'parameters' variable contains the
va_start(parameters,t); // parameters transferred from the editor
m = (int) va_arg(parameters,double); //Initials tokens
time = va_arg(parameters,double); //Waiting time
nin = (int) va_arg(parameters,double);
inf = 1e20; // this number represents infinite
tokens = 0;
message = new Message;
list = new LinkedList();  //list of (token,time)
if (m > 0){
 Sigma = time;
 TimeFlag = 1;
 for (int i=1;i<=m;i++)
  list -> Insert(message,time);
}
else{
 Sigma = inf;
 TimeFlag = 0; //no waiting tokens on the list
}
```

Código B.4 Lugar: Internal transition.

```
int length = list -> GetLength();
if (TimeFlag == 3){
 transition -> externalinput(t,(int)message);
 TimeFlag = auxTimeFlag;
 Sigma = auxSigma;
}
else if (TimeFlag == 1){ //time fulfilled by one or more tokens
 list -> UpdateTime (list -> GetLessTime());
 tokens = tokens + list -> CheckMessages(); // get tokens out
 Sigma = 0;                                 // of the list
 TimeFlag = 4;
}
else if (TimeFlag == 4){
 if (length == 0){
  TimeFlag = 0;
  Sigma = inf;
 }
 else{
  TimeFlag = 1;
  Sigma = list -> GetLessTime();
 }
}
else if (TimeFlag == 2){ // if some transition erase tokens
  Sigma = auxSigma;      // of the place
  TimeFlag = auxTimeFlag;
 }
else  //TimeFlag == 0
 Sigma = inf;
```

Código B.5 Lugar: External transition.

```
Message* mens = (Message*)x.value;
list -> UpdateTime(e); // update the time of the tokens on the list
if (x.port == -1){     // virtual connection
 if (mens ->src == this){
  tokens = tokens - mens -> tokens;
  auxSigma = Sigma - e;
  auxTimeFlag = TimeFlag;
  Sigma = 0;
  TimeFlag = 2;
 }
 else if (TimeFlag == 1 || TimeFlag == 0){ // transition's Request
  transition = mens ->src2;  // transition's ID
  auxSigma = Sigma - e;
  auxTimeFlag = TimeFlag;
  Sigma = 0;
  TimeFlag = 3;
 }
 else  //it has already started the hand-Shake with other transition
  Sigma = Sigma - e;
}
else{
 list -> Insert(mens,time);
 if (Sigma > time){  //Sigma = inf in the beginning
  Sigma = time;
  TimeFlag = 1;
 }
 else
  Sigma = Sigma - e;
}
```

Código B.6 Lugar: Output.

```
if (TimeFlag == 3)
 return Event();

if (TimeFlag == 1) //time fulfilled by one or more tokens on the list
 return Event();

if (TimeFlag == 4){
 message -> y[0] = tokens;
 message -> src = this;
 message -> src2 = this;
 message -> tokens = tokens;
 return Event(message,0);
}

if (TimeFlag == 2){
 message -> y[0] = tokens;
 message -> src = this;
 message -> src2 = this;
 message -> tokens = tokens;
 return Event(message,0);
}
```

Transición

Código B.7 Transición: State variables and parameters

```
// Declare the state, output variables and parameters
double Sigma;
double time, inf;
bool enabled;
int nin, numacks;
DataBase *data;
Message *message;
int HandShake;
Simulator *predecessor;
/*
HandShake = 0  Hand-Shake has to be done
HandShake = 1  Hand-Shake is doing now
HandShake = 2  Hand-Shake finished
HandShake = 3  Hand-Shake doesn't have to be done (source transitions)
*/
```

Código B.8 Transición: Init.

```
va_list parameters;
va_start(parameters,t);
time = va_arg(parameters,double);
nin = (int) va_arg(parameters,double);
inf = 1e20;
enabled = false;
numacks = 0;
message = new Message;
message -> y[0] = 0;
data = new DataBase(nin);
if (nin == 0){ // source transition
 HandShake = 3;
 Sigma = time;
}
else{
 HandShake = 0;
 Sigma = inf;
}
```

Código B.9 Transición: Time advance.

```
return Sigma;
```

Código B.10 Transición: Internal transition.

```
PlaceWeight *datos;
 if (enabled){
  enabled = false;
  HandShake = 1;
  data -> Clean();
  Sigma = inf;
  for (int i=0;i<data -> GetTotalPlaces(); i++){
   datos = data -> GetPlace();  // get the predecessor place
   predecessor = datos -> src;
   message -> src = this;       //Request message
   message -> src2 = this;
   message -> tokens =  datos -> peso;
   predecessor -> externalinput(t,(int)message);
  }
 }
 else if (HandShake == 2){
  enabled = false;
  HandShake = 0;
  data -> Clean();
  Sigma = inf;
  for (int i=0;i<data -> GetTotalPlaces(); i++){
   datos = data -> GetPlace();
   predecessor = datos -> src;
   message -> src = datos -> src;
   message -> tokens =  datos -> peso;
   predecessor -> externalinput(t,(int)message);
  }
 }
 else // (HandShake == 3) Only for source transition
  Sigma = time;
```

Código B.11 Transición: External transition.

```
Message* mens = (Message*)x.value;
if (x.port == -1){  // Hand-Shake ack
 numacks ++;
 if (numacks == data -> GetTotalPlaces()){ // true: all the
  Sigma = 0;                               // places answer
  HandShake = 2;
  numacks = 0;
 }
 else
  Sigma = inf; //wait the rest of akcs
}
else{
 bool active;
 data -> Update(x.port,mens);
 active = data -> Check();
 if(active){
  if (enabled)
   Sigma = Sigma - e;
  else if (HandShake == 1){ //the Hand-Shake was interrupt
   enabled = true;
   Sigma = 0;
   numacks = 0;
  }
```

Código B.12 Transición: External transition (continuación).

```
  else if (HandShake == 2)
   Sigma = 0;
  else{
   Sigma = time;
   enabled = true;
   numacks = 0;
  }
 }
 else if (HandShake == 1)
  Sigma = inf;
 else if (HandShake == 2)
  Sigma = 0;
 else{
  enabled = false;
  Sigma = inf;
 }
}
```

Código B.13 Transición: Output.

```
if (enabled) //it is ready to transition
 return Event();

else if (HandShake == 2){ // beginning to send messages
 message-> src = this;
 message -> tokens = 0;
 return Event(message,0);
}

else if (HandShake == 3){ // only for source transitions
 message-> src = this;
 message -> tokens = 0;
 return Event(message,0);
}
```

Apéndice C

Código: Enfoque Final

Lugar

Código C.1 Lugar: State variables and parameters.

```
// Declare the state, output variables and parameters
double Sigma;
int nin, m, tokens;
double time, inf;
Message *message;
LinkedList *list;
int TimeFlag;
Simulator *transition;
```

Código C.2 Lugar: Init.

```
va_list parameters;     // The 'parameters' variable contains
va_start(parameters,t); // the parameters transferred from the editor.
char *fvar;
fvar = va_arg(parameters, char*);
m = getScilabVar(fvar); //initial tokens
fvar = va_arg(parameters, char*);
time = getScilabVar(fvar);
nin = (int)va_arg(parameters,double);
inf = 1e20; //this number represent infinite
tokens = 0;
message = new Message;
list = new LinkedList(); // List that has messages with their times
if (m > 0){              // in order of appearing
 Sigma = time;
 TimeFlag = 1;
 for (int i=1;i<=m;i++)
  list -> Insert(message,time);
}
else{
 Sigma = inf;
 TimeFlag = 0; //No messages waiting on the list
}
```

Código C.3 Lugar: Time advance.

```
return Sigma;
```

Código C.4 Lugar: Internal transition.

```
int length = list -> GetLength();
if (TimeFlag == 1){ //if the token's time is zero enter here
 int auxtokens;
 list -> UpdateTime (list -> GetLessTime());
 auxtokens = tokens;
 tokens = tokens + list -> CheckMessages(); // take out the tokens
 Sigma = 0;                                 // of the Linked List
 TimeFlag = 4;
}
else if (TimeFlag == 4){
 if (length == 0){
  TimeFlag = 0;
  Sigma = inf;
 }
 else{
  TimeFlag = 1;
  Sigma = list -> GetLessTime();
 }
}
```

Código C.5 Lugar: External transition.

```
Message* mens = (Message*)x.value;
list -> UpdateTime(e); // Update the time of Linked List always
if ( x.port == -1){
 Sigma = 0;
 TimeFlag = 4;
}
else{
 list -> Insert(mens,time);
 if (Sigma > time){ // Sigma = inf when it is the first message on
  Sigma = time;     // arrived
  TimeFlag = 1;
 }
 else
  Sigma = Sigma - e;
}
```

Código C.6 Lugar: Output.

```
if (TimeFlag == 1){ // Are the tokens's time on the list zero?
 return Event();
else if (TimeFlag == 4){
 message -> y[0] = tokens;
 message -> src = this;
 message -> tokens = tokens;
 return Event(message,0);
}
```

Código C.7 Lugar: Exit.

```
delete message;
delete list;
```

Transición

Código C.8 Transición: State variables and parameters.

```
// Declare the state, output
// variables and parameters
double Sigma;
double time, inf;
bool enabled, source, working, portsinit;
int nin;
DataBase *data;
Message *message;
place *predecessor;
```

Código C.9 Transición: Init.

```
va_list parameters;
va_start(parameters,t);
char *fvar;
fvar = va_arg(parameters, char*);
time = getScilabVar(fvar);
nin = (int) va_arg(parameters,double); // number of input ports
inf = 1e20;
enabled = false;
working = false;
message = new Message;
portsinit = false;
if (nin == 0){ // source transitions only ought to have one input port
 source = true;
 Sigma = time;
}
else{
 data = new DataBase(nin);
 source = false;
 Sigma = inf;
}
```

Código C.10 Transición: Time advance.

```
return Sigma;
```

Código C.11 Transición: Internal transition.

```
PlaceWeight *datos;
int weight;
if(enabled){
 enabled = false;
 data -> Clean(); // erase the messages received and assing last = 0
 Sigma = inf;
 int j = 0;
 for (int i=0;i<data -> GetTotalPlaces(); i++){
  datos = data -> GetPlace();
  predecessor = (place*)datos -> src; //pointer to predecessor place
  weight = datos -> weight;
  if (weight <= predecessor -> GetTokens())
   j++;
 }
 data -> Clean();
 if(j == data -> GetTotalPlaces()){
  data -> Clean();
  for (int i=0;i<data -> GetTotalPlaces(); i++){
   datos = data -> GetPlace();
   predecessor = (place*)datos -> src;
   weight = datos -> weight;
   predecessor -> SetTokens(predecessor -> GetTokens() - weight);
  }
  Sigma = 0;
  working = true;
 }
 else
  Sigma = inf; // the transition has been deactivated
}
```

```
else if(working){
 working = false;
 data -> Clean();
 Sigma = inf;
 for (int i=0;i<data -> GetTotalPlaces(); i++){
  datos = data -> GetPlace();
  predecessor = (place*)datos -> src;
  predecessor -> externalinput(t,1);
 }
}
else if(source)
  Sigma = time;
else
 Sigma = inf;
```

Código C.12 Transición: External transition.

```
Message* mens = (Message*)x.value;
bool active = false;
data -> Update(x.port,mens); // update the DataBase with (port,message)
if(portsinit == false && data->CalculateTotalPlaces())
 portsinit = true;
if(portsinit){
 data -> Update2(x.port,mens);
 active = data -> Check(); // Is the transition activated?
}
if(active){
 if (enabled) // the transition has already activated
  Sigma = Sigma - e;
 else{ // now, the transition is activated
  Sigma = time;
  enabled = true;
 }
}
else{ // the transition is not activated
 enabled = false;
 Sigma = inf;
}
```

Código C.13 Transición: Output.

```
if(enabled)
 return Event();  // inform that it is ready for transitioning
else if(working){ // beginning to send add and subtract messages
 message -> y[0] = 0;
 message-> src = this;
 message -> tokens = 0;
 return Event(message,0); // the subtract message is sent by the port 0
}

else if(source){ // only for source transitions
 message -> y[0] = 0;
 message -> src = this;
 message -> tokens = 0;
 return Event(message,0); // the add message is sent
}                         // by the port 0
```

Código C.14 Transición: Exit.

```
if (source == false) // if the transition is activated and it is not a
 data -> Remove();   // source transition, the simulation finish
if (nin != 0)
 delete data;
delete message;
```

Transición Híbrida

Código C.15 Transición Híbrida: State variables and parameters.

```
// Declare the state, output
// variables and parameters
double Sigma;
double inf;
Message *message;
void *thing;
```

Código C.16 Transición Híbrida: Init.

```
inf = 1e20;
message = new Message;
message -> y[0] = 0;
Sigma = inf;
```

Código C.17 Transición Híbrida: Time advance.

```
return Sigma;
```

Código C.18 Transición Híbrida: Internal transition.

```
Sigma = inf;
```

Código C.19 Transición Híbrida: External transition.

```
thing = x.value;
Sigma = 0;
```

Código C.20 Transición Híbrida: Output.

```
message-> src = this; // this field is not important
message -> tokens = 0; // this field is not important
return Event(message,0); // the message is sent by the port 0
```

Código C.21 Transición Híbrida: Exit.

```
delete message;
```

Apéndice D

Código: Structures, database y linkedlist

Código D.1 structures.h

```
#include "simulator.h"
#if !defined ESTRUCTURAS
#define ESTRUCTURAS

typedef struct sms{
        double y[10];
        Simulator *src;
        int tokens;
        }Message;

typedef struct node{
        Message *message;
        double time;
        struct node *next;
        } Node;

typedef struct placeweight{
        Simulator *src;
        int tokens;
        int weight;
        }PlaceWeight;
#endif
```

Código D.2 database.h

```
#include "structures.h"
#if !defined DATABASE
#define DATABASE

class DataBase{
 public:
  DataBase(int n);
  ~DataBase();
  void Update(int port, Message *item);
  void Update2 (int port, Message *item);
  bool CalculateTotalPlaces();
  int GetTotalPlaces();
  PlaceWeight* GetPlace();
  void Clean();
  void Remove();
  bool Check();

 private:
  int length; // size of the array "ports" (number of input ports)
  Message** ports;
  PlaceWeight** places;
  int last; //position of the last place sent in the array "places"
  int totalPlaces; //Total of precedent places
};
#endif
```

Código D.3 database.cpp

```
#include "database.h"

DataBase::DataBase(int n){
 ports = new Message* [n];
 places = new PlaceWeight* [n];
 length = n;
 for(int i=0;i<length;i++){
   ports[i] = NULL;
   places[i] = NULL;
 }
 last = 0;
 totalPlaces = 0;
}

DataBase::~DataBase(){
 delete []ports;
 delete []places;
}

void DataBase::Update(int port,Message *item){ // each array position
 ports[port] = item;                      // represent one input port
}
```

```
bool DataBase::CalculateTotalPlaces(){
 int i, j, weight;
 bool visto;
 for (i=0;i<length;i++)
   if (ports[i] == NULL) // Are all the input ports accupied
    return false;

 int cant_lug = 0; // number of precedent places

 for (i=0;i<length;i++){
  visto = false;
  for (j=0;j<cant_lug;j++) // Has already counted this place?
   if (ports[i]->src == places[j]->src)
     visto = true;

if (visto == false){ // if the place was'nt counted, enter
   weight = 0; // the compared place is counted in the comparison
   for (j=i;j<length;j++)
    if(ports[i]->src == ports[j]->src)
     weight++;
  places[cant_lug] = new PlaceWeight;
  places[cant_lug] -> src = ports[i] -> src;
  places[cant_lug] -> tokens = ports[i] -> tokens;
  places[cant_lug] -> weight = weight;
  cant_lug++;
  }
 }
 totalPlaces = cant_lug;
 return true;
}
```

```
void DataBase::Update2 (int port, Message *item){
 for (int i=0;i<totalPlaces;i++)
  if (places[i] -> src == item -> src){
   places[i] -> tokens = item -> tokens;
   return;
  }
}

bool DataBase::Check(){
 int i, valida = 0;
 for (i=0;i<totalPlaces;i++)
  if (places[i]->weight <= places[i]->tokens)
   valida++;

 if(valida == totalPlaces)
  return true;
 else
  return false;
}

/* Get the next record (Placesweight) to be sent*/
PlaceWeight* DataBase::GetPlace(){
 if (places[last]!= NULL){
  last++;
  return places[last-1];
 }
 return NULL;
}
```

```
/* Clean all the entries of the array */
void DataBase::Clean(){
 last = 0;
 for(int i=0;i<length;i++)
  ports[i] = NULL;
}

int DataBase::GetTotalPlaces(){
 return totalPlaces;
}

void DataBase::Remove(){
  for (int i=0;i<totalPlaces;i++)
   delete places[i];
}
```

Código D.4 linkedlist.h

```
#include "structures.h"
#if !defined LINKEDLIST
#define LINKEDLIST

class LinkedList{
 public:
  LinkedList();
  ~LinkedList();
  bool IsEmpty();
  void Insert (Message *m, double t);
  Message* Remove();
  int CheckMessages();       // check how many messages fulfilled
  void UpdateTime(double t); // their time and then erase them
  int GetLength();
  double GetLessTime();

 private:
  Node *first;
  Node *last;
  int length;
};
#endif
```

Código D.5 linkedlist.cpp

```
#include "linkedlist.h"

LinkedList::LinkedList(){
 first = last = NULL;
 length = 0;
}

LinkedList::~LinkedList(){
 while (Remove() != NULL);
}

bool LinkedList::IsEmpty(){
 if (first == NULL)
  return true;
 else
  return false;
}

void LinkedList::Insert (Message *m, double t){
 length++;
 Node *nuevo = new Node;
 nuevo -> message = m;
 nuevo -> time = t;
 nuevo -> next = NULL;

 if (IsEmpty()){
  first = nuevo;
  last = nuevo;
 }
 else{
  last -> next = nuevo;
  last = nuevo;
 }
}
```

```
Message* LinkedList::Remove(){
 Node *element = first;
 Message* thing;
 if (IsEmpty())
  return NULL;

 length--;
 thing = first -> message;

 if (first == last){
  first = NULL;
  last = NULL;
 }
 else{
  first = element -> next;
 }

 delete element;
 return thing;
}
```

```
int LinkedList::CheckMessages(){
 int contador = 0;

 if (IsEmpty())
  return 0;

 if (first == last){
  Remove();
  return counter+1;
 }

 while (first->time <= 0){
  Remove();
  counter++;
  if (first == last && first->time <= 0){
   Remove();
   return counter+1;
  }
 }
 return counter;
}
```

```
void LinkedList::UpdateTime(double t){
 if (IsEmpty() == false){
  Node *ptr;
  for (ptr = first; ptr != NULL; ptr = ptr->next){
   if (ptr -> time >= t)
    ptr -> time = ptr -> time - t;
   else
    ptr -> time = 0;
  }
 }
}

int LinkedList::GetLength(){
 return length;
}

double LinkedList::GetLessTime(){
 if (length == 0)
  return 1e20; // if the list is empty return infinite
 else
  return first -> time;
}
```

Bibliografía

[1] F. Bergero and E. Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 2010. in Press.

[2] F. Bergero, E. Kofman, C. Basabilbaso, and J. Zúccolo. Desarrollo de un simulador de sistemas híbridos en tiempo real. In *Proceedings of AADECA 2008*, Buenos Aires, Argentina, 2008.

[3] P. Bonet, C.M. Llado, R. Puijaner, and W.J. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, San Jose, Costa Rica, 2007.

[4] J. Byg, K.Y. Joergensen, and J. Srba. TAPAAL: Editor, simulator and verifier of timed-arc Petri nets. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799 of *LNCS*, pages 84–89. Springer-Verlag, 2009.

[5] Francois E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, New York, 2006.

[6] Guy Cohen. *Análisis y control de sistemas de eventos discretos: de redes de Petri temporizadas al álgebra, segunda edición*. UNR EDITORA, Argentina, 2003. Cuadernos del instituto de mátematica "BEPPO LEVI".

[7] DEVSJava. `http://www.acims.arizona.edu/SOFTWARE/devsjava3.0/setupGuide.html`.

[8] C. Jacques and G. Wainer. Using the CD++ DEVS Tookit to Develop Petri Nets. In *Proceedings of the Summer Computer Simulation Conference*, 2002.

[9] Ernesto Kofman. Introducción a la modelización y simulación de sistemas de eventos discretos con el formalismo DEVS. Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Argentina, 2003.

[10] Pieter J. Mosterman, Martin Otter, and Hilding Elmqvist. Modeling Petri Nets as local constraint equations for hybrid systems using Modelica. In *Proceedings of the Summer Computer Simulation Conference*, pages 314–319, 1998.

[11] Jim Nutaro. *ADEVS (A Discrete EVent System simulator)*. Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson, AZ.

[12] C. Panayiotou. *PetriNets and Other Models*. Dep. of ECE, University of Cyprus, 2006. Lecture notes for Performance Evaluation and Simulation class, Available at `http://www.eng.ucy.ac.cy/Christos/Courses/ECE658/Lectures/PetriNets.pdf`.

[13] Taihú Pire, Federico Bergero, and Ernesto Kofman. Modelado y simulación de redes de Petri con el formalismo DEVS. *AADECA 2010 - Semana del Control Automático - XXII° Congreso Argentino de Control Automático*, Agosto 2010.

[14] Lawrence L. Rose. The Development of HP-SIM: Hierarchical Process-Oriented Simulation Software. Technical Report 86(1), University of Pittsburgh, 1986. 19 pages.

[15] A. Silberschatz and J. L. Peterson, editors. *Operating systems concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[16] Simulaworks, System Simulator. `http://stuffmate.com/download/simulaworks_system_simulator/`.

[17] Kim Tag Gon, Praehofer Herbert, and Bernard Zeigler. *Theory of Modeling and Simulation*. John Wiley & Sons, second edition, New York, 1976.

[18] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[19] Gabriel Wainer. Cd++: a toolkit to develop devs models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[20] Petri Nets World. `http://www.informatik.uni-hamburg.de/TGI/PetriNets`.

[21] Martina Svdov Zdenk, Martina Svádová, and Zdenek Hanzálek. Matlab Toolbox For Petri Nets. In *22nd International Conference ICATPN 2001*, pages 32–36, 2001.

[22] Bernard Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation.* Academic Press, New York, second edition, 2000.