MODELADO DE SISTEMAS MEDIANTE DEVS

Teoría y Práctica

Texto base de la asignatura
"Modelado de Sistemas Discretos"
de la Ingeniería en Informática de la UNED

Alfonso Urquía

Departamento de Informática y Automática
Escuela Técnica Superior de Ingeniería Informática
Universidad Nacional de Educación a Distancia (UNED)
Juan del Rosal 16, 28040, Madrid, España
aurquia@dia.uned.es
http://www.euclides.dia.uned.es

ÍNDICE

Tema 1

INTRODUCCIÓN AL MODELADO Y LA SIMULACIÓN

OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema debería saber:

- Discutir los conceptos “sistema”, “modelo” y “simulación”.
- Discutir en qué situaciones el modelado y la simulación son técnicas adecuadas para el estudio de sistemas.
- Describir y comparar los diferentes tipos de modelos.
- Comparar y reconocer los distintos tipos de modelos matemáticos.
- Discutir los niveles en el conocimiento de los sistemas de la clasificación de Klir, y su relación con los conceptos “análisis”, “inferencia” y “diseño” de sistemas.
- Discutir los niveles en la especificación de sistemas (clasificación de Zeigler).
- Discutir las entidades y las relaciones básicas del marco formal para el modelado y la simulación.

1.1. INTRODUCCIÓN

En este primer tema se definen conceptos fundamentales en el ámbito del modelado y la simulación, como son "modelo", "sistema", "experimento" y "simulación". Se describirán cuatro tipos diferentes de modelos, uno de los cuales es el empleado en simulación: el modelo matemático. También se discutirán varias clasificaciones de los modelos matemáticos, realizadas atendiendo a diferentes criterios.

Se describirá una clasificación en cuatro niveles del conocimiento acerca de un sistema, a partir de la cual se definirán tres actividades relacionadas con los sistemas: el análisis, la inferencia y el diseño.

Finalmente, se planteará una clasificación de los niveles en la descripción de un sistema, en el ámbito del modelado y la simulación. Esta discusión motivará la explicación de conceptos tales como "modularidad" y "jerarquía" en la descripción de los modelos, así como la descripción de las entidades y relaciones básicas del modelado y la simulación.

1.2. SISTEMAS, MODELOS Y EXPERIMENTACIÓN

En esta sección se definen los términos "modelo" y sistema". Asimismo, se muestran dos formas de estudiar los sistemas. La primera es mediante experimentación con el sistema real. La segunda es mediante la realización de un modelo del sistema real y la experimentación con el modelo. Veremos que, en ocasiones, esta segunda forma es la única posible.

1.2.1. Definición de los términos "modelo" y "sistema"

En el sentido amplio del término, puede considerarse que un **modelo** es *una representación de un sistema desarrollada para un propósito específico.*

Puesto que la finalidad de un modelo es ayudarnos a responder preguntas sobre un determinado sistema, el primer paso en la construcción de un modelo debería ser definir cuál es el sistema y cuáles son las preguntas.

En este contexto, se entiende por **sistema** *cualquier objeto o conjunto de objetos cuyas propiedades se desean estudiar.*

Con una definición tan amplia, *cualquier fuente potencial de datos* puede considerarse un sistema. Algunos ejemplos de sistema son:

- Una planta de fabricación con máquinas, personal, dispositivos de transporte y almacén.
- El servicio de emergencias de un hospital, incluyendo al personal, las salas, el equipamiento y el transporte de los pacientes.
- Una red de ordenadores con servidores, clientes, dispositivos de disco, impresoras, etc.
- Un supermercado con control de inventario, cajeros y atención al cliente.
- Un parque temático con atracciones, tiendas, restaurantes, trabajadores, clientes y aparcamientos.

1.2.2. Experimentación con el sistema real

Un procedimiento para conocer el comportamiento de los sistemas es la experimentación. De hecho, éste ha sido el método empleado durante siglos para avanzar en el conocimiento: plantear las preguntas adecuadas acerca del comportamiento de los sistemas y responderlas mediante experimentación.

Puede considerarse que un **experimento** es *el proceso de extraer datos de un sistema sobre el cual se ha ejercido una acción externa.*

Por ejemplo, el encargado de un supermercado puede ensayar diferentes procedimientos de control del inventario y de distribución del personal, para determinar qué combinaciones demuestran ser más rentables y proporcionan un mejor servicio.

Cuando es posible trabajar directamente con el sistema real, el método experimental presenta indudables ventajas. Sin embargo, para que los resultados del experimento sean válidos, debe garantizarse que no existen variables ocultas "confundidas" con las variables experimentales. Por ejemplo, continuando con el modelo del supermercado, si la afluencia de público durante el periodo en que se experimenta una estrategia de servicio es significativamente mayor que durante el periodo en que se experimenta la otra, no podrá extraerse ninguna conclusión válida sobre el tiempo medio de espera de los clientes en la cola de las cajas.

1.2.3. Experimentación con el modelo

El *método experimental* está basado en sólidos fundamentos científicos. Sin embargo, tiene sus limitaciones, ya que en ocasiones es imposible o desaconsejable experimentar con el sistema real. En estos casos, el modelado y la simulación son las técnicas adecuadas para el análisis del sistema, puesto que, *a excepción de la experimentación con el sistema real, la simulación es la única técnica disponible que permite analizar sistemas arbitrarios de forma precisa, bajo diferentes condiciones experimentales.*

Existen diversas razones por las cuales la experimentación con el sistema real puede resultar inviable. Algunas de estas razones son las siguientes:

- Quizá la más evidente de ellas es que el sistema aun no exista físicamente. Esta situación se plantea frecuentemente en la fase de diseño de nuevos sistemas, cuando el ingeniero necesita predecir el comportamiento de los mismos antes de que sean construidos.
- Otra posible razón es el elevado coste económico del experimento. Consideremos el caso de un empresario que planea ampliar las instalaciones de su empresa, pero que no está seguro de que la ganancia potencial justifique el coste de la ampliación. Ciertamente, no sería razonable que, para salir de dudas, realizara la ampliación y luego se volviera atrás si ésta demostrara no ser rentable. Una alternativa consiste en simular la operación de la configuración actual de la empresa, simular la operación de la configuración alternativa y realizar una comparación de los resultados económicos de ambas.
- El experimento puede producir perjuicio o incomodidad. Por ejemplo, experimentar con un nuevo sistema de facturación en un aeropuerto puede producir retrasos y problemas imprevisibles que perjudiquen al viajero.
- En ocasiones el tiempo requerido para la realización del experimento lo hace irrealizable. Casos extremos pueden encontrarse en los estudios geológicos o cosmológicos, de evolución de las especies, sociológicos, etc.
- Algunos experimentos son peligrosos, y por tanto es desaconsejable realizarlos. Por ejemplo, sería peligroso usar el sistema real para adiestrar a los operarios de una central nuclear acerca de cómo deben reaccionar ante situaciones de emergencia.

– En ocasiones el experimento requiere modificar variables que en el sistema real o bien no están accesibles, o bien no pueden ser modificadas en el rango requerido. Con un modelo matemático adecuado, se pueden ensayar condiciones de operación extremas que son impracticables en el sistema real.

1.3. TIPOS DE MODELOS

Tal como se ha explicado en la sección anterior, una forma de estudiar los sistemas consiste en experimentar directamente con ellos. Otra forma es realizar un modelo del sistema y emplear el modelo para extraer conclusiones acerca del comportamiento del sistema. Como se explicará a continuación, pueden distinguirse cuatro tipos de modelo (véase la Figura 1.1): mental, verbal, físico y matemático.

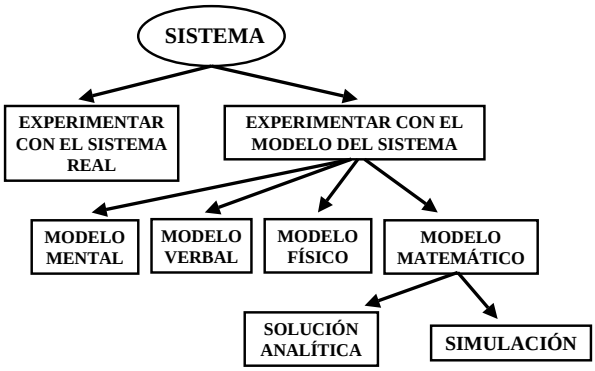

Figura 1.1: Formas de estudiar un sistema.

1.3.1. Modelos mentales

En nuestra vida cotidiana empleamos continuamente *modelos mentales* para comprender y predecir el comportamiento de los sistemas. Por ejemplo, considerar que alguien es "amable" constituye un modelo del comportamiento de esa persona. Este modelo nos ayuda a responder, por ejemplo, a la pregunta de cómo reaccionará si le pedimos un favor.

También disponemos de modelos de los sistemas técnicos, que están basados en la intuición y en la experiencia. Por ejemplo, aprender a conducir un coche consiste parcialmente en desarrollar un modelo mental de las propiedades de la conducción

del coche. Asimismo, un operario trabajando en determinado proceso industrial sabe cómo el proceso reacciona ante diferentes acciones: el operario, mediante el entrenamiento y la experiencia, ha desarrollado un modelo mental del proceso.

1.3.2. Modelos verbales

Otro tipo de modelos son los *modelos verbales*, en los cuales el comportamiento del sistema es descrito mediante palabras: si se aprieta el freno, entonces la velocidad del coche se reduce. Los sistemas expertos son ejemplos de modelos verbales formalizados. Es importante diferenciar entre los modelos mentales y los verbales. Por ejemplo, nosotros usamos un modelo mental de la dinámica de la bicicleta cuando la conducimos. Sin embargo, no es sencillo convertirlo en un modelo verbal.

1.3.3. Modelos físicos

Además de los modelos mentales y verbales, existe otro tipo de modelos que tratan de imitar al sistema real. Son los *modelos físicos*, tales como las maquetas a escala que construyen los arquitectos, diseñadores de barcos o aeronaves, para comprobar las propiedades estéticas, aerodinámicas, etc.

1.3.4. Modelos matemáticos

Finalmente, existe un cuarto tipo de modelo: el *modelo matemático*. En él las relaciones entre las magnitudes del sistema que pueden ser observadas (distancias, velocidades, flujos, etc.) son descritas mediante relaciones matemáticas.

La mayoría de las teorías sobre las leyes de la naturaleza son descritas empleando modelos matemáticos. Por ejemplo, para el sistema "masa puntual", la Ley de Newton del movimiento describe la relación entre la fuerza y la aceleración. Asimismo, para el sistema "resistencia eléctrica", la Ley de Ohm describe la relación entre la caída de tensión y la corriente eléctrica.

En algunos casos, las relaciones matemáticas que constituyen los modelos son sencillas y pueden resolverse analíticamente. Sin embargo, en la mayoría de los casos los modelos no pueden resolverse analíticamente y deben estudiarse con ayuda del ordenador aplicando métodos numéricos. Este experimento numérico realizado sobre el modelo matemático recibe el nombre de **simulación**.

Al igual que se distingue entre el sistema real y el experimento, es conveniente distinguir entre la *descripción del modelo* y la *descripción del experimento*. Esto es así tanto desde el punto de vista conceptual como práctico.

Sin embargo, en el caso de los modelos matemáticos esta separación implica cierto riesgo: el empleo del modelo matemático en unas condiciones experimentales para las cuales no es válido.

Cuando se trabaja con sistemas reales este riesgo nunca existe, ya que un sistema real es válido para cualquier experimento.

Por el contrario, cualquier modelo matemático está fundamentado en un determinado conjunto de hipótesis. Cuando las condiciones experimentales son tales que no se satisfacen las hipótesis del modelo, éste deja de ser válido.

Para evitar este problema, la descripción del modelo matemático debe ir acompañada de la documentación de su **marco experimental**. Éste establece *el conjunto de experimentos para el cual el modelo matemático es válido.*

1.4. CLASIFICACIÓN DE LOS MODELOS MATEMÁTICOS

La finalidad de un estudio de simulación (es decir, las preguntas que debe responder) condiciona las hipótesis empleadas en la construcción del modelo y éstas a su vez determinan qué tipo de modelo resulta más adecuado. De hecho, un mismo sistema puede ser modelado de múltiples formas, empleando diferentes tipos de modelos, dependiendo de la finalidad perseguida en cada caso.

Existen diferentes clasificaciones de los modelos matemáticos, atendiendo a diferentes criterios. A continuación, se describen algunas de las clasificaciones más comúnmente usadas.

1.4.1. Determinista o estocástico

Un modelo matemático es *determinista* cuando todas sus variables de entrada son deterministas. Es decir, el valor de cada una de ellas es conocido en cada instante.

Un ejemplo de modelo determinista es un servicio al cual los clientes acceden ordenadamente, cada uno a una hora preestablecida (de acuerdo, por ejemplo, con un libro de citas), y en el cual el tiempo de servicio a cada cliente está igualmente

preestablecido de antemano. No existe incertidumbre en la hora de inicio o de finalización de cada servicio.

Por el contrario, un modelo es *estocástico* cuando alguna de sus variables de entrada es aleatoria. Las variables del modelo calculadas a partir de variables aleatorias son también aleatorias. Por ello, la evolución de este tipo de sistemas debe estudiarse en términos probabilísticos.

Por ejemplo, considérese el modelo de un parking, en el cual las entradas y salidas de coches se producen en instantes de tiempo aleatorios. La aleatoriedad de estas variables se propaga a través de la lógica del modelo, de modo que las variables dependientes de ellas también son aleatorias. Este sería el caso, por ejemplo, del tiempo que transcurre entre que un cliente deja aparcado su vehículo y lo recoge (tiempo de aparcamiento), el número de vehículos que hay aparcados en un determinado instante, etc.

Es importante tener en cuenta que realizar una única réplica de una simulación estocástica es equivalente a realizar un experimento físico aleatorio una única vez. Por ejemplo, si se realiza una simulación del comportamiento del parking durante 24 horas, es equivalente a observar el funcionamiento del parking real durante 24 horas. Si se repite la observación al día siguiente, seguramente los resultados obtenidos serán diferentes, y lo mismo sucede con la simulación: si se realiza una segunda réplica independiente de la primera, seguramente los resultados serán diferentes.

La consecuencia que debe extraerse de ello es que el diseño y el análisis de los experimentos de simulación estocásticos debe hacerse teniendo en cuenta esta incertidumbre en los resultados. Es decir, debe hacerse empleando técnicas estadísticas.

1.4.2. Estático o dinámico

Un *modelo de simulación estático* es una representación de un sistema en un instante de tiempo particular, o bien un modelo que sirve para representar un sistema en el cual el tiempo no juega ningún papel. Por otra parte, un *modelo de simulación dinámico* representa un sistema que evoluciona con el tiempo.

1.4.3. Tiempo continuo o discreto

Un *modelo de tiempo continuo* está caracterizado por el hecho de que el valor de sus variables de estado puede cambiar infinitas veces (es decir, de manera continua) en un intervalo finito de tiempo. Un ejemplo es el nivel de agua en un depósito.

Por el contrario, en un *modelo de tiempo discreto* los cambios pueden ocurrir únicamente en instantes separados en el tiempo. Sus variables de estado pueden cambiar de valor sólo un número finito de veces por unidad de tiempo.

Pueden definirse modelos con algunas de sus variables de estado de tiempo continuo y las restantes de tiempo discreto. Este tipo de modelo, con parte de tiempo continuo y parte de tiempo discreto, de llama *modelo híbrido*.

Tal como se ha indicado al comienzo de la sección, la decisión de realizar un modelo continuo o discreto depende del objetivo específico del estudio y no del sistema en sí. Un ejemplo de ello lo constituyen los modelos del flujo de tráfico de vehículos. Cuando las características y el movimiento de los vehículos individuales son relevantes, puede realizarse un modelo de tiempo discreto. En caso contrario, puede resultar más sencillo realizar un modelo de tiempo continuo.

En este punto es conveniente realizar una consideración acerca de los modelos de tiempo continuo y discreto. Al igual que las *variables continuas* (aquellas que pueden tomar cualquier valor intermedio en su rango de variación) son una idealización, también lo son los *modelos de tiempo continuo*.

Cualquiera que sea el procedimiento que se emplee para medir el valor de una variable, tendrá un límite de precisión. Este límite implica la imposibilidad de obtener una medida continua y supone que, en la práctica, todas las medidas son discretas. Igualmente, los ordenadores trabajan con números representados mediante un número finito de dígitos.

Por otra parte, al simular mediante un computador digital un modelo de tiempo continuo, debe discretizarse el eje temporal a fin de evitar el problema de los infinitos cambios en el valor de los estados. Esta discretización constituye una aproximación (con su error asociado) que transforma el modelo de tiempo continuo en un modelo de tiempo discreto. Por ejemplo, si se discretiza el eje temporal del modelo de tiempo continuo

$$\frac{dx}{dt} = f\left(x, u, t\right) \tag{1.1}$$

con un intervalo de discretización Δt, se obtiene (empleando el método de Euler explícito) el modelo de tiempo discreto siguiente:

$$\frac{x_{K+1} - x_K}{\Delta t} = f\left(x_K, u_K, t_K\right) \longrightarrow x_{K+1} = x_K + \Delta t \cdot f\left(x_K, u_K, t_K\right) \tag{1.2}$$

1.5. NIVELES EN EL CONOCIMIENTO DE LOS SISTEMAS

En la teoría de sistemas se diferencian dos aspectos fundamentales, ortogonales entre sí, en lo que respecta a la especificación de los sistemas.

Por una parte, se encuentra el *formalismo* empleado para la especificación del sistema, que determina el tipo de modelo matemático a emplear para describir el sistema. Por ejemplo, si va a emplearse un modelo de tiempo discreto, de tiempo continuo o híbrido.

Por otra parte, se encuentra el *nivel de conocimiento* que poseemos del sistema. A este respecto, G.J. Klir estableció una clasificación del conocimiento que puede poseerse acerca de un sistema. Distinguió cuatro niveles en el conocimiento, de modo que en cada nivel se conocen aspectos importantes del sistema que no se conocen en los niveles inferiores del conocimiento.

1.5.1. Niveles en el conocimiento de un sistema

De acuerdo con la clasificación propuesta por G.J. Klir, los cuatro niveles en el conocimiento de un sistema son los siguientes:

- **Nivel 0 - Fuente.** En este nivel, se identifica la porción del mundo real que vamos a modelar y las maneras mediante las cuáles vamos a observarlo. Dicha porción del mundo real puede denominarse *sistema fuente*, puesto que constituye una fuente de datos.
- **Nivel 1 - Datos.** En este nivel, disponemos de una base de datos de medidas y observaciones del sistema fuente.
- **Nivel 2 - Generación.** En este nivel, somos capaces de recrear estos datos usando una representación más compacta, por ejemplo, mediante fórmulas matemáticas. Puesto que es posible generar un determinado conjunto de datos

mediante diferentes fórmulas y empleando otros procedimientos, la manera en concreto a la que hemos llegado para generar los datos constituye un conocimiento que no teníamos al Nivel 1 (datos). En el contexto del modelado y la simulación, cuando se habla de un *modelo*, frecuentemente nos referimos a una representación del conocimiento a este nivel.

– **Nivel 3 - Estructura.** En este último nivel, disponemos de un tipo muy específico de sistema de generación de datos. Es decir, sabemos cómo generar los datos observados en el Nivel 1 de una manera más específica: en términos de componentes interconectados entre sí.

1.5.2. Análisis, inferencia y diseño

Esta clasificación proporciona una perspectiva unificada de varios conceptos. Por ejemplo, desde esta perspectiva hay únicamente tres tipos básicos de problemas relacionados con sistemas, y éstos implican moverse entre los distintos niveles del conocimiento. Los tres tipos básicos de problema son los siguientes:

– En el *análisis de un sistema* se intenta comprender el comportamiento del sistema, existente o hipotético, basándose para ello en el conocimiento que se tiene de su estructura.

– En la *inferencia sobre un sistema* se intenta conocer la estructura del sistema a partir de las observaciones que pueden realizarse del mismo.

– En el *diseño de un sistema* se investigan diferentes estructuras alternativas para un sistema completamente nuevo o para el rediseño de uno ya existente.

La idea central es que cuando nos movemos hacia niveles inferiores del conocimiento, como sucede en el caso del *análisis* de sistemas, no estamos generando conocimiento nuevo. Estamos únicamente haciendo explícito lo que ya está implícito en la descripción que tenemos.

Podría considerarse que hacer explícito un conocimiento implícito ayuda a comprender o entender mejor el sistema, con lo cual es una forma de adquirir nuevo conocimiento. Sin embargo, ganar esa clase de conocimiento subjetivo, dependiente de la persona que realiza el modelo, no se considera (en este contexto) adquirir conocimiento nuevo.

Abundando en la idea anterior, una forma de *análisis* de sistemas en el ámbito del modelado y la simulación es la simulación por ordenador, que genera datos de

acuerdo a las instrucciones proporcionadas por un modelo. Aunque con ello no se genera nuevo conocimiento, pueden salir a relucir propiedades interesantes de las que no éramos conscientes antes de iniciar el análisis.

Por otra parte, la *inferencia* acerca de los sistemas y el *diseño* de sistemas son problemas que requieren ascender en los niveles de conocimiento. En ambos casos, tenemos una descripción del sistema de bajo nivel y queremos obtener una descripción equivalente de más alto nivel.

En la *inferencia*, disponemos de una base de datos del comportamiento del sistema fuente y tratamos de encontrar una representación del conocimiento al Nivel 2 (generación) o al Nivel 3 (estructura), que nos permita recrear los datos de que disponemos. Este proceso de denomina *construcción del modelo*.

En el caso de la *inferencia*, el sistema fuente existe. Sin embargo, en el caso del *diseño* el sistema fuente no existe y el objetivo es construir un sistema que tenga la *funcionalidad* deseada, es decir, funcione de la manera deseada. Si el objetivo es llegar a construir el sistema, debe llegarse a un Nivel 3 (estructura) del conocimiento, puesto que la construcción se realizará mediante la interconexión de diferentes componentes tecnológicos.

Finalmente, el proceso denominado *ingeniería inversa* tiene elementos tanto de *inferencia* como de *diseño*. Para hacer ingeniería inversa de un sistema existente, en primer lugar se realiza un gran número de observaciones de él. A partir de estas observaciones, se infiere el comportamiento del sistema y se diseña una estructura alternativa que tenga ese comportamiento.

1.6. JERARQUÍA EN LA ESPECIFICACIÓN DE SISTEMAS

Los cuatro niveles epistemológicos (del conocimiento) que han sido descritos en la Sección 1.5 fueron propuestos por G.J. Klir a principios de la década de 1970. Aproximadamente en la misma fecha, B.P. Zeigler introdujo una clasificación similar, pero más enfocada al modelado y la simulación.

Esta jerarquía en la especificación de sistemas, a diferencia de la propuesta por Klir, emplea el concepto de *sistema dinámico* y reconoce que la simulación está relacionada con la *dinámica*, es decir, con la forma en que los sistemas se comportan en el tiempo.

Asimismo, reconoce que conceptualmente puede considerarse que los sistemas tienen *interfaces*, a través de las cuales pueden interactuar con otros sistemas. Los

sistemas reciben estímulos ordenados en el tiempo a través de sus *puertos de entrada* y responden a través de sus *puertos de salida*. Obsérvese que el término *puerto* conlleva implícitamente una manera específica de interpretar la interacción con el sistema, estimulándolo a través de sus puertos de entrada y observándolo a través de sus puertos de salida.

Las estímulos aplicados a las entradas del sistema, ordenados en el tiempo, se denominan *trayectorias de entrada*, mientras que las observaciones realizadas en las salidas del sistema, ordenadas en el tiempo, se denominan *trayectorias de salida*. Por definición, los *puertos* son los únicos canales a través de los cuales se puede interaccionar con el sistema. Esto es equivalente a considerar que los sistemas son *modulares*.

La jerarquía en la especificación de los sistemas consta de los cinco niveles descritos a continuación. Un determinado sistema puede ser descrito de varias maneras, correspondientes a diferentes niveles en su especificación. Cuanto mayor es el nivel en la descripción de un sistema, mayor es el conocimiento sobre él que se hace explícito en dicha descripción.

1.6.1. Nivel 0 - Marco de observación

En este nivel, sabemos cómo estimular las entradas del sistema, qué variables de salida medir y cómo observarlas en función del tiempo. La descripción del sistema a este nivel es como una caja negra, cuyo comportamiento sólo es observable a través de sus puertos de entrada y de salida. Este nivel en la especificación del sistema se corresponde con el "Nivel 0 - Fuente" de la clasificación de Klir.

En este nivel de conocimiento del sistema, suele darse la circunstancia de que, para unas mismas trayectorias de entrada, se obtienen diferentes trayectorias de salida. Es decir, si se replica el experimento varias veces, usando en todos los casos unas determinadas trayectorias de entrada, en las sucesivas réplicas se obtienen diferentes trayectorias de salida.

En la Figura 1.2 se muestra un ejemplo de especificación al Nivel 0 de un sistema consistente en un bosque, sobre el cual caen rayos y hay viento y lluvia. Estas variables constituyen los puertos de entrada: los rayos, la lluvia y el viento. El humo producido por un incendio en el bosque es el puerto de salida (véase la Figura 1.2a). Los valores que puede tomar la variable *humo*, denominados el *rango* de la variable, se obtienen aplicando algún procedimiento de medida, por ejemplo, midiendo la densidad de partículas en el aire.

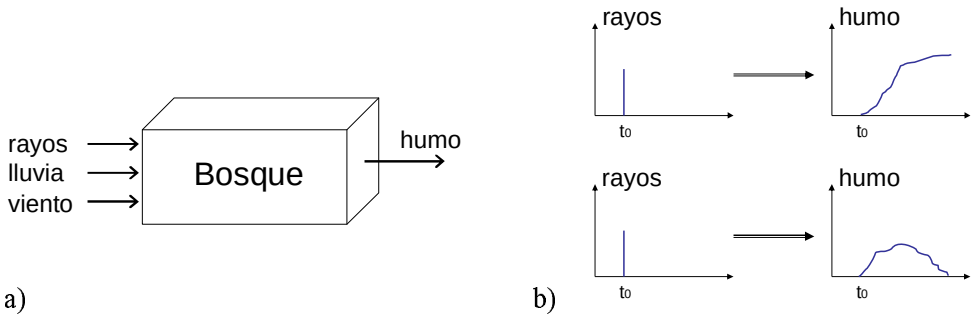

Figura 1.2: Modelo de un bosque: a) especificado en el "Nivel 0 - Marco de observación"; y b) algunas parejas de trayectorias de entrada y salida.

Obsérvese que la elección de las variables que incluimos en los *puertos*, así como su *orientación* (si son puertos de entrada o de salida), depende de cuáles sean nuestros objetivos al realizar el modelo.

1.6.2. Nivel 1 - Comportamiento E/S

En la Figura 1.2b se muestran algunos ejemplos de trayectorias de entrada y de salida. La trayectoria de entrada muestra la caída de un rayo en el instante t_0. Es el único rayo que cae en el periodo de tiempo representado.

La evolución del humo (trayectoria de salida) mostrada en la parte superior de la Figura 1.2b parece indicar que el fuego se propaga, y se hace más y más extenso con el paso del tiempo. Por el contrario, la evolución del humo en la parte inferior de la Figura 1.2b parece indicar que el fuego se extingue transcurrido un cierto tiempo.

Esta situación, en la cual se obtienen diferentes trayectorias de salida para una misma trayectoria de entrada, es típica del conocimiento en el Nivel 1.

La colección de todos los pares de trayectorias de entrada y salida, recogidas mediante observación, se llama comportamiento de entrada/salida del sistema (abreviadamente, *comportamiento de E/S*) y corresponde con el Nivel 1 de especificación del sistema.

1.6.3. Nivel 2 - Función E/S

Supongamos ahora que somos capaces de predecir cuál será la evolución del humo producido en respuesta a la caída de un rayo. Por ejemplo, sabemos que si la vegetación está húmeda, el fuego se extinguirá rápidamente. Si está seca, se propagará. El conocimiento acerca de este factor determinante de la respuesta (vegetación seca o húmeda), que es el *estado inicial* en que se encuentra el sistema, supone un conocimiento en el "Nivel 2 - Función E/S".

En el Nivel 2 se dispone del conocimiento de los niveles inferiores y además se conoce la influencia del estado inicial en la respuesta del sistema. Conocido el estado en que se encuentra inicialmente el sistema, se conoce la relación funcional entre las trayectorias de entrada y de salida. En otras palabras, el estado inicial permite determinar de manera unívoca cuál es la trayectoria de salida para una determinada trayectoria de entrada (véase la Figura 1.3).

1.6.4. Nivel 3 - Transición de estado

En el Nivel 3 de la especificación del sistema, no sólo se aporta información sobre cómo afecta el estado inicial a la trayectoria de salida, sino también cómo evoluciona el estado en respuesta a la trayectoria de entrada.

Los dos ejemplos mostrados en la Figura 1.4 tratan de ilustrar este concepto. En la parte superior de la figura se presenta la situación en la cual el bosque está en el estado {vegetación seca, sin quemar} cuando en el instante t_0 cae un rayo. El estado en el que se encuentra el bosque cuando cae el segundo rayo es {vegetación seca, quemado}. Este estado refleja el hecho de que el fuego ha estado ardiendo en el intervalo de tiempo que va desde t_0 hasta t_1. Dado que el bosque se encuentra en un estado diferente, el efecto del segundo rayo es diferente al efecto del primero: puesto que cuando cae el segundo rayo queda poco por quemar, el efecto del segundo rayo es despreciable.

Por el contrario, en la parte inferior de la Figura 1.4 se muestra la situación en la cual el bosque está en el estado {vegetación húmeda, sin quemar} cuando cae el primer rayo. El rayo no produce un incendio importante, pero seca la vegetación, de modo que el estado cuando cae el segundo rayo es {vegetación seca, sin quemar}. Por este motivo, la caída del segundo rayo produce un incendio importante, análogamente a lo que sucedía con el primer rayo en la situación mostrada en la parte

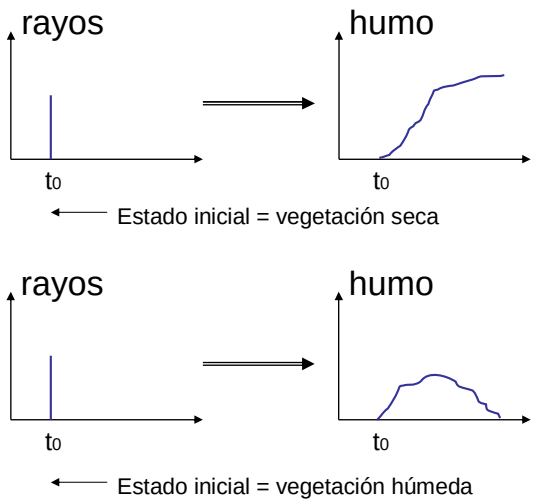

Figura 1.3: La especificación de Nivel 2 toma en consideración el estado inicial.

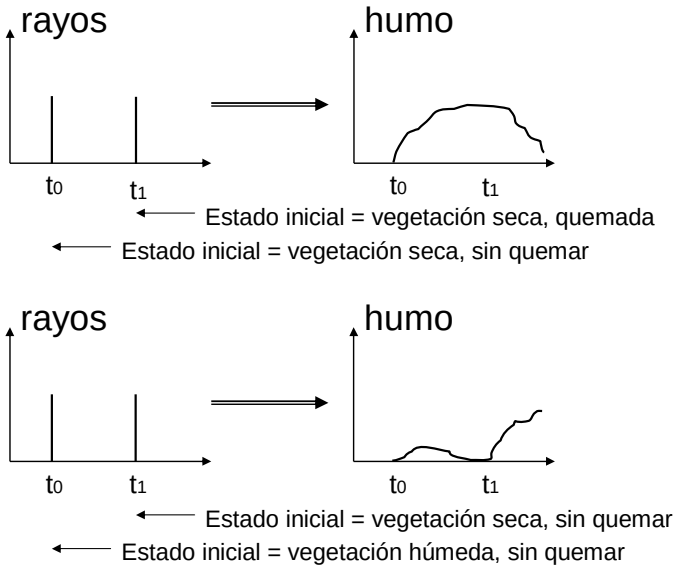

Figura 1.4: La especificación de Nivel 3 describe además la evolución el estado.

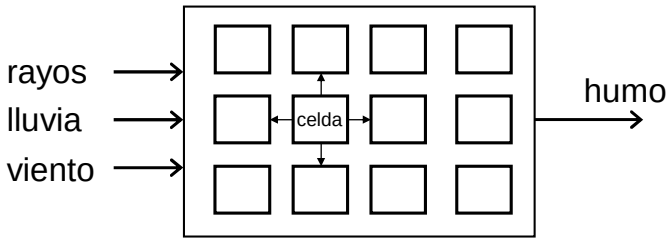

Figura 1.5: Especificación de Nivel 4 - Estructura del bosque.

superior de la figura: puesto que en ambos casos el estado y la trayectoria de entrada son las mismas, la respuesta del sistema es la misma.

1.6.5. Nivel 4 - Componentes acoplados

Al mayor nivel de la especificación del sistema, describimos su estructura interna. En los niveles 0 a 3, el sistema se describe como una caja negra, observable sólo a través de sus puertos de entrada y salida. Por el contrario, en el Nivel 4 se describe cómo el sistema está compuesto de componentes interactuantes.

Por ejemplo, en la Figura 1.5 se muestra que el modelo del bosque está compuesto de celdas interactuantes, cada una representativa de una cierta región espacial, de modo que las celdas adyacentes se hayan conectadas entre sí. Cada una de las celdas es modelada al Nivel 3: se emplea la definición de sus transiciones de estado y de la generación de sus salidas, para describir cómo la celda actúa sobre las demás celdas y el efecto de éstas sobre ella. Las celdas son conectadas por medio de sus puertos. Los puertos de salida de una celda se conectan a los puertos de entrada de las celdas vecinas.

Obsérvese que en esta clasificación se distingue entre el conocimiento de la *estructura* del sistema (cómo está constituido internamente) y el conocimiento de su *comportamiento* (su manifestación externa). Viendo el sistema como una caja negra, el *comportamiento externo* del sistema es la relación que éste impone entre las entradas aplicadas sobre él y la secuencia de salidas obtenidas. La *estructura interna* del sistema incluye su estado, el mecanismo de transición de los estados (cómo las entradas transforman el estado del sistema), y la relación entre el estado y las salidas. Conocer la estructura del sistema permite *deducir* su comportamiento. Por el contrario, normalmente no es posible *inferir* de forma unívoca la estructura del sistema a partir de la observación de su comportamiento.

1.7. MODELADO MODULAR Y JERÁRQUICO

Una manera de abordar el modelado de sistemas complejos consiste en descomponerlos y describirlos mediante cierto número de componentes que interactúan entre sí. En lugar de intentar describir globalmente el comportamiento del sistema completo, es más sencillo realizar un *análisis por reducción*. Es decir, dividir el sistema en partes, modelar las partes independientemente y finalmente describir la interacción entre las partes.

Para describir un sistema de esta manera, es preciso definir los componentes que componen el sistema y especificar cómo los componentes interaccionan entre sí. La interacción entre los componentes puede describirse de las dos formas siguientes:

- *Acoplamiento modular*: la interacción entre los componentes es descrita mediante la conexión de los puertos de sus interfaces. Es decir, la interacción de cada componente con el resto de componentes es descrita mediante la conexión de sus puertos de entrada y salida de manera modular. Este tipo de modelos se denominan *modelos modulares*.

- *Acoplamiento no modular*: consiste en describir la interacción entre los componentes a través de la influencia que tiene el estado de unos componentes (componentes *influenciadores*) sobre la transición del estado de otros (componentes *influenciados*). En este tipo de modelos, el estado de los componentes *influenciadores* interviene directamente en las funciones de transición de estados de los componentes *influenciados*. Como veremos, este tipo de acoplamiento se emplea comúnmente en los autómatas celulares.

En los modelos modulares, cada componente es a su vez un modelo con sus propios puertos de entrada y de salida. La transmisión de los valores desde los puertos de salida hasta los puertos de entrada se produce instantáneamente. Igualmente, la transmisión de los valores desde las entradas al modelo compuesto hasta las entradas a los componentes, o desde las salidas de los componentes hasta las salidas del modelo compuesto, se producen instantáneamente.

Inmediatamente surge la cuestión de qué condiciones deben satisfacer los modelos de los componentes y la conexión entre ellos para garantizar que el modelo compuesto resultante está bien definido. La respuesta a esta pregunta, planteada de forma general, no se conoce. No obstante, estas condiciones sí son bien conocidas para los formalismos básicos, tales como DEVS, DTSS, DESS y DEV&DESS, los cuales serán descritos en los sucesivos temas de este texto.

Se dice que un formalismo es *cerrado bajo acoplamiento* si el modelo resultante de cualquier conexión de componentes descritos mediante ese formalismo es, a su vez, un modelo especificado en dicho formalismo. Esta propiedad de los formalismos básicos de ser cerrados bajo acoplamiento hace posible la *construcción jerárquica* de modelos.

Los conceptos de *modularidad* y de *acoplamiento* permiten: a) desarrollar y probar los modelos como unidades independientes; b) situarlos en un repositorio de modelos;

y c) reutilizarlos en cualquier contexto de aplicación en el cual su comportamiento sea apropiado y tenga sentido conectarlos a otros componentes.

1.8. MARCO FORMAL PARA EL MODELADO Y LA SIMULACIÓN

En esta sección se presenta un marco formal para el modelado y la simulación, en el cual se define un conjunto de entidades (*sistema fuente*, *base de datos del comportamiento*, *modelo*, *simulador* y *marco experimental*) y las relaciones entre ellas. De entre estas relaciones cabe destacar las dos fundamentales: la *relación de modelado* y la *relación de simulación*.

Como veremos, todo ello está relacionado con los cinco niveles para la especificación de sistemas descritos en la Sección 1.6. En la Figura 1.6 se han representado esquemáticamente las cinco entidades y las dos relaciones fundamentales entre ellas. A continuación, se describen las entidades, así como las relaciones de modelado y de simulación.

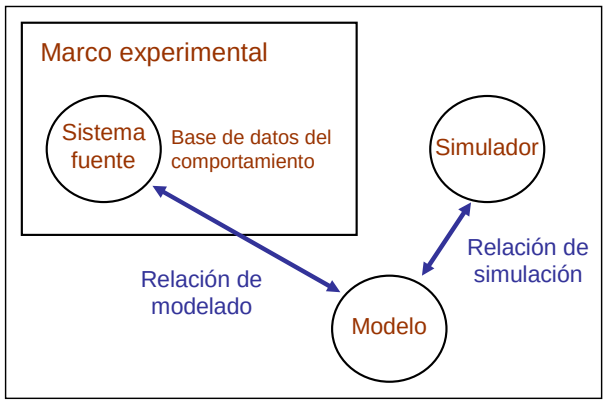

Figura 1.6: Entidades básicas del modelado y simulación, y su relación.

1.8.1. Sistema fuente

El *sistema fuente* es el entorno real o virtual que estamos interesados en modelar, el cual constituye una *fuente de datos observables*, en la forma de trayectorias (observaciones indexadas en el tiempo) de variables. Esta entidad es conocida en el "Nivel 0 - Marco de observación" de la especificación del sistema.

1.8.2. Base de datos del comportamiento

Los datos que se han recogido a partir de observaciones o experimentando con el sistema se llaman *base de datos del comportamiento* del sistema. Estas observaciones son características de la especificación al "Nivel 1 - Comportamiento E/S" del sistema. Los datos son observados o adquiridos a través de *marcos experimentales* de interés para quien realiza el modelo.

Las distintas aplicaciones del modelado y la simulación difieren en la cantidad de datos disponibles para poblar la *base de datos del comportamiento* del sistema. En *entornos ricos en datos*, suele disponerse de abundantes datos históricos y pueden obtenerse fácilmente nuevos datos realizando medidas. Por el contrario, los *entornos pobres en datos* ofrecen pocos datos históricos o datos de baja calidad (cuya representatividad del sistema de interés es cuestionable). En algunos casos, es imposible obtener mejores datos y en otros casos es caro hacerlo. En este último caso, el proceso de modelado puede dirigir la adquisición de datos a aquellas áreas que tienen un mayor impacto en el resultado final.

1.8.3. Marco experimental

Un *marco experimental* es una especificación de las condiciones bajo las cuales el sistema es observado o se experimenta con él. Es la formulación operacional de los objetivos que motivan un proyecto de modelado y simulación.

Continuando con el ejemplo del modelado del bosque, de la multitud de variables que podrían estudiarse, el conjunto {rayos, lluvia, viento, humo} representa una elección en particular. Este *marco experimental* está motivado por el interés en modelar la propagación del incendio provocado por un rayo. Un marco experimental más refinado contendría además variables tales como el contenido en humedad de la vegetación y la cantidad de material sin quemar.

Así pues, pueden formularse muchos marcos experimentales para un mismo sistema y un mismo marco experimental puede aplicarse a varios sistemas. Cabe preguntarse cuál es el propósito de definir varios marcos experimentales para un mismo sistema, o de aplicar el mismo marco experimental a varios sistemas. Las razones son las mismas por las cuales tenemos diferentes objetivos al modelar un mismo sistema, o perseguimos un mismo objetivo al modelar diferentes sistemas.

Hay dos visiones igualmente válidas acerca de qué es un *marco experimental*. La primera ve el marco experimental como la definición del tipo de datos que

se incluirán en la base de datos del comportamiento del sistema. La segunda ve el marco experimental como un sistema que interactúa con el sistema de interés para obtener observaciones, bajo determinadas condiciones experimentales. En esta visión, el marco experimental está caracterizado por su implementación como sistema de medida u observador, y típicamente tiene los tres componentes siguientes (véase la Figura 1.7):

Generador	Genera las secuencias de entrada al sistema.
Receptor	Monitoriza el experimento, para comprobar que se satisfacen las condiciones experimentales requeridas.
Transductor	Observa y analiza las secuencias de salida del sistema.

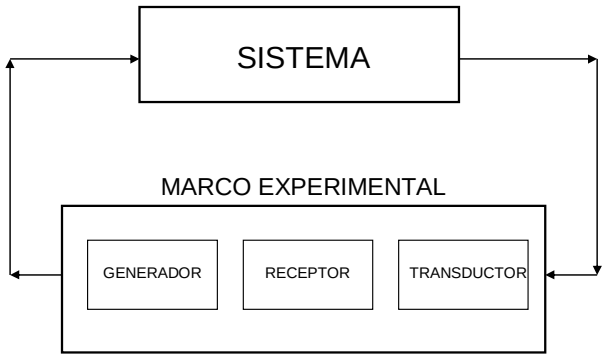

Figura 1.7: Marco experimental y sus componentes.

Como ya se ha indicado anteriormente, es importante establecer lo antes posible en el proceso de desarrollo del modelo cuáles son los objetivos del estudio, ya que los objetivos sirven para enfocar el modelo en aquellos aspectos del sistema que son relevantes para el propósito del estudio. En otras palabras, conocer los objetivos del estudio permite plantear los marcos experimentales adecuados. Los marcos experimentales trasladan los objetivos a condiciones de experimentación más precisas para el sistema fuente y sus modelos.

Una vez fijados los objetivos, presumiblemente existirá un nivel en la especificación del sistema que será el más adecuado para contestar la cuestión planteada. Cuanto más exigentes sean las preguntas, normalmente mayor es la resolución necesaria (nivel en la descripción del modelo) para contestarlas. Por ello, la elección de un nivel de abstracción apropiado repercute en la consecución de los objetivos.

Un procedimiento para transformar los objetivos en marcos experimentales, en aquellos casos en que los objetivos se refieren al diseño del sistema, es el siguiente:

1. Se establece qué medidas van a usarse para evaluar las diferentes alternativas de diseño. Estas medidas deben cuantificar la eficacia con la que el sistema cumple con sus objetivos. Llamaremos a estas medidas las *medidas de salida*.

2. Para calcular esas medidas, el modelo deberá incluir ciertas variables (llamadas *variables de salida*) cuyo valor deberá calcularse durante la ejecución del modelo.

3. El cálculo de las *medidas de salida* a partir de las *variables de salida* se realiza en el componente *transductor* del marco experimental.

Para ilustrar cómo diferentes objetivos conducen a diferentes marcos experimentales y a diferentes modelos, consideremos nuevamente el problema de los incendios forestales. Hay dos aplicaciones fundamentales del modelado y la simulación en este área. La primera relacionada con la extinción del incendio una vez se ha desencadenado. La segunda relacionada con la prevención y la evaluación de los daños. Formulados como objetivos para el modelado, estos propósitos conducen a marcos experimentales diferentes.

En el primer marco experimental, actuación en tiempo real por parte de los bomberos, se precisan predicciones precisas de hacia dónde se extenderá el fuego en las próximas horas. Estas predicciones se emplean para situar los recursos en los lugares más adecuados, con el fin de alcanzar la máxima eficacia. Dado que los bomberos pueden ser situados en un grave peligro, es necesario realizar predicciones a corto plazo muy precisas. Una pregunta típica planteada en este tipo de estudios es: en qué medida es seguro situar durante las próximas horas una cuadrilla de bomberos en un determinado emplazamiento, que se encuentra al alcance del frente del fuego. Para mejorar la capacidad de realizar predicciones precisas, el estado del modelo debe ser actualizado con datos de satélite para mejorar su correspondencia con la situación del fuego real según éste evoluciona.

En la prevención de incendios, el énfasis se pone menos en las predicciones a corto plazo de propagación del fuego y más en responder preguntas del tipo "qué pasa si" relativas a la planificación. Por ejemplo, los planificadores urbanísticos pueden preguntar qué anchura debe tener el cortafuegos alrededor de una zona residencial de modo que haya una probabilidad inferior al 0.1 % de que las casas sean alcanzadas por el fuego. En este caso, el modelo debe describir una mayor área de terreno y necesita una precisión considerablemente menor que en el caso anterior. De hecho, un modelo puede ser capaz de ordenar diferentes alternativas sin necesariamente predecir adecuadamente la propagación del fuego.

Así pues, los marcos experimentales desarrollados para estos dos objetivos, extinción y prevención, son diferentes. El primero de ellos (extinción) sugiere experimentar con un modelo en el cual el estado inicial viene determinado por el material combustible, el viento y las condiciones topográficas. La salida deseada es un mapa detallado de la propagación del fuego después de, por ejemplo, 5 horas, en la región de interés.

El segundo marco experimental (prevención) sugiere un mayor alcance, menor resolución en la representación del terreno en el cual los regímenes esperados de rayos, viento, lluvia y temperaturas son inyectados como trayectorias de entrada. Puede colocarse el modelo en diferentes estados, correspondientes a diferentes alternativas de prevención. Por ejemplo, pueden investigarse diferentes distribuciones de los cortafuegos. La salida de cada ejecución puede ser tan simple como una variable binaria que indique si la zona residencial fue o no alcanzada por el fuego. El resultado global del estudio, en el cual se recopila la información de todas las ejecuciones, puede ser una ordenación de las diferentes alternativas, mostrando la eficacia de cada una de ellas en prevenir que el fuego alcance la zona residencial. Por ejemplo, para cada alternativa puede mostrarse el porcentaje de ejecuciones en las cuales la zona residencial no fue alcanzada por el fuego.

1.8.4. Modelo

En general, un modelo es una especificación del sistema proporcionada en cualquiera de los niveles descritos en la Sección 1.6. Normalmente, en el ámbito del modelado y la simulación, los modelos se especifican en los Niveles 3 y 4, es decir, transición de estados y estructura.

Una posible definición del término "*modelo*" es la siguiente. Un modelo es *un conjunto de instrucciones, reglas, ecuaciones o ligaduras para reproducir el comportamiento de entrada/salida.*

1.8.5. Simulador

Dado que un modelo es un conjunto de instrucciones, reglas, ecuaciones o ligaduras, es necesario disponer de un agente capaz de obedecer las instrucciones y reglas, y de evaluar las ecuaciones, con el fin de generar el comportamiento descrito en el modelo. Este agente se denomina *simulador*.

Así pues, un simulador es *cualquier agente computacional (tal como un único procesador, una red de procesadores, la mente humana, o de manera más abstracta, un algoritmo) capaz de ejecutar el modelo para generar su comportamiento.*

Un simulador en sí es típicamente un sistema especificado al "Nivel 4 - Componentes acoplados", puesto que es un sistema que diseñamos intencionadamente para ser sintetizado a partir de la conexión de componentes tecnológicos bien conocidos.

1.8.6. Relación de modelado: validez

La relación básica de modelado, denominada *validez*, se refiere a la relación entre el modelo, el sistema y el marco experimental.

A menudo se piensa en la *validez* como el grado en el cual el modelo representa fielmente al correspondiente sistema. Sin embargo, resulta más práctico requerir que el modelo capture de forma fiel el comportamiento del sistema sólo hasta el punto demandado por los objetivos del estudio de simulación. De esta forma, el concepto de *validez* responde a la pregunta de si es posible distinguir entre el modelo y el sistema en el marco experimental de interés.

El tipo más básico de validez, la *validez replicativa*, se afirma si, para todos los posibles experimentos del marco experimental, el comportamiento del modelo y del sistema se corresponden dentro de una tolerancia aceptable. Así pues, la *validez replicativa* requiere que el modelo y el sistema concuerden al "Nivel 1 - Relacion E/S".

Formas más estrictas de validez son la *validez predictiva* y la *validez estructural*. En la *validez predictiva* no sólo requerimos *validez replicativa*, sino también la habilidad de predecir. La *validez predictiva* requiere la concordancia con el sistema en el siguiente nivel de la jerarquía de sistemas, es decir, al "Nivel 2 - Función E/S".

Finalmente, *validez estructural* requiere concordancia en el "Nivel 3 - Transición de estados" o en el "Nivel 4 - Acoplamiento de componentes". Esto significa que el modelo no sólo es capaz de replicar los datos observados del sistema, sino que también reproduce paso a paso, componente a componente, la forma en que el sistema realiza sus transiciones.

El término *precisión* se usa a veces en lugar de *validez*. Otro término, *fidelidad*, se usa a menudo para una combinación de *validez* y *detalle*. Así, un modelo de gran *fidelidad* se refiere a un modelo que tiene tanto un gran *detalle* como una gran *validez* (en determinado marco experimental). Obsérvese que el *detalle* no siempre implica

validez, puesto que un modelo puede ser muy detallado y tener, sin embargo, mucho error, simplemente porque algunos de los componentes resueltos con gran detalle funcionan de forma diferente a los correspondientes en el sistema real.

1.8.7. Relación de simulación: corrección

La relación básica de simulación, denominada la *corrección del simulador*, es una relación entre un simulador y un modelo. Un simulador simula correctamente un modelo si genera fielmente la trayectoria de salida del modelo dados su estado inicial y su trayectoria de entrada. Así pues, la corrección del simulador se define en términos del "Nivel 2 - Función E/S".

En la práctica, los simuladores son construidos para ejecutar no sólo un modelo, sino una familia de posibles modelos. Esta flexibilidad es necesaria si el simulador va a emplearse en un rango de aplicaciones. En estos casos, debemos establecer que el simulador ejecuta correctamente esa clase particular de modelos.

El hecho inexcusable acerca del modelado es que está severamente constreñido por las limitaciones de la complejidad. La *complejidad* de un modelo puede medirse por los recursos requeridos por un determinado simulador para interpretar el modelo correctamente. Si bien la complejidad se mide relativa a un determinado simulador o familia de simuladores, a menudo las propiedades intrínsecas al modelo están fuertemente correlacionadas con su complejidad, la cual es prácticamente independientemente del simulador.

El modelado exitoso puede verse como la *simplificación válida*. Necesitamos *simplificar*, o reducir la complejidad, para posibilitar que nuestros modelos sean ejecutados eficientemente en los simuladores (de recursos limitados) de que disponemos.

En el proceso de la simplificación están implicados dos modelos: el *modelo base* y el *modelo simplificado*. El modelo base es "más capaz", pero requiere más recursos para ser interpretado que el modelo simplificado. En este contexto, "más capaz", significa que el modelo base es válido dentro de un conjunto de marcos experimentales (con respecto a un sistema real) más amplio que el modelo simplificado. Sin embargo, el punto importante es que, dentro del marco experimental particular de interés, el modelo simplificado y el modelo base sean igualmente válidos.

1.9. EJERCICIOS DE AUTOCOMPROBACIÓN

Ejercicio 1.1

Describa cuál sería en su opinión la forma más eficaz de estudiar cada uno de los sistemas siguientes, en términos de las posibilidades mostradas en la Figura 1.1.

1. Un ecosistema compuesto por varias especies (animales y vegetales) y por recursos (agua, luz, etc.).
2. Una glorieta en la que convergen varias calles y que frecuentemente presenta atascos.
3. Una presa para el suministro de agua y electricidad que se planea construir en un río.
4. El servicio de urgencias de un hospital que se encuentra en funcionamiento.
5. Un circuito eléctrico.

Ejercicio 1.2

Describa qué características tiene cada uno de los tipos de modelo siguientes: *mental*, *verbal*, *físico* y *matemático*. Ponga un ejemplo de modelo de cada tipo, indicando cuál es su finalidad.

Ejercicio 1.3

Para cada uno de los sistemas mencionados en el Ejercicio 1.1, suponga que se ha decidido realizar el estudio mediante simulación. Discuta de qué tipo debería ser, en su opinión, el modelo matemático: estático o dinámico, determinista o estocástico, de tiempo continuo, discreto o híbrido.

Ejercicio 1.4

Plantee un ejemplo de conocimiento de un sistema en cada uno de los cuatro niveles de la clasificación de Klir.

Ejercicio 1.5

Para un sistema de su elección, plantee un ejemplo de análisis del sistema, inferencia sobre el sistema y diseño del sistema. Indique el nivel en el conocimiento del sistema que se precisa, según la clasificación de Klir, para realizar cada una de estas tres actividades.

Ejercicio 1.6

Para un sistema y un marco experimental de su elección, indique cómo podría descomponerse el sistema de manera modular y jerárquica con el fin de realizar su análisis por reducción. Para cada uno de los componentes, indique qué magnitudes escogería como puertos de la interfaz.

Ejercicio 1.7

Describa cómo podría especificarse el sistema que ha propuesto al resolver el Ejercicio 1.4 en cada uno de los cinco niveles para la especificación de sistemas propuestos por Zeigler.

Ejercicio 1.8

Discuta la diferencia entre *acoplamiento modular* y *acoplamiento no modular*. Ponga un ejemplo.

Ejercicio 1.9

Explique el significado de los tres conceptos siguientes: *validez replicativa*, *validez predictiva* y *validez estructural*.

Ejercicio 1.10

Ponga un ejemplo de *modelo base* y, para un determinado marco experimental de su elección, describa un posible *modelo simplificado*.

1.10. SOLUCIONES A LOS EJERCICIOS

Solución al Ejercicio 1.1

La forma más adecuada de estudiar cada uno de los sistemas depende de cuál sea en cada caso el objetivo del estudio. Puesto que en el enunciado no se indican los objetivos, al contestar a la pregunta debe plantearse el objetivo del estudio de cada sistema y, una vez fijado el objetivo, debería explicarse qué forma o formas de estudio del sistema podrían ser válidas para alcanzar ese objetivo.

En algunos de los casos citados en el enunciado sería posible experimentar con el sistema real, mientras que en otros puede ser costoso y complejo (como en el caso del ecosistema), o imposible (como en el caso del diseño de una presa, puesto que el sistema aun no existe).

La experimentación con el sistema real y el modelado matemático son frecuentemente actividades complementarias, dado que los datos experimentales sirven de base para la construcción del modelo y para realizar su validación. En aquellos casos en que no se disponga de datos del sistema real, la construcción del modelo puede basarse en el conocimiento teórico disponible acerca del comportamiento de los diferentes componentes del sistema. Por ejemplo, el diseño de la presa puede realizarse empleando modelos matemáticos basados en una combinación de leyes físicas, relaciones empíricas y conjuntos de datos que describen el comportamiento mecánico de los materiales de construcción, del agua, etc.

A continuación, se plantea un posible objetivo en el estudio de cada sistema y una forma de estudiarlo que podría permitir alcanzar el objetivo. Otros objetivos y formas de estudio podrían ser igualmente válidas.

1. *Un ecosistema compuesto por varias especies (animales y vegetales) y por recursos (agua, luz, etc.).*

   El objetivo del estudio podría ser analizar la evolución del número de individuos de determinadas especies animales y vegetales, a partir de unas determinadas condiciones iniciales y para una cierta evolución en el tiempo de los factores ambientales. Para ello, podría plantearse un modelo matemático del sistema y experimentar con él mediante simulación.

   Puede considerarse que la variación en el tiempo de la cantidad de individuos de una especie vegetal depende de determinados factores ambientales (agua, luz, etc.) y de su tasa de reproducción, así como de la densidad de depredadores

(herbívoros) e individuos de otras especies vegetales que compitan con ella por los recursos (por ejemplo, por la luz).

Análogamente, podría suponerse que la variación en el número de individuos de una especie animal depende de su frecuencia reproductora, así como de la densidad de depredadores y presas, y de la abundancia de recursos tales como el agua.

2. *Una glorieta en la que convergen varias calles y que frecuentemente presenta atascos.*

   El objetivo del estudio podría ser estudiar la conveniencia de colocar semáforos en las calles de acceso a la glorieta y determinar qué programación de los semáforos resulta más adecuada en función de la hora del día. Podría emplearse para ello la simulación de un modelo matemático.

   Si el objetivo del estudio fuera señalizar los accesos a la glorieta, de modo que se diera prioridad a unos frente a otros, quizá podría emplearse un modelo mental.

3. *Una presa para el suministro de agua y electricidad que se planea construir en un río.*

   Si la finalidad del estudio es determinar la forma, altura y espesor de la presa, podría emplearse simulación de modelos matemáticos. Una vez determinada la forma, dimensiones y ubicación de la presa, sería útil realizar un modelo físico (una maqueta a escala) con el fin de dar a conocer las conclusiones del estudio.

4. *El servicio de urgencias de un hospital que se encuentra en funcionamiento.*

   Si el objetivo es estimar el tiempo de espera de los pacientes en función de su frecuencia de llegada, del tiempo del proceso de admisión, del número de boxes, enfermeros y médicos, entonces podría emplearse la simulación de un modelo matemático.

5. *Un circuito eléctrico.*

   Si el objetivo es calcular la tensión en los nodos del circuito y la corriente que circula a través de los componentes eléctricos, entonces podría emplearse un modelo matemático. En algunos casos sencillos el modelo matemático podría ser resuelto de manera analítica. En la práctica lo más habitual es analizar el modelo matemático mediante su simulación por ordenador.

   Otra posibilidad sería construir el circuito y realizar directamente las medidas.

En algunos casos sencillos podría emplearse un modelo mental, para decidir si es necesario aumentar o disminuir el valor de ciertos componentes, con el fin de conseguir modificar la tensión o la corriente de determinada forma.

Solución al Ejercicio 1.2

Un *modelo mental* es un conjunto de ideas que sirve de ayuda para predecir el comportamiento de un sistema. El modelo mental puede haber sido desarrollado a través de la experiencia, la observación o el aprendizaje. Por ejemplo, cuando vamos a cruzar una calle y vemos un coche que se aproxima, empleamos un modelo mental para decidir si nos dará tiempo a cruzar o si, por el contrario, debemos esperar.

Asimismo, los *modelos mentales* constituyen el estadio inicial en el desarrollo de cualquiera de los otros tipos de modelos, ya que no será posible desarrollar un modelo verbal, físico o matemático sin previamente haber desarrollado un modelo mental del sistema.

Un *modelo verbal* es la descripción del comportamiento del sistema mediante palabras. Se ajusta al esquema: "si se cumple esta condición, entonces debería ocurrir aquello". Un ejemplo de modelo verbal sería la descripción verbal del funcionamiento de un electrodoméstico (por ejemplo, una lavadora) donde se van describiendo las posibles acciones a realizar y sus resultados.

Un *modelo físico* es una maqueta que reproduce total o parcialmente un sistema. Un ejemplo de modelo físico es una maqueta de un vehículo cuyas características aerodinámicas quieren estudiarse en el túnel de viento. Otra finalidad de los modelos físicos es analizar los diseños desde el punto de vista estético.

Un *modelo matemático* consiste en la representación de las magnitudes de interés del sistema mediante variables y de la relación entre estas magnitudes mediante ecuaciones. Un ejemplo de modelo matemático serían las ecuaciones de Maxwell, que describen los fenómenos electromagnéticos.

Solución al Ejercicio 1.3

El modelo matemático del *ecosistema* podría ser dinámico, determinista y de tiempo continuo. Dinámico dado que se quiere estudiar la evolución en el tiempo de las variables. Determinista puesto que se conocen de manera precisa las condiciones iniciales y además se conoce en cada instante el valor de las variables de entrada al

modelo (los factores ambientales). Es de tiempo continuo puesto que las variables analizadas varían de forma continua en el tiempo.

El modelo de la *glorieta* podría ser dinámico, estocástico y de tiempo discreto. Dinámico puesto que se pretende analizar la evolución temporal del sistema. Es estocástico si el proceso de llegada de los coches por cada una de las calles que confluye en la glorieta se describe mediante un proceso estocástico. Es decir, si se supone que el tiempo que transcurre entre la llegada de dos coches sucesivos está distribuido de acuerdo a cierta distribución de probabilidad. Es de tiempo discreto si el modelo se construye de modo que el valor de sus variables sólo cambie en los instantes de tiempo en que se producen eventos: cuando se produce la llegada de un nuevo vehículo, cuando cambia el estado de un semáforo, etc.

Otra alternativa sería describir la *glorieta* mediante un modelo de tiempo continuo: en lugar de representar cada coche como una entidad independiente, puede ser más eficiente computacionalmente considerar que el tráfico de coches es un flujo continuo, de valor cero cuando los coches están detenidos y de valor $f(t)$ vehículos/minuto cuando los coches están en movimiento. Se escribe $f(t)$ para indicar que el flujo, f, es una función del tiempo, t.

El modelo de la *presa* podría ser estático y determinista. El modelo sería estático si se trata de calcular la distribución de estrés en los materiales de la pared, que corresponde a cierta presión ejercida por el agua. Sería determinista si el valor de las variables de entrada al modelo (por ejemplo, la presión ejercida por el agua) es conocida.

El modelo del servicio de urgencias, como sucede habitualmente con los modelos de los procesos logísticos, podría ser un modelo dinámico, estocástico y de tiempo discreto. El modelo es estocástico ya que los procesos de llegada son estocásticos y los tiempos de proceso son variables aleatorias. Es de tiempo discreto ya las variables del sistema sólo cambian en los instantes en que se producen los eventos: llegada de un paciente, un paciente comienza o termina de ser atendido en alguno de los procesos (recepción, primera diagnosis, curas, etc.). En los intervalos de tiempo entre eventos las variables del modelo permanecen constantes.

Finalmente, el modelo del circuito eléctrico podría ser dinámico, determinista y de tiempo continuo. Dinámico puesto que se desea conocer la evolución temporal de tensiones y corrientes. Determinista si se conoce de manera precisa el estado inicial del modelo y en cada instante el valor de las variables de entrada. De tiempo continuo ya que las variables del modelo varían continuamente en el tiempo.

Solución al Ejercicio 1.4

Veamos dos ejemplos. En primer lugar, consideremos los niveles en el conocimiento acerca de un dispositivo diseñado y fabricado por el hombre, como puede ser un *colector solar*, también llamado *panel solar térmico*. Se trata de un dispositivo que forma parte de los calentadores solares y que aprovecha la energía de la radiación solar para calentar un fluido (por ejemplo, una mezcla de agua y glicol) que circula por unos conductos situados dentro del panel. El fluido caliente circula desde el panel hasta su punto de uso (por ejemplo, los radiadores del sistema de calefacción de una vivienda), retornando luego al panel.

- En el Nivel 0 del conocimiento se identifica la porción del mundo que vamos a modelar y las maneras mediante las cuáles vamos a observarlo. Esta porción del sistema real, que en nuestro caso será el panel solar térmico, constituye el sistema fuente. La forma en que vamos a observar el colector será midiendo la radiación solar incidente, la temperatura de entrada del fluido y su temperatura de salida.
- En el Nivel 1 disponemos de una base de datos de medidas del sistema. En este caso, de las temperaturas de entrada y salida del fluido que se han medido para un cierto valor medido de la radiación incidente.
- En el Nivel 2 hemos sido capaces de correlacionar el incremento en la temperatura del fluido con la intensidad de la radiación incidente. Esta correlación la hemos obtenido ajustando los parámetros de una fórmula a los datos experimentales.
- Finalmente, en el Nivel 3 conocemos la estructura del sistema y somos capaces de entender su comportamiento a partir del comportamiento y la interacción de sus componentes. Sabemos que el panel solar térmico está compuesto por una caja rectangular, dentro del cual está el sistema captador de calor. Una de las caras de esta caja está cubierta de un vidrio muy fino y las otras cinco caras son opacas y están aisladas térmicamente. Dentro de la caja, expuesta al sol, se sitúa una placa metálica, que está tratada para que aumente su absorción del calor, y a la cual están soldados los conductos por los que circula el fluido transportador del calor.

Podría ponerse como ejemplo también un sistema logístico, como podría ser una gasolinera, compuesta por varios surtidores, una tienda y varias cajas para realizar el pago.

- En el Nivel 0 conocemos qué porción del mundo vamos a estudiar y las maneras mediante las cuales vamos a observarlo. Una forma de observar el funcionamiento de la gasolinera sería medir el instante de tiempo en que se produce la llegada de un cliente y el instante en que se produce su marcha. Con ello podría estimarse la distribución de probabilidad del tiempo que se tarda en atender a un cliente. También podría conocerse el número de clientes que están siendo atendidos en la gasolinera en cada instante. Entre ambos estadísticos existirá una correlación.

- En el Nivel 1 dispondríamos de una base de datos de medidas: número de clientes en la gasolinera y tiempos de atención.

- En el Nivel 2 dispondríamos de un modelo que permite predecir el tiempo medio de atención al cliente en función del número de clientes que se encuentran en la gasolinera.

- En el Nivel 3 conocemos de manera detallada la estructura de la gasolinera: número de surtidores, número de cajas de pago, tipo de cola que se forma frente a las cajas de pago (una cola común a todas o una cola frente a cada caja), etc. Para un determinado proceso de llegada de clientes y conociendo la distribución de los tiempos de proceso de llenado en los surtidores, compra en la tienda y cobro en las cajas, podemos simular el funcionamiento de la gasolinera y calcular estadísticos representativos del mismo. Es decir, podemos conocer el funcionamiento del sistema completo a partir del funcionamiento e interacción de sus partes.

Solución al Ejercicio 1.5

Consideremos el *colector solar* descrito al contestar al Ejercicio 1.4. El *análisis* del sistema consistiría en intentar comprender su funcionamiento basándose para ello en el conocimiento que se tiene de su estructura. A partir del análisis de su estructura, podemos comprender que el colector solar funciona aprovechando el efecto invernadero. El vidrio deja pasar la luz visible del sol, que incide en la placa metálica calentándola. Sin embargo, el vidrio no deja salir la radiación infrarroja de baja energía que emite la placa metálica caliente. A consecuencia de ello, y a pesar de las pérdidas por transmisión (el vidrio es mal aislante térmico), el volumen interior de la caja se calienta por encima de la temperatura exterior.

En la *inferencia*, disponemos de una base de datos del comportamiento del sistema fuente y tratamos de encontrar una representación del conocimiento al Nivel

2 (generación) o al Nivel 3 (estructura), que nos permita recrear los datos de que disponemos. Por ejemplo, hemos observado experimentalmente que el rendimiento de los colectores mejora cuanto menor sea la temperatura de trabajo. El modelo que describe la estructura del sistema debería reproducir este comportamiento: a mayor temperatura dentro de la caja (en relación con la exterior), mayores serán las pérdidas por transmisión en el vidrio. También, a mayor temperatura de la placa captadora, más energética será su radiación, y más transparencia tendrá el vidrio a ella, disminuyendo por tanto la eficiencia del colector.

En el *diseño* de un sistema se investigan diferentes estructuras alternativas para un sistema completamente nuevo o para el rediseño de uno ya existente. Como parte del diseño del colector solar podrían investigarse el efecto sobre la absorción de calor de diferentes longitudes de los conductos, diferente volumen interno de la caja, diferentes pinturas aplicadas sobre la capa metálica, diferentes condiciones de operación (por ejemplo, causal de fluido), etc.

Solución al Ejercicio 1.6

El colector solar podría analizarse descomponiéndolo en los componentes siguientes: el vidrio, el volumen de aire contenido dentro de la caja, la placa metálica, el tubo por el cual circula el fluido y el fluido en sí. Los modelos del volumen de aire, la placa metálica, el tubo y el fluido deberían describir cómo cambia la temperatura de cada uno de estos elementos en función del flujo de energía entrante y saliente al elemento. Así pues, la interfaz de estos componentes debería contener al menos las dos siguientes variables: el flujo de energía y la temperatura. El modelo del vidrio debería permitir calcular qué proporción de la radiación incidente es transmitida, con lo cual en la interfaz del modelo debería aparecer la intensidad de la radiación incidente y transmitida.

La gasolinera podría analizarse descomponiéndola en diferentes procesos, a través de los cuales van pasando las entidades (los clientes). El proceso de llenado de combustible tendría tantos recursos como surtidores y tendría asociada una distribución de probabilidad del tiempo de proceso. Igualmente sucedería con las cajas de pago. Cada uno de estos procesos tendría asociadas unas colas. La interfaz entre los componentes debe transmitir al menos la información acerca de qué cliente abandona o accede al proceso (por ejemplo, cada cliente podría ser identificado mediante un número). Esa información puede ser usada por cada uno de los componentes (procesos) para actualizar el valor de estadísticos del proceso (por ejemplo, el tiempo medio en cola o en número medio de clientes en cola), de variables globales del modelo

y de atributos asociados a cada entidad (por ejemplo, el tiempo total que lleva cada cliente en el sistema).

Solución al Ejercicio 1.7

En el Nivel 0 se sabe cómo estimular el sistema, qué variables de salida medir y cómo observarlas en función del tiempo. En el caso del colector solar, podríamos medir la radiación solar incidente, la temperatura del fluido que entra al colector y la temperatura del fluido que sale de él.

En el Nivel 1, disponemos de medidas de la radiación incidente, y de la temperatura a la entrada y a la salida.

En el Nivel 2, podemos predecir la temperatura de salida del fluido dadas una determinada temperatura de entrada, una radiación incidente y sabiendo que los componentes en el interior de la caja se encuentran a determinada temperatura.

En el Nivel 3, seríamos además capaces también de predecir la evolución de la variable de estado: la temperatura dentro de la caja.

Finalmente, en el Nivel 4 somos capaces de explicar el funcionamiento del sistema en términos de componentes acoplados. El comportamiento de cada uno de los componentes (vidrio, aire contenido dentro de la caja, placa metálica, tubo y fluido) podrían describirse al Nivel 3.

Solución al Ejercicio 1.8

En el *acoplamiento modular*, la interacción entre los componentes es descrita mediante la conexión de los puertos de sus interfaces.

En el *acoplamiento no modular*, la interacción entre los componentes es descrita a través de la influencia que tiene el estado de unos componentes (componentes influenciadores) sobre la transición del estado de otros (componentes influenciados).

Un ejemplo de modelo con acoplamiento modular sería el de la gasolinera descrito anteriormente. En ese modelo los componentes que describen los procesos están conectados a través de sus interfaces.

El modelo modular del colector solar podría realizarse conectando entre sí las interfaces de los componentes. Estas interfaces transmiten información acerca de la temperatura del componente y del flujo de energía establecido al conectar los

componentes entre sí. Este mismo modelo podría formularse de manera no modular, describiendo la variación en la temperatura de cada componente en función de la temperatura a la que se encuentren los demás componentes y del flujo de energía proveniente de la radiación solar.

Solución al Ejercicio 1.9

El concepto de *validez* responde a la pregunta de si es posible distinguir entre el modelo y el sistema en el marco de experimentación de interés.

La *validez replicativa* es el tipo más básico de validez. Solamente requiere que el comportamiento del modelo y del sistema concuerden dentro de una cierta tolerancia aceptable para el propósito del estudio. Este tipo de validez requiere que el modelo y el sistema concuerden al "Nivel 1, Relación E/S".

En la *validez predictiva* se necesita que, además de cumplirse la validez replicativa, haya la capacidad de predecir. La concordancia entre modelo y sistema debe ser al "Nivel 2, Función E/S".

La *validez estructural* requiere que el modelo sea capaz de reproducir paso a paso, componente a componente, la forma en que el sistema realiza sus transiciones. La concordancia entre modelo y sistema debe ser ahora al "Nivel 3, Transición de estados" o al "Nivel 4, Acoplamiento de componentes".

Solución al Ejercicio 1.10

En ingeniería es práctica frecuente escoger entre modelos con diferente nivel de detalle dependiendo de la aplicación en concreto a la que se destinen. Los modelos con menor nivel de detalle normalmente constan de menor número de ecuaciones que los modelos más detallados, con lo cual pueden ser resueltos numéricamente en un tiempo menor. Por tanto, elección de un modelo u otro se realiza atendiendo a la precisión necesaria y al tiempo de simulación admisible.

Por ejemplo, se han desarrollado diferentes modelos matemáticos de los dispositivos semiconductores (diodos, transistores MOS, BJT, etc.), que son empleados dependiendo de su aplicación. Cuando el dispositivo forma parte de un circuito digital, el modelo del dispositivo necesita ser menos detallado que cuando ese mismo dispositivo forma parte de un circuito analógico.

En general, el propósito del estudio de simulación tiene implicaciones decisivas en el nivel de detalle requerido del modelo. Por ejemplo, si el propósito es evaluar el comportamiento de un sistema en términos absolutos, deberá existir un alto grado de correspondencia entre el comportamiento del modelo y del sistema. Por el contrario, si el propósito del estudio es comparar varios diseños, el modelo puede ser válido en un sentido relativo incluso cuando sus respuestas en un sentido absoluto difieran considerablemente de las del sistema real.

Tema 2

FORMALISMOS DE MODELADO Y SUS SIMULADORES

OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema debería saber:

- Describir modelos de tiempo discreto mediante la tabla de transición/salidas, y mediante las funciones de transición de estados y salida.
- Plantear el algoritmo de simulación de modelos sencillos de tiempo discreto.
- Discutir qué es un autómata celular y plantear su algoritmo de simulación.
- Discutir qué es un autómata conmutado.
- Discutir qué es una red lineal de tiempo discreto, expresarla en forma matricial y, a partir de dicha expresión, generar las trayectorias de salida y del estado.
- Asignar la causalidad computacional de un modelo de tiempo continuo, plantear el diagrama de flujo de su algoritmo de simulación, y codificarlo y ejecutarlo empleando algún lenguaje de programación.
- Aplicar los siguientes métodos de integración numérica: Euler explícito, punto medio (Euler-Richardson), Runge-Kutta y Runge-Kutta-Fehlberg.
- Discutir las diferencias conceptuales entre el modelado de tiempo discreto y de eventos discretos.
- Discutir las reglas y la simulación del Juego de la Vida.
- Simular modelos sencillos de eventos discretos empleando el método de la planificación de eventos.
- Discutir e identificar los componentes de los modelos de eventos discretos.
- Discutir las diferencias entre el modelado orientado a los eventos y al proceso.

2.1. INTRODUCCIÓN

En este tema se presentan, de manera informal, los formalismos básicos para el modelado de sistemas de tiempo discreto, de tiempo continuo y de eventos discretos. Se pretende proporcionar una idea de cuál es el tipo de modelo empleado en cada formalismo, cómo emplearlos para representar el comportamiento del sistema y qué tipo de comportamiento cabe esperar observar de cada tipo de modelo.

2.2. MODELADO Y SIMULACIÓN DE TIEMPO DISCRETO

Los modelos de tiempo discreto son normalmente el tipo de modelo más fácil de entender de manera intuitiva. En este formalismo, el modelo va ejecutándose en sucesivos instantes de tiempo, equiespaciados entre sí. El intervalo de tiempo entre dos instantes sucesivos se denomina *periodo de muestreo*.

En cada instante, el modelo se encuentra en un estado, recibe unas entradas y genera unas salidas. El modelo permite predecir, a partir de su estado y de sus entradas actuales, cuáles son sus salidas actuales y cuál será su estado en el siguiente instante de tiempo. El estado en el siguiente instante normalmente depende del estado actual y de las entradas actuales al modelo.

Los modelos de tiempo discreto tienen numerosas aplicaciones. Una aplicación importante es la descripción de circuitos digitales síncronos, en los cuales la frecuencia del reloj define el tamaño del paso de avance en el tiempo.

Asimismo, los modelos de tiempo discreto se emplean frecuentemente como aproximaciones de modelos de tiempo continuo. En este caso, se escoge una *base de tiempo* (por ejemplo, 1 segundo, 1 minuto o 1 año), que determina los instantes de evaluación del modelo, y se hace avanzar el *reloj de la simulación* en pasos que son múltiplos enteros de esta base de tiempo. El sistema es representado mediante el cambio que sufre su estado entre los sucesivos instantes de evaluación. Es decir, entre un instante de "observación" y el siguiente. Así pues, para construir un modelo de tiempo discreto a partir de un modelo de tiempo continuo, debemos definir cómo el estado actual y las entradas determinan el siguiente estado del modelo de tiempo discreto.

2.2.1. Descripción de modelos de tiempo discreto

Cuando el sistema tiene un número finito de estados y sus entradas pueden tomar un número finito de posibles valores, una forma sencilla de especificar el comportamiento del modelo es mediante una tabla, en la cual se escriben todas las posibles combinaciones de valores de los estados y las entradas, y para cada una de éstas se indica el valor de la salida y del siguiente estado. Esta tabla se denomina *tabla de transición/salidas*. A continuación, se muestra un ejemplo.

Ejemplo 2.2.1. *La tabla de transición/salidas siguiente corresponde a un sistema con dos estados, una entrada y una salida.*

Estado actual	*Entrada actual*	*Estado siguiente*	*Salida actual*
0	*0*	*0*	*0*
0	*1*	*1*	*1*
1	*0*	*0*	*0*
1	*1*	*1*	*1*

Los dos estados están representados mediante 0 y 1. La entrada puede tomar dos posibles valores, representados mediante 0 y 1. Hay cuatro combinaciones entre los posibles valores del estado y los posibles valores de la entrada. Estas cuatro combinaciones se escriben en las dos primeras columnas de la tabla. El número de combinaciones determina el número de filas de la tabla. Para cada una de estas combinaciones, se escribe en la tercera columna el valor del estado siguiente. En la cuarta columna se escribe el valor de la salida actual del sistema. Por ejemplo, en la primera fila se indica que cuando el estado actual es 0 y la entrada es 0, entonces la salida del sistema es 0 y el estado en el siguiente instante de tiempo vale 0. □

En los modelos de tiempo discreto, el tiempo avanza a pasos, que normalmente son múltiplos enteros de la base de tiempo. La *tabla de transición/salidas* puede reinterpretarse especificando cómo cambia el estado en función del tiempo:

> Si el estado en el instante t es q y la entrada en el instante t es x, entonces el estado en el instante $t + 1$ será $\delta(q, x)$ y la salida y en el instante t será $\lambda(q, x)$.

La función δ de denomina *función de transición de estado* y permite representar, de manera más abstracta, la información mostrada en las tres primeras columnas

de la tabla de transición/salidas. La función λ se denomina la *función de salida* y corresponde con las primeras dos columnas y con la última columna de la tabla.

Las funciones de transición de estados y de salida constituyen una forma más general de representar la información. Por ejemplo, el modelo de la tabla mostrada en el Ejemplo 2.2.1 puede representarse, de manera más compacta, de la forma siguiente:

$$\delta\left(q, x\right) = x \tag{2.1}$$
$$\lambda\left(q, x\right) = x \tag{2.2}$$

que indica que el estado siguiente y la salida actual están ambos determinados por la entrada actual.

La secuencia de estados, $q(0)$, $q(1)$, $\cdots$ se denomina *trayectoria de los estados*. Conocido el estado inicial, $q(0)$, los siguientes estados de la secuencia se determinan de la forma siguiente:

$$q(t+1) = \delta\left(q(t), x(t)\right) \tag{2.3}$$

y similarmente, la *trayectoria de salida* viene dada por la expresión siguiente:

$$y(t) = \lambda\left(q(t), x(t)\right) \tag{2.4}$$

A continuación, se muestra un algoritmo para calcular el estado y la trayectoria de salida de un modelo de tiempo discreto, a partir de la trayectoria de entrada y el estado inicial. Este algoritmo es un ejemplo de *simulador*.

```
Ti = 0, Tf = 9  tiempo inicial y final, en este caso 0 y 9
x(0) = 1, ..., x(9) = 0   trayectoria de entrada
q(0) = 0   estado inicial
t = Ti
while ( t <= Tf ) {
   y(t)   = λ( q(t), x(t) )
   q(t+1) = δ( q(t), x(t) )
   t      = t + 1
}
```

2.2.2. Autómatas celulares

Aunque el algoritmo anterior es muy sencillo, la naturaleza abstracta de las funciones de transición de estado y de salida permite aplicarlo a la resolución de problemas complejos. Por ejemplo, podemos conectar sistemas en fila, de modo que cada uno de ellos tenga conectado un sistema a su izquierda y otro a su derecha. En la Figura 2.1 se muestra una representación esquemática de un modelo de este tipo.

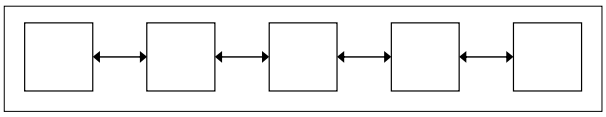

Figura 2.1: Espacio celular unidimensional.

Supongamos que cada sistema tiene dos estados, 0 y 1, y que recibe como entrada los estados de los sistemas vecinos. En este caso, hay 8 posibles combinaciones de los valores de las 2 entradas y del estado del sistema. La *tabla de transición de estados* de cada uno de los sistemas será de la forma siguiente:

Estado actual	Entrada izquierda actual	Entrada derecha actual	Estado siguiente
0	0	0	?
0	0	1	?
0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?

Asignando valor a la cuarta columna, que hemos dejado con interrogaciones, queda definida la *función de transición de estado*. Dado que la cuarta columna tiene 8 filas y que el estado del sistema puede tomar 2 valores, hay $2^8 = 256$ posibles *funciones de transición de estado*. Este tipo de modelo se denomina *automata celular*. Para simular un modelo de este tipo, debemos escoger la función de transición de estados, asignar valor inicial al estado de cada uno de los sistemas que componen el espacio celular y aplicar una versión adecuadamente adaptada del algoritmo simulador.

Un *autómata celular* es una idealización de un fenómeno físico en el cual el espacio y el tiempo están discretizados, y el conjunto de estados es discreto y finito. El autómata celular está compuesto de componentes iguales entre sí, llamados *células*, todos ellos con idéntico aparataje computacional. La distribución espacial de las células puede ser un mallado unidimensional, bidimensional o multidimensional, conectado de manera uniforme. Las células que influyen sobre una célula en particular, denominadas sus *vecinas*, son a menudo aquellas situadas más cerca en el sentido geométrico.

Los autómatas celulares, introducidos originalmente por von Neumann y Ulam como una idealización de la autorreproducción de los sistemas biológicos, muestran comportamientos muy interesantes y diversos. En el texto (Wolfram 1986) se investigan sistemáticamente todas las posibles funciones de transición en autómatas celulares unidimensionales. La conclusión fue que existen cuatro tipos de autómatas celulares cuyo comportamiento difiere significativamente. Estos son:

1. Autómatas en los cuales cualquier dinámica se extingue rápidamente.
2. Autómatas que rápidamente exhiben un comportamiento periódico.
3. Autómatas que muestran un comportamiento caótico.
4. Finalmente, los más interesantes: autómatas cuyo comportamiento es no periódico e impredecible, pero que muestran patrones regulares interesantes.

Un ejemplo interesante de autómata bidimensional es el denominado **Juego de la Vida de Conway** (Gardner 1970). El juego tiene lugar en un espacio celular bidimensional, cuyo tamaño puede ser finito o infinito. Cada celda está acoplada a aquellas que se encuentran más próximas a ella, tanto lateral como diagonalmente. Esto significa que, para una célula situada en el punto $(0, 0)$, sus células vecinas laterales están situadas en los puntos $(0, 1)$, $(1, 0)$, $(0, -1)$ y $(-1, 0)$, y sus células vecinas diagonales están situadas en $(1, 1)$, $(-1, 1)$, $(1, -1)$ y $(-1, -1)$, como se muestra en la Figura 2.2a. En la Figura 2.2b se muestran las células vecinas de una célula situada en la posición (i, j).

El estado de cada célula puede tomar dos valores: 1 (viva) y 0 (muerta). Cada una de las células puede sobrevivir (está viva y permanece viva), nacer (su estado pasa de 0 a 1) y morir (su estado pasa de 1 a 0) a medida que el juego progresa. Las reglas, tal como fueron definidas por Conway, son las siguientes:

1. Una célula permanece viva si tiene en su vecindad 2 ó 3 células vivas.

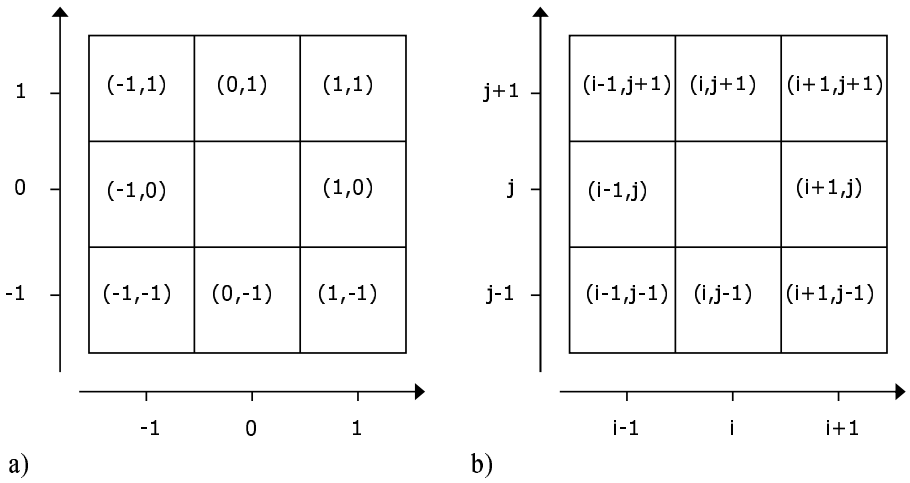

Figura 2.2: Células vecinas en el *Juego de la Vida* a la situada en: a) $(0,0)$; y b) (i,j).

2. Una célula muere debido a superpoblación si hay más de 3 células vivas en su vecindad.

3. Una célula muere a causa del aislamiento si hay menos de 2 células vivas en su vecindad.

4. Una célula muerta vuelve a la vida si hay exactamente 3 células vivas en su vecindad.

Cuando el juego comienza con ciertas configuraciones de células vivas, el *Juego de la Vida* muestra una evolución interesante. A medida que la *trayectoria de los estados* evoluciona, la forma de los grupos de células vivas va cambiando. La idea del juego es encontrar nuevos patrones y estudiar su comportamiento. En la Figura 2.3 se muestran algunos patrones interesantes.

El *Juego de la Vida* ejemplifica algunos de los conceptos expuestos anteriormente. Se desarrolla en una base de tiempo discreto (el tiempo avanza a pasos: 0, 1, 2, $\cdots$) y se trata de un modelo multicomponente: compuesto de componentes (células) conectados entre sí. En contraste con el *estado local* (el estado de la célula), el *estado global* es el conjunto de estados de todas las células en un determinado instante de tiempo. Cada uno de estos estados globales inicia una *trayectoria de estados* (una secuencia de estados globales indexada en el tiempo) que termina en un ciclo o continúa evolucionando para siempre.

El procedimiento básico para simular un automata celular sigue el algoritmo descrito anteriormente. Esto es, en cada instante de tiempo se inspeccionan todas las

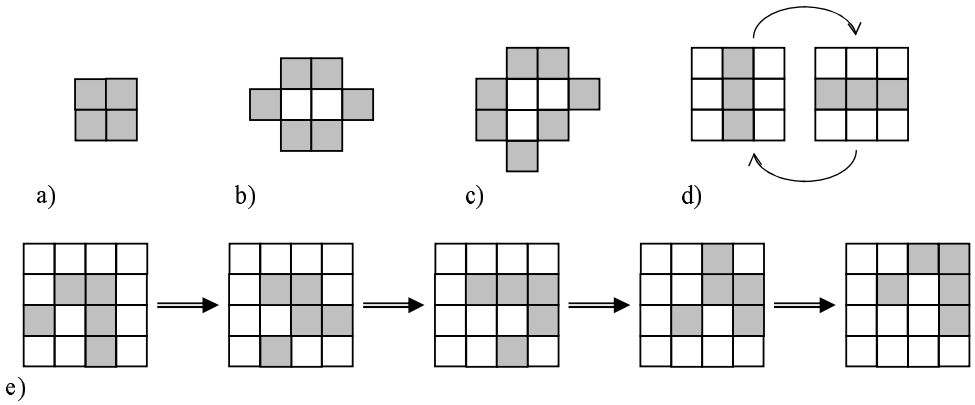

Figura 2.3: Algunos patrones que aparecen en el *Juego de la Vida de Conway*. Los patrones a), b) y c) son estables, ya que no cambian. El patrón d) es oscilante. En e) se muestra un ciclo de patrones que se mueve.

células, aplicando la *función de transición de estado* a cada una de ellas, y salvando el siguiente *estado global* en una segunda copia de la estructura de datos empleada para almacenar el estado global. Una vez calculado el siguiente estado global, éste se convierte en el estado global actual y se avanza el reloj de la simulación un paso.

En aquellos casos en que el espacio de células es infinito, el tiempo necesario para calcular el siguiente estado de todas las células es infinito. Para evitar este problema, la parte del espacio examinada se limita a una cierta región finita. En muchos casos, esta región finita es fija a lo largo de la simulación. Por ejemplo, un espacio bidimensional puede estar representada por una matriz cuadrada de lado N. En este caso, el algoritmo examina N^2 células en cada paso de tiempo.

Debe idearse un procedimiento adicional que tenga en cuenta el hecho de que las células de los bordes de la región no poseen todos sus vecinos. Una solución es asumir que las células situadas en los bordes de la región mantienen un estado constante (por ejemplo, todas están muertas). Otra solución es envolver el espacio de manera toroidal, es decir, interpretar el índice N también como 0.

Existe un procedimiento alternativo al de examinar las N^2 células en cada paso en el tiempo. Consiste en examinar únicamente aquellas células cuyo estado puede potencialmente cambiar. Este procedimiento se denomina de *eventos discretos*.

En los modelos de *tiempo discreto*, en cada paso de tiempo cada componente sufre una "transición en el estado", lo cual sucede con independencia de que el valor del estado cambie o no. Por ejemplo, en el *Juego de la Vida* si una célula está muerta y todas las que le rodean también lo están, entonces en el siguiente paso de tiempo

la célula continua muerta. Si en el espacio celular infinito la inmensa mayoría de las células están muertas, pocas de ellas cambian su estado. En otras palabras, si de definen los *eventos* como cambios en el estado, entonces en la mayoría de los casos se producirán relativamente pocos eventos en el sistema.

Inspeccionar todas las células en una región infinita es imposible. Pero hacerlo, incluso si la región es finita, es claramente ineficiente. En contraste con los *simuladores de tiempo discreto*, que inspeccionan las N^2 células en cada paso de tiempo, los *simuladores de eventos discretos* se concentran en procesar los eventos en lugar de las células, lo cual es considerablemente más eficiente.

La idea básica es tratar de predecir cuándo una célula puede posiblemente cambiar su estado o, por el contrario, cuándo su estado permanecerá inalterado en el estado global del siguiente paso en el tiempo. Supongamos que conocemos el conjunto de células cuyo estado potencialmente puede cambiar. Entonces sólo tendríamos que examinar esas células, en algunas de las cuales cambiará el valor del estado y en otras no cambiará. A continuación, en las nuevas circunstancias, obtenemos el conjunto de células que potencialmente pueden cambiar en el siguiente paso en el tiempo.

Puede obtenerse el conjunto de células susceptibles de cambiar su estado realizando la consideración siguiente. *Si ninguna de sus vecinas cambia su estado en el instante actual, entonces la célula en cuestión no cambiará su estado en el siguiente instante de tiempo.* El procedimiento para la *simulación basada en eventos* sigue la lógica siguiente:

> En una transición de estado deben señalarse aquellas células cuyo estado cambia. A continuación, establézcase el conjunto de todas las células vecinas de aquellas. Este conjunto contiene todas las células que pueden posiblemente cambiar su estado en el siguiente paso de tiempo. El estado de las células no pertenecientes a este conjunto permanecerá inalterado en el siguiente paso de tiempo.

Obsérvese que aunque podemos determinar el conjunto de células que posiblemente cambiarán su estado, no sabremos si cada una de ellas realmente cambia el estado o no hasta que no evaluemos su función de transición de estado.

2.2.3. Autómatas conmutados

Los *autómatas celulares* son uniformes tanto en su composición como en el patrón de sus interconexiones. Si eliminamos estas restricciones, considerando aun la

conexión de componentes de estado finito entre sí, obtenemos otra clase de modelos de tiempo discreto.

Los *autómatas conmutados* (también llamados *circuitos digitales*) se construyen a partir de *flip-flops* (también denominados *biestables*) y puertas lógicas. El biestable más sencillo es el tipo D, cuya tabla de transición de estado y salidas es la siguiente:

Estado actual	Entrada actual	Estado siguiente	Salida actual
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

En lugar de permitir que la entrada del flip-flop se propague directamente a su salida, la salida es el estado actual (al igual que en el autómata celular). El estado siguiente es igual a la entrada actual. En la tabla siguiente, se muestra un ejemplo del funcionamiento del biestable D, en el cual se comprueba que la salida sigue a la entrada, pero retrasada un paso en el tiempo:

Tiempo	0	1	2	3	4	5	6	7	8	9	
Trayectoria de entrada	1	0	1	0	1	0	1	0	1	0	
Trayectoria de estado	0	1	0	1	0	1	0	1	0	1	0
Trayectoria de salida	0	1	0	1	0	1	0	1	0	1	

La diferencia entre los modelos de máquinas secuenciales de *Mealy* y de *Moore* es que en los modelos de *Mealy* la entrada puede propagarse instantáneamente a la salida. Esta característica de los modelos de *Mealy* (la propagación a través del espacio en un tiempo cero) permite que la realimentación de la salida a la entrada pueda producir "círculos viciosos".

Para la construcción de redes, además de flip-flops podemos usar puertas lógicas, que permiten implementar funciones booleanas. La salida de las puertas lógicas está determinado por sus entradas, sin intervención de ningún estado interno. El reloj, empleado normalmente en circuitos síncronos, determina el paso de avance en el tiempo, que es constante.

Por ejemplo, en la Figura 2.4 se muestran un autómata conmutado construido conectando en serie varios flip-flops tipo D. La realimentación está definida mediante la conexión de varias puertas XOR. Cada flip-flop es un componente elemental de memoria, como sucedía con las células. Los flip-flops q_1 a q_7 están conectados en serie, es decir, el estado del flip-flop i define el siguiente estado del flip-flop $i-1$.

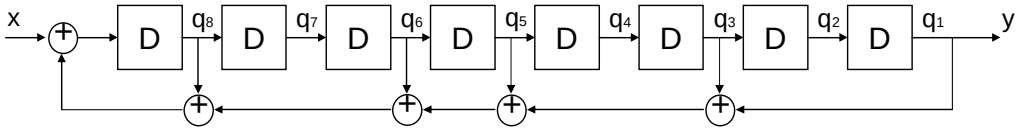

Figura 2.4: Registro de desplazamiento construido conectando flip-flops D y puertas XOR.

Esta secuencia lineal de flip-flops se llama *registro de desplazamiento*, ya que desplaza la entrada hacia el primer componente de la derecha en cada paso de tiempo. La entrada al flip-flop situado más a la izquierda se construye usando puertas lógicas. En este caso, mediante la conexión XOR de estados y de la entrada global: $x \oplus q_8 \oplus q_6 \oplus q_5 \oplus q_3 \oplus q_1$. La salida de una puerta XOR es la operación or-exclusiva de sus entradas: la salida vale 0 si y sólo si ambas entradas tienen el mismo valor; en caso contrario la salida vale 1.

2.2.4. Redes lineales de tiempo discreto

En la Figura 2.5 se muestra una *red de Moore* con dos elementos de memoria y dos elementos sin memoria. A diferencia de los autómatas conmutados descritos en la Sección 2.2.3, que emplean números binarios, la red está definida sobre los números reales.

La estructura de la red puede representarse mediante una matriz, $\begin{pmatrix} 0 & g \\ -g & 0 \end{pmatrix}$, donde g y $-g$ representan las ganancias de los elementos sin memoria. Supongamos que el estado inicial de los elementos con memoria está representado por el vector $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, es decir, cada retardo está inicialmente en el estado 1. Entonces, puede calcularse el estado de la red en el siguiente paso de tiempo, $\begin{pmatrix} q_1 \\ q_2 \end{pmatrix}$, de la forma siguiente:

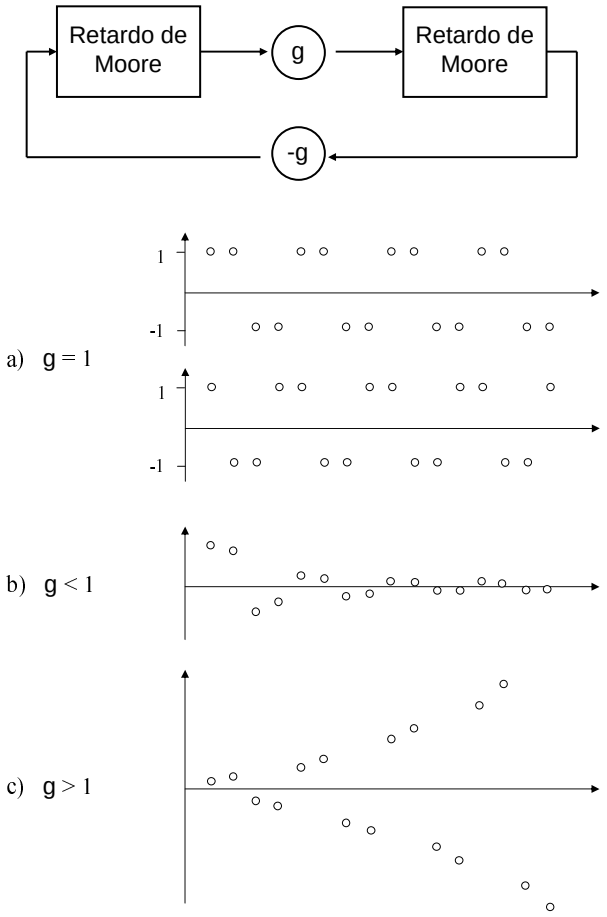

Figura 2.5: Red de Moore lineal y trayectorias del estado en función del valor de g.

$$\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} 0 & g \\ -g & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} g \\ -g \end{pmatrix} \tag{2.5}$$

Es posible emplear una representación matricial y la operación producto porque la red tiene una *estructura lineal*. Esto significa que todos los componentes producen salidas o estados que son combinación lineal de sus entradas y estados.

Un retardo es un elemento lineal muy simple: su siguiente estado es igual a su entrada actual. Un elemento sin memoria con un factor de ganancia se denomina un elemento *coeficiente*, y multiplica su entrada actual por la ganancia para producir la salida. Ambos son ejemplos sencillos de linealidad. Un sumador, que es un elemento sin memoria que suma sus entradas para obtener la salida, es también un elemento lineal.

Un *sistema de tiempo discreto de Moore* está en forma *matricial lineal* si sus funciones de transición de estado y de salida pueden expresarse mediante matrices $\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$. Esto significa que su función de transición de estado puede expresarse de la forma siguiente:

$$\delta(\mathbf{q}, \mathbf{x}) = \mathbf{A}\mathbf{q} + \mathbf{B}\mathbf{x} \tag{2.6}$$

donde $\mathbf{q}$ es un vector n-dimensional de estado y $\mathbf{A}$ es una matriz cuadrada de dimensión $n \times n$. Igualmente, $\mathbf{x}$ es un vector m-dimensional de entrada y $\mathbf{B}$ es una matriz de dimensión $n \times m$. La función de salida es:

$$\lambda(\mathbf{q}) = \mathbf{C}\mathbf{q} \tag{2.7}$$

donde, si la salida $\mathbf{y}$ es un vector p-dimensional, entonces $\mathbf{C}$ es una matriz de dimensión $p \times n$.

La trayectoria en el estado del sistema lineal mostrado en la Figura 2.5 se obtiene multiplicando la matriz $\mathbf{q}$ por los sucesivos estados. Así, si $\begin{pmatrix} g \\ -g \end{pmatrix}$ es el estado que sigue a $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, entonces el siguiente estado es:

$$\begin{pmatrix} 0 & g \\ -g & 0 \end{pmatrix} \begin{pmatrix} g \\ -g \end{pmatrix} = \begin{pmatrix} -g^2 \\ -g^2 \end{pmatrix} \tag{2.8}$$

y así sucesivamente. En este sistema, hay tres tipos diferentes de trayectorias del estado (véase la Figura 2.5):

- Cuando $g = 1$ hay una oscilación indefinida entre los valores 1 y -1.
- Cuando $g < 1$ la envolvente de los valores decae exponencialmente.
- Cuando $g > 1$ la envolvente de los valores crece exponencialmente.

2.3. MODELADO Y SIMULACIÓN DE TIEMPO CONTINUO

En los modelos de tiempo discreto, si se conoce el valor actual de las variables de estado y de las entradas, entonces la *función de transición de estado* permite calcular las variables de estado en el siguiente paso de tiempo.

Por el contrario, los modelos de tiempo continuo no especifican de manera directa el valor de las variables de estado en el siguiente paso de tiempo. En su lugar, el modelo describe la velocidad con la que cambian en el tiempo las variables de estado. La *derivada* de una variable de estado es la velocidad de cambio de dicha variable de estado respecto al tiempo.

En un determinado instante de tiempo, conocido el valor de las variables de estado y de las entradas, podemos calcular el valor de la derivada respecto al tiempo de las variables de estado, es decir, la velocidad a la que cambian las variables de estado. Conocido el valor de las derivadas de las variables de estado, mediante integración numérica se calcula el valor de las variables de estado en el instante futuro de tiempo. A continuación, describiremos todo esto de manera más detallada.

2.3.1. Variables y ecuaciones

Los modelos matemáticos de tiempo continuo están compuestos por *ecuaciones*, que describen la relación entre las magnitudes relevantes del sistema. Estas magnitudes reciben el nombre de *variables*. El siguiente ejemplo pretende ilustrar estos conceptos.

Ejemplo 2.3.1. *Sobre un objeto de masa constante (m) actúan dos fuerzas: la fuerza de la gravedad ($m \cdot g$), y una fuerza armónica de amplitud (F_0) y frecuencia (ω) constantes. La fuerza total (F) aplicada sobre el objeto es:*

$$F = m \cdot g + F_0 \cdot \sin(\omega \cdot t) \tag{2.9}$$

donde g es la aceleración gravitatoria, que se considera constante. Se aplica el criterio siguiente: la aceleración tiene signo positivo cuando tiene sentido vertical ascendente. Por ejemplo, la aceleración gravitatoria terrestre sería $g = -9.8\ m \cdot s^{-2}$

La fuerza neta (F) aplicada sobre el objeto hace que éste adquiera una aceleración (a), que viene determinada por la relación siguiente:

$$m \cdot a = F \tag{2.10}$$

La posición y la velocidad del objeto pueden calcularse teniendo en cuenta que la derivada respecto al tiempo de la posición es la velocidad, y que la derivada respecto al tiempo de la velocidad es la aceleración. El modelo obtenido es el siguiente:

$$F = m \cdot g + F_0 \cdot \sin(\omega \cdot t) \quad (2.11)$$
$$m \cdot a = F \quad (2.12)$$
$$\frac{dx}{dt} = v \quad (2.13)$$
$$\frac{dv}{dt} = a \quad (2.14)$$

Este modelo está compuesto por cuatro ecuaciones, Ecs. (2.11) – (2.14), las cuales describen la relación existente entre las magnitudes relevantes del sistema, que son las variables del modelo:

- *La aceleración gravitatoria (g),*
- *La amplitud (F_0) y frecuencia (ω) de la fuerza armónica,*
- *La fuerza neta (F) aplicada sobre el objeto*
- *La masa (m), posición (x), velocidad (v) y aceleración (a) del objeto.*

□

Parámetros, variables de estado y variables algebraicas

El punto de partida para la simulación del modelo consiste en clasificar sus variables de acuerdo al criterio siguiente:

- **Parámetros**. Son aquellas variables cuyo valor permanece constante durante la simulación.
- **Variables de estado**. Son las variables que están derivadas respecto al tiempo.
- **Variables algebraicas**. Son las restantes variables del modelo. Es decir, aquellas que no aparecen derivadas en el modelo y que no son constantes.

Ejemplo 2.3.2. *De acuerdo con la clasificación anterior, el modelo descrito en el Ejemplo 2.3.1 tiene:*

- *Cuatro parámetros: g, m, F_0, ω.*

– *Dos variables de estado: x, v.*

– *Dos variables algebraicas: a, F.*

Obsérvese que la variable tiempo (t) no se incluye en la clasificación. □

Ejemplo 2.3.3. *Considérese el circuito eléctrico mostrado en la Figura 2.6, el cual está compuesto por un generador de tensión, dos resistencias y un condensador.*

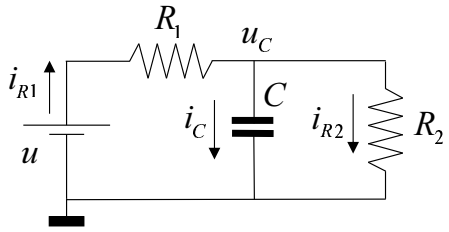

Figura 2.6: Circuito RC.

El modelo de este circuito consta de las ecuaciones siguientes:

$$u = u_0 \cdot \sin(\omega \cdot t) \tag{2.15}$$
$$i_{R1} = i_{R2} + i_C \tag{2.16}$$
$$u - u_C = R_1 \cdot i_{R1} \tag{2.17}$$
$$C \cdot \frac{du_C}{dt} = i_C \tag{2.18}$$
$$u_C = i_{R2} \cdot R_2 \tag{2.19}$$

La Ec. (2.15) es la relación constitutiva del generador de tensión. La amplitud (u_0) y la frecuencia (ω) son independientes del tiempo (t).

Las Ecs. (2.17) y (2.19) son las relaciones constitutivas de las resistencias. La Ec. (2.18) es la relación constitutiva del condensador. Los valores de la capacidad (C) y de las resistencias (R_1, R_2) son independientes del tiempo.

Finalmente, la Ec. (2.16) impone que la suma de las corrientes entrantes a un nodo debe ser igual a la suma de las corrientes salientes del mismo.

Las variables de este modelo se clasifican de la manera siguiente:

– *Parámetros: u_0, ω, C, R_1, R_2.*

- *Variable de estado:* u_C.
- *Variables algebraicas:* u, i_{R1}, i_{R2}, i_C.

□

Ecuaciones

En la siguiente tabla se muestra una clasificación de los modelos de tiempo continuo descritos mediante ecuaciones algebraicas y diferenciales. El vector $\mathbf{x}$ representa las variables del modelo. En la formulación semi-explícita, en la cual las derivadas aparecen despejadas, el vector $\mathbf{x}$ se ha dividido en dos vectores, $\mathbf{x_1}$ y $\mathbf{x_2}$, que representan las variables de estado y algebraicas respectivamente.

	Semi-explícito	**Implícito**
Lineal	$\begin{cases} \dot{\mathbf{x}}_\mathbf{1} + \mathbf{B_{11}}(\mathbf{t})\,\mathbf{x_1}(\mathbf{t}) + \mathbf{B_{12}}(\mathbf{t})\,\mathbf{x_2}(\mathbf{t}) = \mathbf{f_1}(\mathbf{t}) \\ \mathbf{B_{21}}(\mathbf{t})\,\mathbf{x_1}(\mathbf{t}) + \mathbf{B_{22}}(\mathbf{t})\,\mathbf{x_2}(\mathbf{t}) = \mathbf{f_2}(\mathbf{t}) \end{cases}$	$\mathbf{A}(\mathbf{t})\,\dot{\mathbf{x}}(\mathbf{t}) + \mathbf{B}(\mathbf{t})\,\mathbf{x}(\mathbf{t}) = \mathbf{f}(\mathbf{t})$
No lineal	$\begin{cases} \dot{\mathbf{x}}_\mathbf{1}(\mathbf{t}) = \mathbf{f_1}(\mathbf{t}, \mathbf{x_1}(\mathbf{t}), \mathbf{x_2}(\mathbf{t})) \\ \mathbf{0} = \mathbf{f_2}(\mathbf{t}, \mathbf{x_1}(\mathbf{t}), \mathbf{x_2}(\mathbf{t})) \end{cases}$	$\mathbf{F}(\mathbf{t}, \mathbf{x}(\mathbf{t}), \dot{\mathbf{x}}(\mathbf{t})) = \mathbf{0}$

2.3.2. Algoritmo para la simulación

En la Figura 2.7 se muestra un algoritmo para la simulación de modelos de tiempo continuo. Puede comprobarse que la clasificación de las variables del modelo en parámetros, variables de estado y variables algebraicas constituye la base para la simulación del modelo:

- Al comenzar la simulación, se asignan valores a los *parámetros*. Estos valores permanecen constantes durante toda la simulación.
- Las *variables de estado* son calculadas mediante la integración numérica de sus derivadas. Por ejemplo, la función de paso del método explícito de Euler para

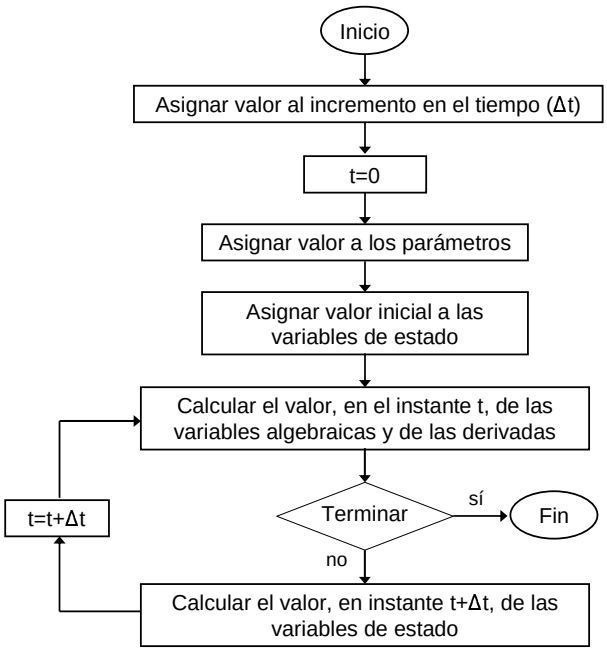

Figura 2.7: Algoritmo de la simulación de los modelos matemáticos de tiempo continuo.

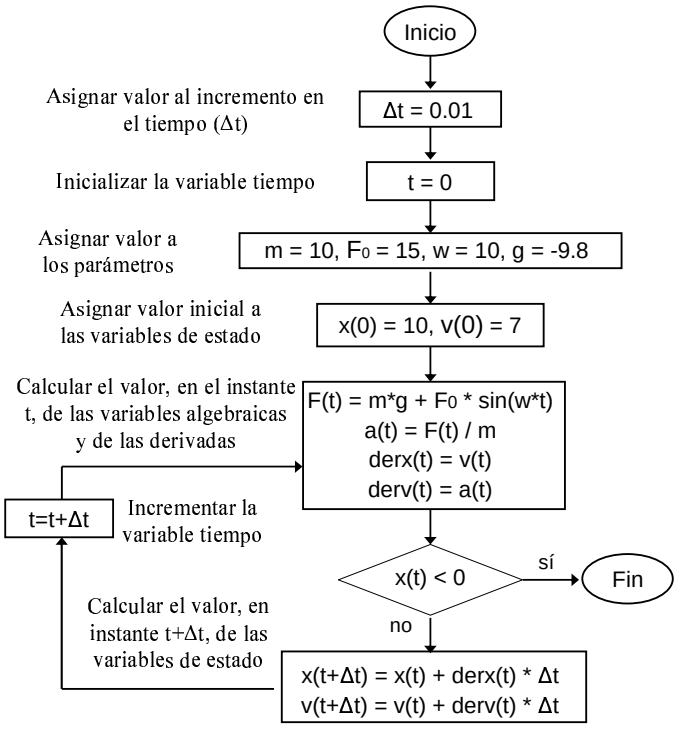

Figura 2.8: Algoritmo de la simulación del modelo descrito en el Ejemplo 2.3.1.

la ecuación diferencial ordinaria

$$\frac{dx}{dt} = f\left(x, t\right) \tag{2.20}$$

es la siguiente:

$$x_{i+1} = x_i + f\left(x_i, t_i\right) \cdot \Delta t \tag{2.21}$$

donde x_i y x_{i+1} representan el valor de la variable de estado x en los instantes t_i y $t_i + \Delta t$ respectivamente, y $f\left(x_i, t_i\right)$ representa el valor de la derivada de x (es decir, $\frac{dx}{dt}$) en el instante t_i.

– El valor de las *variables algebraicas* se calcula, en cada instante de tiempo, a partir de valor de las variables de estado en ese instante y del valor de los parámetros.

La condición de terminación de la simulación depende del estudio en concreto que vaya a realizarse sobre el modelo. Puede ser, por ejemplo, que se alcance determinado valor de la variable tiempo, o que una determinada variable satisfaga cierta condición.

El valor del *tamaño del paso de integración* (Δt) debe escogerse alcanzando un compromiso entre precisión y carga computacional. Cuanto menor sea el valor de Δt, menor es el error que se comete en el cálculo de las variables del modelo, pero mayor es el tiempo de ejecución de la simulación.

Un procedimiento para estimar el error cometido al escoger un determinado valor de Δt es comparar los resultados obtenidos usando ese valor y los obtenidos usando un valor menor, por ejemplo, $\frac{\Delta t}{2}$. Si la diferencia entre ambos resultados es aceptable para los propósitos del estudio de simulación que se está realizando, entonces el valor Δt es adecuado. En caso contrario, se comparan los resultados obtenidos usando $\frac{\Delta t}{2}$ y $\frac{\Delta t}{4}$. Si el error es aceptable, de emplea $\frac{\Delta t}{2}$. Si el error es demasiado grande, se investigan los valores $\frac{\Delta t}{4}$ y $\frac{\Delta t}{8}$, y así sucesivamente.

El método explicado anteriormente es conceptualmente muy sencillo, sin embargo no es eficiente desde el punto de vista computacional. Por ese motivo, los métodos numéricos de paso variable no emplean este procedimiento. Como ilustración de los procedimientos empleados en la práctica, en la Sección 2.3.4 se describirá el método para adaptar el tamaño del paso de integración empleado por el algoritmo de Runge-Kutta-Fehlberg (RKF45) .

Ejemplo 2.3.4. *En la Figura 2.8 se muestra el algoritmo de la simulación del modelo descrito en el Ejemplo 2.3.1. Obsérvese que:*

– *Las derivadas de las variables de estado se han sustituido por variables auxiliares, cuyo nombre es igual al de la variable de estado, pero anteponiéndole el prefijo "der":*

$$\frac{dx}{dt} \rightarrow derx \tag{2.22}$$
$$\frac{dv}{dt} \rightarrow derv \tag{2.23}$$

– *Para realizar el cálculo de las variables algebraicas y las derivadas, se ha despejado de cada ecuación la variable a calcular y se han ordenado las ecuaciones, de modo que sea posible resolverlas en secuencia. Se ha obtenido la siguiente secuencia de asignaciones:*

$$F = m \cdot g + F_0 \cdot \sin(\omega \cdot t) \tag{2.24}$$
$$a = \frac{F}{m} \tag{2.25}$$
$$derx = v \tag{2.26}$$
$$derv = a \tag{2.27}$$

La secuencia de cálculos, según se muestra en la Figura 2.8, es la siguiente:

$t = 0$ $m = 10, F_0 = 15, \omega = 10, g = -9.8$
$x(0) = 10, v(0) = 7$
$F(0) = m \cdot g + F_0 \cdot \sin(\omega \cdot t) \rightarrow F(0) = 10 \cdot (-9.8) + 15 \cdot \sin(10 \cdot 0) = -98$
$a(0) = F(0)/m \rightarrow a(0) = -98/10 = -9.8$
$derx(0) = v(0) \rightarrow derx(0) = 7$
$derv(0) = a(0) \rightarrow derv(0) = -9.8$
$x(0.01) = x(0) + derx(0) \cdot \Delta t \rightarrow x(0.01) = 10 + 7 \cdot 0.01 = 10.07$
$v(0.01) = v(0) + derv(0) \cdot \Delta t \rightarrow v(0.01) = 7 + -9.8 \cdot 0.01 = 6.902$

$t = 0.01$ $F(0.01) = m \cdot g + F_0 \cdot \sin(\omega \cdot t) \rightarrow F(0.01) = 10 \cdot (-9.8) + 15 \cdot \sin(10 \cdot 0.01) = -96.5025$
$a(0.01) = F(0.01)/m \rightarrow a(0.01) = -96.5025/10 = -9.65025$
$derx(0.01) = v(0.01) \rightarrow derx(0.01) = 6.902$
$derv(0.01) = a(0.01) \rightarrow derv(0.01) = -9.65025$
$x(0.02) = x(0.01) + derx(0.01) \cdot \Delta t \rightarrow x(0.02) = 10.07 + 6.902 \cdot 0.01 = 10.139$
$v(0.02) = v(0.01) + derv(0.01) \cdot \Delta t \rightarrow v(0.02) = 6.902 + -9.65025 \cdot 0.01 = 6.8055$

$t = 0.02$ *Y así sucesivamente ...*

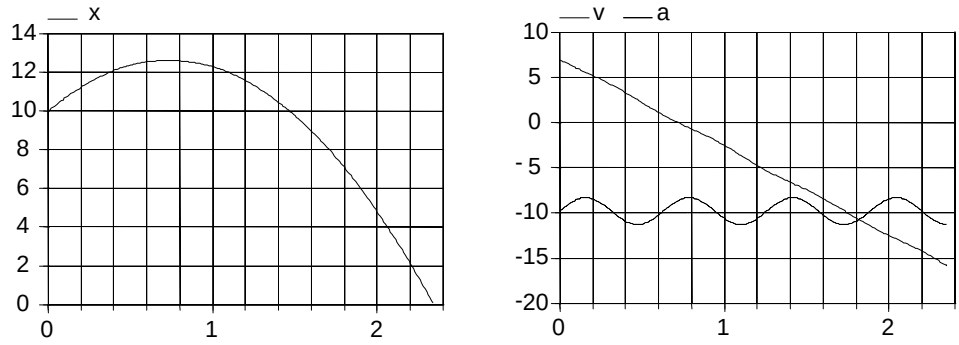

Figura 2.9: Resultado de la ejecución del algoritmo mostrado en la Figura 2.8.

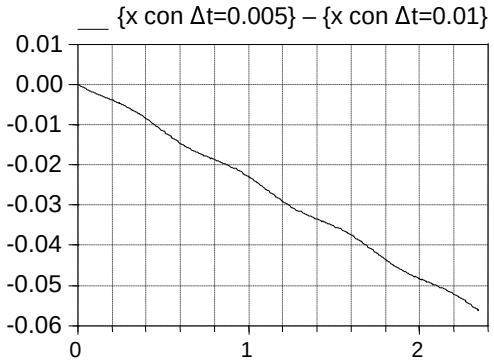

Figura 2.10: Diferencia en la evolución temporal de la posición para dos valores de Δt.

En la Figura 2.9 se muestra la evolución de la posición, la velocidad y la aceleración del objeto, obtenidas ejecutando el algoritmo mostrado en la Figura 2.8. En el eje horizontal de ambas gráficas se representa el tiempo.

La condición de finalización de la simulación es que el objeto toque el suelo. Es decir, que se satisfaga la condición: $x < 0$.

Cabe plantearse si el valor 0.01 *para el tamaño del paso de integración (*Δt*) es adecuado. En la Figura 2.10 se muestra la diferencia entre la posición del objeto, calculada usando* $\Delta t = 0.005$*, y la posición calculada usando* $\Delta t = 0.01$*. En el eje horizontal está representado el tiempo. Esta gráfica permite obtener una idea de cómo va acumulándose el error y cuál es la magnitud de éste. Dependiendo cuál sea el objetivo del estudio de simulación, esta magnitud del error (y consecuentemente, el valor* $\Delta t = 0.01$*) será o no aceptable.* □

2.3.3. Causalidad computacional

Como se ha mostrado en el Ejemplo 2.3.4, para realizar el cálculo de las variables algebraicas y las derivadas, es preciso despejar de cada ecuación la variable a calcular y ordenar las ecuaciones, de modo que sea posible resolverlas en secuencia. Esto tiene un carácter general. Para plantear el algoritmo de la simulación de un modelo de tiempo continuo, es preciso realizar las tareas siguientes:

1. Decidir qué variable debe calcularse de cada ecuación y cómo deben ordenarse las ecuaciones del modelo, de modo que puedan ser resueltas en secuencia. A esta decisión se la denomina *asignación de la causalidad computacional*.
2. Una vez se ha decidido qué variable debe evaluarse de cada ecuación, debe manipularse simbólicamente la ecuación a fin de despejar dicha variable.

Ejemplo 2.3.5. *Supóngase que se desea modelizar una resistencia eléctrica mediante la Ley de Ohm. La Ley de Ohm establece que la caída de potencial, u, entre los bornes de una resistencia es igual al producto de un parámetro característico de la resistencia, R, por la intensidad de la corriente eléctrica, i, que circula a través de la resistencia:*

$$u = i \cdot R \qquad \textit{(Ley de Ohm)} \tag{2.28}$$

Esta ecuación constituye una relación entre las tres variables u, R e i, que es válida para cualquiera de las tres posibles causalidades computacionales admisibles de la ecuación:

$$[u] = i \cdot R \qquad [i] = \frac{u}{R} \qquad [R] = \frac{u}{i} \tag{2.29}$$

donde se ha señalado la variable a evaluar de cada ecuación incluyéndola entre corchetes y se ha despejado escribiéndola en el lado izquierdo de la igualdad. □

La consideración fundamental que debe hacerse llegado este punto es que la causalidad computacional de una determinada ecuación del modelo no sólo depende de ella misma, sino que también depende del resto de las ecuaciones del modelo. Es decir, la **causalidad computacional es una propiedad global del modelo completo**.

Ejemplo 2.3.6. *La casualidad computacional de la ecuación de la resistencia, presentada en el Ejemplo 2.3.5, depende del resto de las ecuaciones del modelo. Si la*

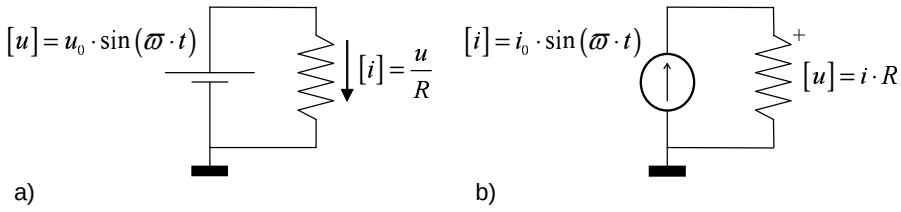

Figura 2.11: La causalidad computacional es una propiedad del modelo completo.

resistencia se conecta a un generador de tensión senoidal, las ecuaciones del modelo son:

$$u = [i] \cdot R \quad \textit{Relación constitutiva de la resistencia} \tag{2.30}$$
$$[u] = u_0 \cdot \sin(\varpi \cdot t) \quad \textit{Relación constitutiva del generador de tensión} \tag{2.31}$$

donde R, u_0 y ϖ son parámetros del modelo, es decir, su valor se supone conocido y no es preciso evaluarlos de las ecuaciones del modelo. En las Ecs. (2.30) y (2.31) se ha señalado entre corchetes la variable que debe evaluarse de cada ecuación.

Para simular el modelo es preciso ordenar las ecuaciones y despejar la variable a evaluar en cada una de ellas. Realizando la ordenación y la manipulación simbólica, se obtiene (véase la Figura 2.11a):

$$[u] = u_0 \cdot \sin(\varpi \cdot t) \tag{2.32}$$
$$[i] = \frac{u}{R} \tag{2.33}$$

Como puede verse, en primer lugar debe calcularse el valor de la caída de tensión, a partir de la relación constitutiva del generador, y a continuación debe calcularse la corriente, a partir de la relación constitutiva de la resistencia (para lo cual debe usarse el valor de la tensión que ha sido previamente calculado).

En cambio, si esta misma resistencia se conecta a un generador sinusoidal de corriente, las ecuaciones del modelo, una vez ordenadas y manipuladas simbólicamente, serán (véase la Figura 2.11b):

$$[i] = i_0 \cdot \sin(\varpi \cdot t) \quad (2.34)$$

$$[u] = i \cdot R \quad (2.35)$$

En este caso, en primer lugar debe calcularse el valor de la corriente, a partir de la relación constitutiva del generador, y a continuación debe calcularse la caída de tensión, a partir de la relación constitutiva de la resistencia (para lo cual debe usarse el valor de la corriente que se ha sido previamente calculado).

Se observa que en el caso de la Ec. (2.33) la relación constitutiva de la resistencia se usa para calcular la corriente, mientras que en el caso de la Ec. (2.35) se usa para calcular la caída de tensión. La variable a evaluar depende no sólo de la relación constitutiva de la resistencia, sino también del resto del modelo. □

A continuación, se describe un procedimiento sistemático para asignar la causalidad computacional de un modelo. Sin embargo, para aplicarlo deben previamente clasificarse las variables del modelo en *conocidas* y *desconocidas*, según sean conocidas o desconocidas en el instante de evaluación. En primer lugar, por tanto, se explicará cómo realizar esta clasificación.

Variables conocidas y desconocidas

A efectos de la asignación de la causalidad computacional, las variables del modelo se clasifican en desconocidas (o incógnitas) y conocidas. Dicha clasificación se realiza en función de que el valor de la variable deba o no ser evaluado de las ecuaciones del modelo. Las **variables conocidas** son las siguientes:

- La variable *tiempo*.
- Los *parámetros del modelo*. La persona que formula el modelo decide qué variables son parámetros. La característica distintiva de este tipo de variables es que no son calculadas de las ecuaciones del modelo, sino que se les asigna valor al inicio de la simulación (en la llamada fase de ïnicialización del modelo") y éste permanece constante durante toda la simulación. Puesto que los parámetros no deben ser calculados de las ecuaciones del modelo, a efectos de la asignación de la causalidad computacional, se considera que los parámetros son variables conocidas.

– Las *entradas globales* al modelo, es decir, aquellas variables cuyo valor se especifica, independientemente del de las demás variables, para cada instante de la simulación.

– Las variables que aparecen derivadas en el modelo, ya que son consideradas *variables de estado*, es decir, se asume que se calculan mediante la integración numérica de su derivada. El motivo es evitar tener que derivar numéricamente en tiempo de simulación.

Por tanto, toda variable que aparece derivada en el modelo se clasifica como una variable de estado. Con el fin de formular el modelo de manera adecuada para su simulación, se sustituye la derivada de cada variable de estado, allí donde aparezca, por una variable auxiliar (por ejemplo, de nombre igual al de la variable de estado, pero anteponiendo el prefijo "der"). Estas variables auxiliares se clasifican como desconocidas y se añade al modelo la condición de que cada variable de estado se calcula por integración numérica de su correspondiente variable auxiliar. Así pues, las **variables desconocidas** son las siguientes:

– Las *variables auxiliares* introducidas, de la forma descrita anteriormente, sustituyendo a las derivadas de las variables de estado.

– Las restantes variables del modelo, es decir, aquellas que, no apareciendo derivadas, dependen para su cálculo en el instante de evaluación del valor de otras variables. Estas variables se denominan *variables algebraicas*.

Ejemplo 2.3.7. *Considérese el circuito mostrado en la Figura 2.6. El modelo está compuesto por las ecuaciones siguientes:*

$$u = u_0 \cdot \sin(\omega \cdot t) \tag{2.36}$$

$$i_{R1} = i_{R2} + i_C \tag{2.37}$$

$$u - u_C = R_1 \cdot i_{R1} \tag{2.38}$$

$$C \cdot \frac{du_C}{dt} = i_C \tag{2.39}$$

$$u_C = i_{R2} \cdot R_2 \tag{2.40}$$

La variable u_C aparece derivada, con lo cual es una variable de estado del modelo.

Con el fin de realizar la asignación de la causalidad computacional, se sustituye en el modelo $\frac{du_C}{dt}$ por la variable auxiliar $deru_C$. Realizando esta sustitución ($\frac{du_C}{dt}$ por $deru_C$), se obtiene el modelo siguiente:

$$u = u_0 \cdot \sin(\omega \cdot t) \quad (2.41)$$
$$i_{R1} = i_{R2} + i_C \quad (2.42)$$
$$u - u_C = R_1 \cdot i_{R1} \quad (2.43)$$
$$C \cdot deru_C = i_C \quad (2.44)$$
$$u_C = i_{R2} \cdot R_2 \quad (2.45)$$

A efectos de la asignación de la causalidad computacional, se considera que:

- *La variable de estado, u_C, es conocida.*
- *La derivada de la variable de estado, $deru_C$, es desconocida.*

Al realizar la simulación, u_C se calculará integrando $deru_C$.

Descontando del número total de ecuaciones el número de variables de estado, se obtiene el número de ecuaciones disponibles para el cálculo de variables algebraicas. Este número determina el número de variables algebraicas del modelo. El resto de las variables deberán ser parámetros. La persona que realiza el modelo deberá decidir, de entre las variables que no son estados, cuáles son las variables algebraicas y cuáles los parámetros.

En este ejemplo sólo hay una variable de estado: u_C. Así pues, debe emplearse una de las ecuaciones del modelo para evaluar su derivada: $deru_C$. Puesto que $deru_C$ sólo interviene en la Ec. (2.44), debe emplearse esta ecuación para calcular $deru_C$.

Las restantes cuatro ecuaciones permiten calcular cuatro variables algebraicas. Una posible elección de las variables algebraicas es la siguiente: u, i_{R1}, i_{R2}, i_C. En consecuencia, las variables desconocidas son las siguientes: u, i_{R1}, i_{R2}, i_C y $deru_C$. El resto de las variables del modelo deberán ser parámetros: u_0, ω, R_1, R_2 y C.

Este modelo tiene sólo 5 ecuaciones y 5 incógnitas, con lo cual la causalidad computacional puede asignarse de manera sencilla (en el siguiente epígrafe se explica un método sistemático para hacerlo). Asignar la causalidad computacional es decidir en qué orden deben evaluarse las ecuaciones y qué incógnita debe evaluarse de cada ecuación, a fin de calcular las incógnitas u, i_{R1}, i_{R2}, i_C y $deru_C$ del conjunto de Ecuaciones (2.41) – (2.45).

A continuación, se señala en cada ecuación del modelo qué incógnita debe calcularse de ella. Para ello, se incluye entre corchetes la incógnita a evaluar de cada ecuación:

$$
\begin{aligned}
[u] &= u_0 \cdot \sin(\omega \cdot t) && (2.46)\\
i_{R1} &= i_{R2} + [i_C] && (2.47)\\
u - u_C &= R_1 \cdot [i_{R1}] && (2.48)\\
C \cdot [deru_C] &= i_C && (2.49)\\
u_C &= [i_{R2}] \cdot R_2 && (2.50)
\end{aligned}
$$

Ordenando las ecuaciones del modelo, se obtiene:

$$
\begin{aligned}
[u] &= u_0 \cdot \sin(\omega \cdot t) && (2.51)\\
u_C &= [i_{R2}] \cdot R_2 && (2.52)\\
u - u_C &= R_1 \cdot [i_{R1}] && (2.53)\\
i_{R1} &= i_{R2} + [i_C] && (2.54)\\
C \cdot [deru_C] &= i_C && (2.55)
\end{aligned}
$$

Finalmente, despejando en cada ecuación la incógnita que debe ser evaluada de ella, se obtiene el modelo ordenado y resuelto:

$$
\begin{aligned}
[u] &= u_0 \cdot \sin(\omega \cdot t) && (2.56)\\
[i_{R2}] &= \frac{u_C}{R_2} && (2.57)\\
[i_{R1}] &= \frac{u - u_C}{R_1} \cdot && (2.58)\\
[i_C] &= i_{R1} - i_{R2} && (2.59)\\
[deru_C] &= \frac{i_C}{C} && (2.60)
\end{aligned}
$$

Además, debe realizarse el cálculo de la variable de estado mediante la integración de su derivada:

$$\frac{d[u_C]}{dt} = deru_C \qquad (2.61)$$

Obsérvese que, en general, la clasificación de las variables en algebraicas y parámetros puede realizarse de varias maneras, en función de cuál sea el experimento que desee realizarse sobre el modelo.

Para comprobar si una determinada clasificación es válida, es preciso realizar la asignación de la causalidad computacional, y comprobar que efectivamente las variables algebraicas seleccionadas pueden evaluarse de las ecuaciones del modelo.

Una selección de las variables algebraicas alternativa a la anterior consistiría en escoger R_1 y R_2 como variables algebraicas, en lugar de i_{R1} e i_{R2}. En este caso, i_{R1} e i_{R2} serían parámetros. □

Asignación de la causalidad computacional

A continuación, se describe un procedimiento sistemático para asignar la causalidad computacional de un modelo. Consta de dos pasos. En primero consiste en comprobar que el modelo no es estructuralmente singular. El segundo paso es la asignación en sí de la causalidad.

Paso 1 - Singularidad estructural del modelo. Antes de realizar la partición se comprueba la no *singularidad estructural* del modelo. Es decir, se comprueba que:

- El número de ecuaciones y de incógnitas (obtenido siguiendo el criterio anterior de clasificación de las variables en conocidas y desconocidas) es el mismo.
- Cada incógnita puede emparejarse con una ecuación en que aparezca y con la cual no se haya emparejado ya otra incógnita.

Si alguna de estas dos condiciones no se verifica, se dice que el modelo es *singular* y es necesario reformularlo para poder simularlo. Si el modelo no es singular, se procede a asignar la casualidad computacional.

Ejemplo 2.3.8. *Considere un modelo con tres variables: x, y, z. El modelo está compuesto por tres ecuaciones, en cada una de las cuales intervienen las variables siguientes:*

$$f_1(x, y, z) = 0 \tag{2.62}$$

$$f_2\left(x, \frac{dx}{dt}, y\right) = 0 \tag{2.63}$$

$$f_3(x, z) = 0 \tag{2.64}$$

Sustituyendo la derivada de la variable x por la variable auxiliar ($derx$), se obtiene un modelo compuesto por las tres ecuaciones siguientes:

$$f_1(x, y, z) = 0 \tag{2.65}$$
$$f_2(x, derx, y) = 0 \tag{2.66}$$
$$f_3(x, z) = 0 \tag{2.67}$$

Dicho modelo tiene tres incógnitas: $derx$, y, z. La variable x, por aparecer derivada, se supone que es una variable de estado: se calcula integrando $derx$, y no de las ecuaciones del modelo.

Se comprueba que este modelo no es singular, ya que cada ecuación puede asociarse con una incógnita que interviene en ella y que no se ha asociado con ninguna otra ecuación (obsérvese que, en general, esta asociación puede no ser única):

$$f_1(x, y, z) = 0 \quad \rightarrow \quad y \tag{2.68}$$
$$f_2(x, derx, y) = 0 \quad \rightarrow \quad derx \tag{2.69}$$
$$f_3(x, z) = 0 \quad \rightarrow \quad z \tag{2.70}$$

□

Ejemplo 2.3.9. *Considere un modelo con tres variables: x, y, z. El modelo está compuesto por tres ecuaciones, en cada una de las cuales intervienen las variables siguientes:*

$$f_1(z) = 0 \tag{2.71}$$
$$f_2\left(\frac{dx}{dt}, y\right) = 0 \tag{2.72}$$
$$f_3(x) = 0 \tag{2.73}$$

Dicho modelo tiene tres incógnitas: $derx$, y, z. Observe que la tercera ecuación ($f_3(x) = 0$) no contiene ninguna incógnita, mientras que la segunda ecuación contiene dos incógnitas que no aparecen en ninguna otra ecuación del modelo ($f_2(derx, y) = 0$). En consecuencia, el modelo es singular y por tanto es necesario reformularlo. □

Paso 2 - Asignación de la causalidad computacional. Una vez se ha comprobado que el modelo no es *singular*, se realiza la asignación de causalidad computacional siguiendo las tres reglas siguientes:

1. Las variables que aparecen derivadas se consideran variables de estado y se suponen conocidas, ya que se calculan por integración a partir de sus derivadas. Las derivadas de las variables de estado son desconocidas y deben calcularse de las ecuaciones en que aparezcan.

2. Las ecuaciones que poseen una única incógnita deben emplearse para calcularla.

3. Aquellas variables que aparecen en una única ecuación deben ser calculadas de ella.

Aplicando las tres reglas anteriores a sistemas no singulares, pueden darse las dos situaciones siguientes:

1. Se obtiene una solución que permite calcular todas las incógnitas usando para ello todas las ecuaciones. Esto significa que las variables pueden ser resueltas, una tras otra, en secuencia. En este caso, el algoritmo proporciona una ordenación de las ecuaciones tal que en cada ecuación hay una y sólo una incógnita que no haya sido previamente calculada. En ocasiones será posible despejar la incógnita de la ecuación. En otros casos, la incógnita aparecerá de forma implícita y deberán emplearse métodos numéricos para evaluarla.

2. Se llega a un punto en que todas las ecuaciones tienen al menos dos incógnitas y todas las incógnitas aparecen al menos en dos ecuaciones. Corresponde al caso en que hay sistema de ecuaciones. Si en este sistema las incógnitas intervienen linealmente, será posible despejarlas resolviendo el sistema simbólicamente. Si al menos una de las incógnitas interviene de forma no lineal, deberán emplearse métodos numéricos para evaluar las incógnitas. El algoritmo de partición asegura que la dimensión de los sistemas de ecuaciones obtenidos es mínima.

Ejemplo 2.3.10. *A continuación, se realiza la partición del modelo descrito en el Ejemplo 2.3.8. Indicando únicamente las incógnitas que intervienen en cada ecuación se obtiene:*

$$f_1(y, z) = 0 \tag{2.74}$$
$$f_2(derx, y) = 0 \tag{2.75}$$
$$f_3(z) = 0 \tag{2.76}$$

La variable z es la única incógnita que aparece en la ecuación f_3, por tanto debe emplearse esta ecuación para calcular z. Las ecuaciones que quedan por emplear, y

las incógnitas que quedan por evaluar son:

$$f_1(y) = 0 \tag{2.77}$$
$$f_2(derx, y) = 0 \tag{2.78}$$

Debe emplearse f_1 para calcular y. Seguidamente, debe emplearse f_2 para calcular $derx$. Las ecuaciones ordenadas, con la causalidad computacional señalada, son las siguientes:

$$f_3(x, [z]) = 0 \tag{2.79}$$
$$f_1(x, [y], z) = 0 \tag{2.80}$$
$$f_2(x, [derx], y) = 0 \tag{2.81}$$

□

2.3.4. Métodos de integración numérica

Como hemos visto anteriormente, las ecuaciones del modelo permiten calcular las variables algebraicas y las derivadas de las variables de estado. En esta sección analizaremos diferentes formas de calcular las variables de estado a partir del valor de sus derivadas. El cálculo se realiza mediante la aplicación de algún *método numérico de integración.*

Una vez conocidas en el instante t_i las variables de estado, las variables algebraicas y las derivadas de las variables de estado, el problema consiste en calcular la variables de estado en el siguiente instante, t_{i+1}. Se supone que el modelo es continuo en el intervalo $[t_i, t_{i+1}]$, y que las entradas, el estado y las variables de salida cambian de manera continua en ese intervalo de tiempo.

A continuación, se describen cuatro métodos de integración numérica sencillos. Los cuatro métodos pertenecen a la categoría de los *métodos explícitos de integración.* Estos son:

- Euler explícito
- Punto medio (Euler-Richardson)
- Runge-Kutta (4° orden)
- Runge-Kutta-Fehlberg (4°-5° orden)

En los tres primeros métodos, el paso de integración se escoge antes de iniciar la simulación y se mantiene fijo durante toda la simulación. Por el contrario, el método de Runge-Kutta-Fehlberg es de paso variable. Es decir, el algoritmo de integración va estimando el error cometido, y reduce o aumenta en consecuencia el tamaño del paso de integración.

Método de Euler explícito

La función de paso del método de Euler para la ecuación diferencial ordinaria

$$\frac{dx}{dt} = f\left(x, t\right) \tag{2.82}$$

es la siguiente:

$$x_{i+1} = x_i + f\left(x_i, t_i\right) \cdot \Delta t \tag{2.83}$$

que corresponde con un desarrollo de Taylor truncado en el primer orden.

Ejemplo 2.3.11. *El modelo cinemático de una masa puntual (m), sobre la que actúa una fuerza (F), consta de las ecuaciones siguientes:*

$$v = \frac{dy}{dt} \tag{2.84}$$

$$a = \frac{dv}{dt} \tag{2.85}$$

$$a = \frac{F\left(y, v, t\right)}{m} \tag{2.86}$$

El algoritmo para integrar este modelo, empleando el método de Euler explícito, es el siguiente:

1. *Deben conocerse los valores de las variables de estado (y, v) en el instante inicial de la simulación (t_0). Sean estos valores: y_0, v_0.*

   También debe fijarse el tamaño del paso de integración (Δt).

2. *Cálculo de la aceleración de la partícula (a_i), que es una variable algebraica:*

$$a_i = \frac{F\left(y_i, v_i, t_i\right)}{m} \tag{2.87}$$

3. Se calcula el valor de las variables de estado en el nuevo instante de tiempo:

$$v_{i+1} = v_i + a_i \cdot \Delta t \tag{2.88}$$
$$y_{i+1} = y_i + v_i \cdot \Delta t \tag{2.89}$$

La velocidad al final del intervalo (v_{i+1}) se calcula de la aceleración al principio del intervalo (a_i). Igualmente, la posición al final del intervalo (y_{i+1}) se calcula de la aceleración al principio del intervalo (v_i).

4. Avance de un paso en el tiempo e incremento del índice i:

$$t_{i+1} = t_i + \Delta t \tag{2.90}$$
$$i = i + 1 \tag{2.91}$$

5. Volver al Paso 2.

□

Método de Euler-Richardson

El algoritmo de Euler-Richardson usa la derivada en el principio del intervalo para estimar el valor de la variable de estado en el punto medio del intervalo ($t_{med} = t + \frac{\Delta t}{2}$). A continuación, emplea la derivada en este punto medio para calcular la variable de estado al final del intervalo. Este algoritmo también se conoce como algoritmo de Runge-Kutta de 2° orden o del punto medio.

La función de paso del método de Euler-Richardson para la ecuación diferencial ordinaria

$$\frac{dx}{dt} = f(x, t) \tag{2.92}$$

es la siguiente:

$$x_{med} = x_i + f(x_i, t_i) \cdot \frac{\Delta t}{2} \tag{2.93}$$
$$x_{i+1} = x_i + f\left(x_{med}, t_i + \frac{\Delta t}{2}\right) \cdot \Delta t \tag{2.94}$$

Ejemplo 2.3.12. *El modelo de la masa puntual descrito en el Ejemplo 2.3.11 puede integrarse, empleando el algoritmo de Euler-Richardson, de la forma siguiente:*

$$v_{med} = v_i + a_i \cdot \frac{\Delta t}{2} \tag{2.95}$$
$$y_{med} = y_i + v_i \cdot \frac{\Delta t}{2} \tag{2.96}$$
$$a_{med} = \frac{F\left(y_{med}, v_{med}, t_i + \frac{\Delta t}{2}\right)}{m} \tag{2.97}$$

y

$$v_{i+1} = v_i + a_{med} \cdot \Delta t \tag{2.98}$$
$$y_{i+1} = y_i + v_{med} \cdot \Delta t \tag{2.99}$$

□

Método de Runge-Kutta (4° orden)

Este método se emplea con mucha frecuencia. Requiere cuatro evaluaciones de la derivada por cada paso en el tiempo. Se denomina de 4° orden porque considera el desarrollo de Taylor hasta 4° orden. La función de paso para la ecuación diferencial ordinaria

$$\frac{dx}{dt} = f(x, t) \tag{2.100}$$

es la siguiente:

$$k_1 = \Delta t \cdot f(x_i, t_i) \tag{2.101}$$
$$k_2 = \Delta t \cdot f\left(x_i + \frac{k_1}{2}, t_i + \frac{\Delta t}{2}\right) \tag{2.102}$$
$$k_3 = \Delta t \cdot f\left(x_i + \frac{k_2}{2}, t_i + \frac{\Delta t}{2}\right) \tag{2.103}$$
$$k_4 = \Delta t \cdot f(x_i + k_3, t_i + \Delta t) \tag{2.104}$$
$$x_{i+1} = x_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \tag{2.105}$$

Método de Runge-Kutta-Fehlberg (4°-5° orden)

Se trata de un método de 4° orden embebido dentro de un método de 5° orden. El error cometido al aplicar el método de cuarto orden se obtiene restando las funciones

paso de ambos métodos, y corresponderá con el término de 5° orden del desarrollo de Taylor ($\frac{d^5 x_i}{dt^5} \cdot \frac{\Delta t^5}{5!}$).

La función de paso para la ecuación diferencial ordinaria

$$\frac{dx}{dt} = f(x,t) \tag{2.106}$$

es la siguiente:

$$\begin{aligned}
k_1 &= \Delta t \cdot f(x_i, t_i) && (2.107)\\
k_2 &= \Delta t \cdot f\left(x_i + \frac{k_1}{4}, t_i + \frac{\Delta t}{4}\right) && (2.108)\\
k_3 &= \Delta t \cdot f\left(x_i + \frac{3}{32}\cdot k_1 + \frac{9}{32}\cdot k_2, t_i + \frac{3}{8}\cdot \Delta t\right) && (2.109)\\
k_4 &= \Delta t \cdot f\left(x_i + \frac{1932}{2197}\cdot k_1 - \frac{7200}{2197}\cdot k_2 + \frac{7296}{2197}\cdot k_3, t_i + \frac{12}{13}\cdot \Delta t\right) && (2.110)\\
k_5 &= \Delta t \cdot f\left(x_i + \frac{439}{216}\cdot k_1 - 8\cdot k_2 + \frac{3680}{513}\cdot k_3 - \frac{845}{4104}\cdot k_4, t_i + \Delta t\right) && (2.111)\\
k_6 &= \Delta t \cdot f\left(x_i - \frac{8}{27}\cdot k_1 + 2\cdot k_2 - \frac{3544}{2565}\cdot k_3 + \frac{1859}{4104}\cdot k_4 \right. && (2.112)\\
&\qquad \left. -\frac{11}{40}\cdot k_5, t_i + \frac{\Delta t}{2}\right) && (2.113)
\end{aligned}$$

La fórmula de paso de 4° orden es:

$$x_{i+1,\ 4^\circ\ \text{orden}} = x_i + \frac{25}{216}\cdot k_1 + \frac{1408}{2565}\cdot k_3 + \frac{2197}{4104}\cdot k_4 - \frac{k_5}{5} \tag{2.114}$$

y la de 5° orden es:

$$x_{i+1,\ 5^\circ\ \text{orden}} = x_i + \frac{16}{135}\cdot k_1 + \frac{6656}{12825}\cdot k_3 + \frac{28561}{56430}\cdot k_4 - \frac{9}{50}\cdot k_5 + \frac{2}{55}\cdot k_6 \tag{2.115}$$

En cada paso de integración, se compara la solución obtenida aplicando el método de 4° orden con la solución obtenida aplicando el método de 5° orden. La finalidad es determinar si el tamaño del paso que se está usando, Δt, es el adecuado. Conceptualmente el procedimiento es el siguiente. Si ambas soluciones no se encuentran suficientemente próximas, se reduce el tamaño del paso de integración. Si ambas

soluciones coinciden en más dígitos significativos de los necesarios, se aumenta el paso de integración.

2.4. MODELADO Y SIMULACIÓN DE EVENTOS DISCRETOS

En la Sección 2.2 se discutió cómo aplicar una estrategia de *eventos discretos* a la simulación de autómatas celulares, constituyendo una forma más eficiente de realizar la simulación. En esta sección se considerará el *modelado de eventos discretos* como un paradigma en sí mismo. En primer lugar, estudiaremos una formalización del autómata celular de eventos discretos. Finalmente, consideraremos el tipo de modelos más habitualmente asociados con la simulación de eventos discretos.

2.4.1. Autómata celular de eventos discretos

La versión original del *Juego de la Vida* asume que todos los nacimientos y las muertes tardan el mismo tiempo en producirse: un paso de tiempo. Otra posible representación del ciclo de vida de una célula consiste en suponer que el nacimiento y la muerte dependen de una magnitud, denominada *factor ambiental*, que representa en qué medida el entorno le resulta favorable u hostil a la célula.

El entorno resulta favorable para la célula cuando tiene exactamente 3 células vecinas. En este caso, el *factor ambiental* toma un valor positivo. Por el contrario, esta magnitud disminuye rápidamente cuando el entorno se torna hostil y cuando alcanza el valor 0, la célula muere. En la Figura 2.12 se muestra un ejemplo de comportamiento del modelo de la célula en función de su entorno, es decir, en función del número de células vecinas que están vivas.

Tal como se muestra en la Figura 2.12, cuando el número de células vecinas vivas cambia de 4 a 3, el *factor ambiental* comienza a crecer linealmente y en el instante en que el factor ambiental cruza el cero, la célula nace. El *factor ambiental* sigue creciendo hasta que alcanza un valor de saturación, que en este ejemplo se ha considerado que es 6. Por el contrario, cuando el número de células vecinas vivas cambia de 3 a 4, el entorno se vuelve hostil, con lo cual el *factor ambiental* comienza a decrecer linealmente y cuando alcanza el valor 0, se produce la muerte de la célula y el *factor ambiental* pasa de manera discontinua a valer -2, que es el menor valor posible del *factor ambiental*.

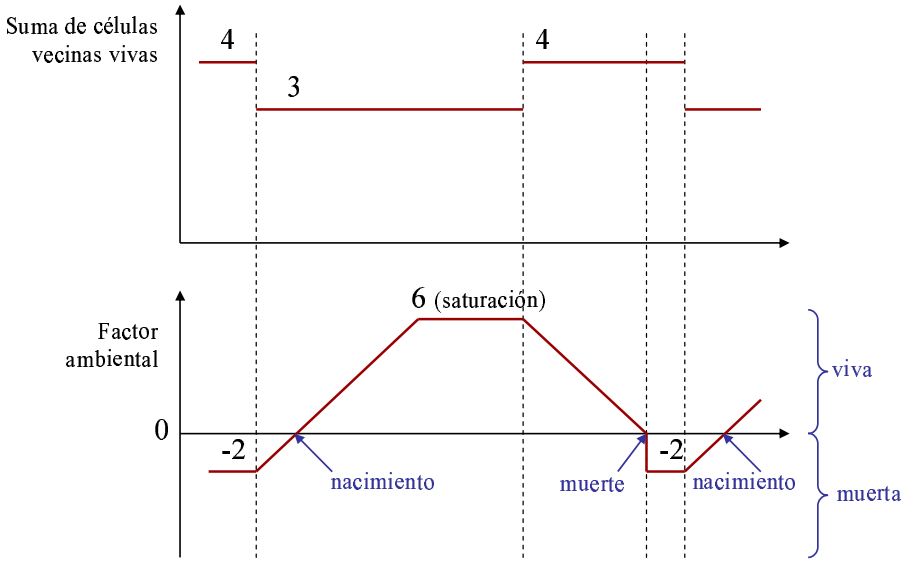

Figura 2.12: Comportamiento del *Juego de la Vida* considerando el *factor ambiental*.

Hay varias formas de modelar el comportamiento de la célula. Una sería calcular la trayectoria del *factor ambiental* y detectar cuándo se produce el cruce por cero, es decir, el nacimiento y la muerte. Este procedimiento sería una combinación de simulación de tiempo continuo y tiempo discreto. Es decir, se trata de un *modelo híbrido*. Sin embargo, este procedimiento es computacionalmente poco eficiente, ya que hay que calcular el valor del *factor ambiental* en cada punto de su trayectoria.

Un procedimiento mucho más eficiente es realizar un *modelo orientado a los eventos*, en el cual nos concentramos únicamente en los eventos de interés: los nacimientos y las muertes, así como los cambios en el número de células vecinas vivas. Un modelo de eventos discretos salta de un evento al siguiente, omitiendo describir el comportamiento sin interés que se produce entremedias.

El requisito para poder realizar un modelo de eventos discretos es tener la capacidad de predecir cuándo se producen los eventos de interés. Los eventos pueden clasificarse en externos e internos.

- *Eventos externos*. Son los eventos generados por el entorno. Por ejemplo, cuando el número de células vecinas cambia. El suceso de este tipo de eventos no está bajo el control del componente modelado (en este caso, de la célula).
- *Eventos internos*. Por otra parte, el componente debe planificar el suceso de cierto tipo de eventos. La ocurrencia de estos eventos, denominados *internos*,

es planificada por el propio componente. Dado un estado en concreto de la célula, es decir, un valor particular de su *factor ambiental*, puede calcularse el tiempo que debe transcurrir hasta que ocurra el siguiente *evento interno* (nacimiento o muerte), supuesto que mientras tanto no haya sucedido ningún evento externo.

Puesto que el valor del *factor ambiental* varía de forma lineal, los instantes de nacimiento y muerte pueden ser calculados de manera muy sencilla. Cuando transcurre el tiempo hasta el siguiente *evento interno* (véase la Figura 2.13), se actualiza el valor del estado de la célula, y se continua avanzando hasta el instante en que se produce el siguiente evento.

En la simulación de eventos discretos del modelo del *Juego de la Vida*, deben ejecutarse los *eventos internos* de las diferentes células en sus respectivos instantes de disparo. Cuando se cambia el estado de una célula debido a un *evento interno*, hay que examinar las células vecinas para ver si se produce algún cambio de estado. Un cambio en el estado puede producir la programación de nuevos eventos y la cancelación de eventos que estaban programados.

En la Figura 2.14 se ilustra este proceso. Supongamos que todas las células que aparecen sombreadas en la Figura 2.14a están vivas. La célula que está en la casilla (0,0) tiene tres vecinas vivas, así que puede nacer. Por el contrario, la célula situada en la casilla $(-1,-1)$ tiene sólo una vecina viva, con lo cual morirá. Por tanto, se programa el nacimiento y la muerte de estas dos células en los instantes t_1 y t_2, respectivamente. Obsérvese que en la Figura 2.14a se ha señalado en cada célula el instante en que está programado el suceso de su siguiente *evento interno*.

En el instante t_1, la célula situada en (0,0) nace. Se ha señalado esta célula con una estrella en la Figura 2.14b. Como resultado de ello, las células situadas en (0,1), $(0,-1)$ y (1,1) pasan a tener 3 vecinas vivas, con lo cual se programa el evento futuro de su nacimiento, que se producirá en el instante t_3. La célula $(-1,1)$ tiene 4 células vecinas vivas, con lo cual se programa su muerte para el instante t_3.

La célula $(-1,-1)$ tenía un evento planificado para el instante t_2: su muerte. Sin embargo, debido al nacimiento de la célula (0,0), ahora tiene dos células vecinas vivas, con lo cual ya no se reúnen la condiciones para que muera: se cancela el evento planificado para t_2. Como vemos, el efecto de las transiciones de estado no sólo es planificar eventos futuros, sino también puede ser cancelar eventos ya planificados.

A continuación, el reloj de la simulación salta hasta el siguiente evento planificado, que sucede en t_3. El hecho de saltar de un evento al siguiente, sin considerar el intervalo de tiempo entre ambos, es una ventaja de la simulación de eventos discretos

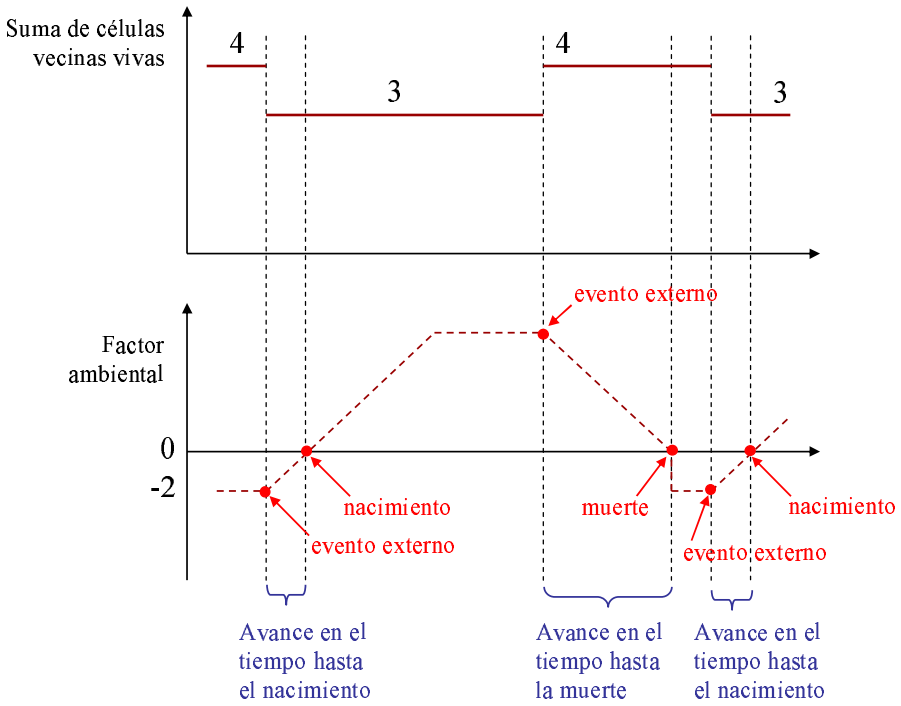

Figura 2.13: Comportamiento del *Juego de la Vida* considerando el *factor ambiental* y empleando simulación de eventos discretos.

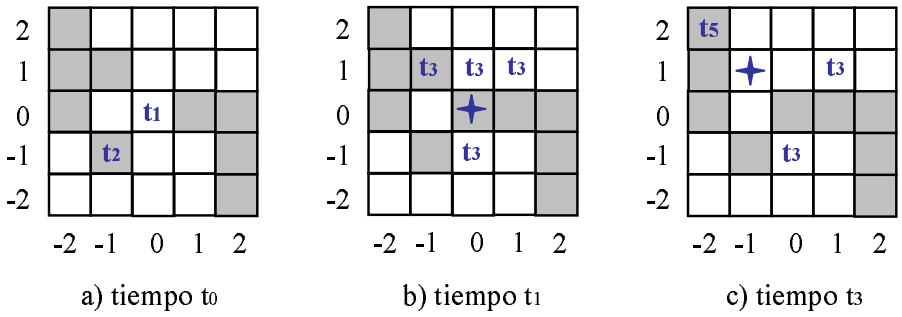

Figura 2.14: Ejemplo de procesamiento de los eventos en el *Juego de la Vida*.

en lo relativo a su eficiencia computacional. Este mecanismo contrasta con el de la *simulación de tiempo discreto*, en el cual las células se inspeccionan en cada paso de tiempo.

La situación en t_3 ilustra un problema que acontece en la simulación de eventos discretos: los *eventos simultáneos*. En la Figura 2.14b vemos que hay varios eventos planificados para el instante t_3: el nacimiento de (0,1), (0,−1) y (1,1), y la muerte de (−1,1). la cuestión es qué evento se ejecuta primero y cuál es el resultado.

- Si en primer lugar (−1,1) muere, entonces deja de satisfacerse la condición de que (0,1) tiene tres células vecinas vivas, con lo cual se cancelaría el evento del nacimiento de (0,1). Véase la Figura 2.14c. Se ha señalado con una estrella la célula (−1,1), que muere, y se ha cancelado el evento en el instante t_3 asociado a (0,1).
- Sin embargo, si en primer lugar se dispara el evento interno de (0,1), entonces efectivamente se produce el nacimiento de (0,1).

En consecuencia, *el resultado es diferente dependiendo del orden de activación de los eventos*. Hay varias posibles soluciones al problema de los eventos simultáneos. Una de ellas es realizar simultáneamente todas las transiciones asociadas a los eventos simultáneos. Como veremos, éste es el procedimiento empleado en el formalismo *DEVS paralelo*. El método empleado por la mayoría de los paquetes de simulación y por el formalismo *DEVS clásico* es definir una prioridad entre los componentes, de tal modo que se dispara el evento del componente con mayor prioridad.

A continuación, se describe el método más comúnmente empleado para la simulación de los modelos de eventos discretos: la *planificación de eventos*. Este método puede emplearse para simular el *Juego de la Vida*.

2.4.2. Simulación de eventos discretos

En los modelos de eventos discretos, los cambios en el estado del sistema físico se representan mediante una serie de cambios discretos o *eventos* en instantes específicos de tiempo. Dado que las variables y atributos permanecen constantes entre eventos consecutivos, los cambios en las variables necesarios para activar un evento en el estado sólo pueden producirse como resultado de la ejecución de un evento en el tiempo. Entre eventos, las variables de estado del sistema permanecen constantes.

La planificación de los eventos en el tiempo es muy sencilla, ya que se sabe con antelación en qué instantes se van a producir. Durante el curso de la simulación se lleva un registro de los instantes de activación de los eventos en el tiempo. Éste se denomina el *calendario de eventos*.

En el instante de inicio de la simulación, se activa el evento "Inicio de la Simulación". Como parte de las acciones asociadas a la ejecución de este evento, se pone el *reloj de la simulación* a cero y se planifican determinados eventos para su ejecución en instantes futuros. Estos eventos en el tiempo son registrados en el *calendario de eventos*, ordenados de menor a mayor instante de ejecución.

Una vez ejecutado este primer evento, el reloj de la simulación es avanzado hasta el instante de ejecución del primer evento del calendario, el cual es entonces borrado del calendario. Se ejecuta el evento, actualizándose, si procede, el calendario de eventos. A continuación, el reloj de la simulación salta hasta el instante de ejecución del evento más próximo (el primero del calendario), éste es borrado del calendario, es ejecutado y, si procede, se actualiza el calendario. Se procede de esta manera hasta que se activa el evento "Finalización de la Simulación". Una vez éste es ejecutado, finaliza la simulación.

En función de su condición de activación, los eventos pueden clasificarse en *eventos en el tiempo* y *eventos en el estado*:

- Los *eventos en el tiempo* son aquellos cuya condición de activación es que el reloj de la simulación (es decir, el tiempo simulado) alcance un determinado valor.
- La condición de activación de los *eventos en el estado* no es función exclusiva del tiempo, sino que también es función de variables del sistema.

Los modelos *orientados a los eventos* únicamente contienen eventos en el tiempo, es decir, la condición de disparo de los eventos es que el *reloj de la simulación* alcance determinado valor. No se contempla la posibilidad de que los eventos puedan dispararse como resultado de inspeccionar el estado del modelo y comprobar que se satisfacen determinadas condiciones de disparo. Es decir, en los modelos de eventos discretos no hay *eventos en el estado*.

Planificar un evento, es decir, determinar con antelación el instante de tiempo en el cual debe dispararse, es sencillo cuando pueden conocerse con antelación todas las condiciones para su disparo. Sin embargo, éste no es siempre el caso.

Ejemplo 2.4.1. *Va a emplearse un modelo sencillo para ilustrar la práctica de la simulación orientada a los eventos: el modelo de una oficina de atención al público*

atendida por un único empleado. La estructura lógica del modelo es la siguiente (véase la Figura 2.15):

- *Si llega un nuevo cliente y el empleado está ocupado, el cliente se pone al final de una cola FIFO en espera de su turno. Si el empleado está libre, el cliente es atendido inmediatamente.*
- *Si el empleado termina de atender a un cliente, éste se marcha y comienza a ser atendido el primer cliente de la cola. Si la cola está vacía, el empleado permanece desocupado hasta la llegada de un nuevo cliente.*

El intervalo de tiempo entre llegadas de clientes y el tiempo de servicio son variables aleatorias, con probabilidad distribuida exponencialmente, expo(10 minutos), y uniformemente, U(5 minutos, 10 minutos), respectivamente.

En la Figura 2.16 se muestra un ejemplo del comportamiento del modelo. El objetivo de la simulación es estimar el tiempo medio de espera en cola y el número medio de clientes que esperan en la cola.

- *El tiempo medio de espera del cliente en la cola, se define de la forma siguiente:*

$$\hat{d}\left(n\right) = \frac{\sum_{i:1}^{n} D_i}{n} \tag{2.116}$$

donde n es el número total del clientes que han abandonado la cola y D_i es el tiempo de espera en la cola del cliente i. Para calcularlo es preciso llevar registro, a lo largo de la simulación, del número de clientes que han abandonado la cola hasta ese momento (n) y de la suma del tiempo de espera de los clientes que han abandonado la cola hasta ese momento:

$$D\left(n\right) = \sum_{i:1}^{n} D_i \tag{2.117}$$

- *El número medio de clientes que componen la cola se define de la forma siguiente:*

$$\hat{q}\left(T\right) = \frac{\int_0^T Q\left(\tau\right) \cdot d\tau}{T} \tag{2.118}$$

donde T es el tiempo que dura la simulación y $Q\left(\tau\right)$ es el número de clientes que hay en la cola en el instante τ.

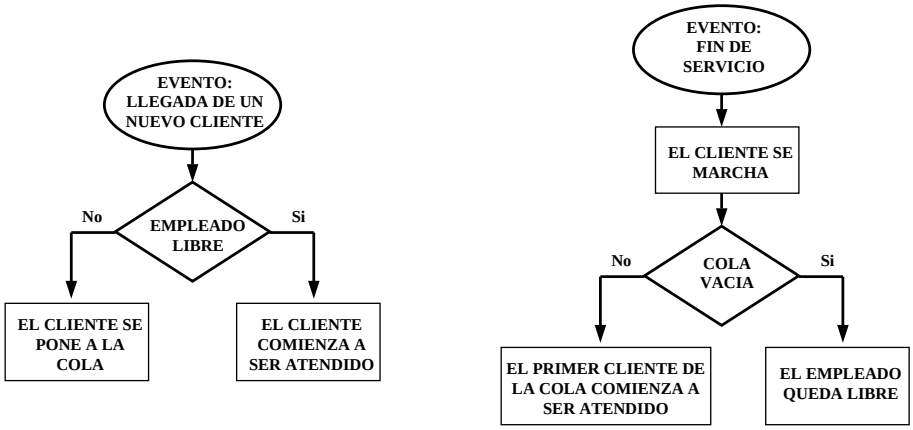

Figura 2.15: Esquema de la estructura lógica del modelo.

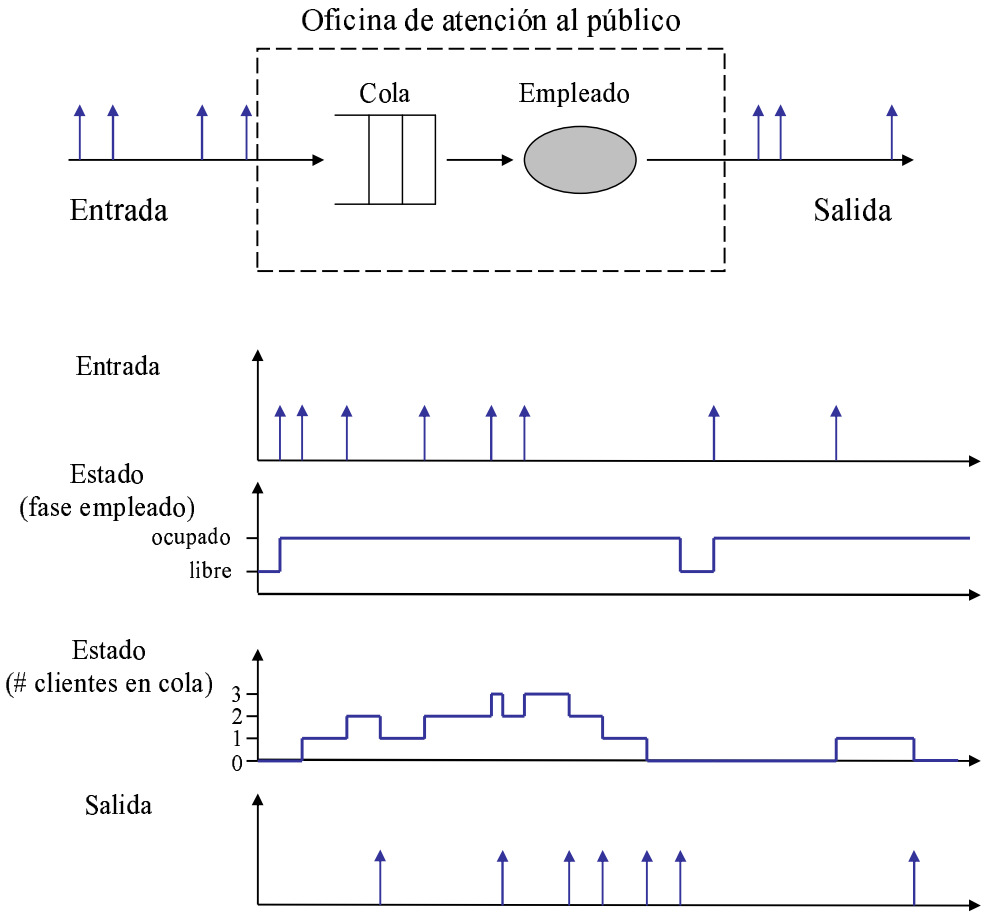

Figura 2.16: Ejemplo del comportamiento del modelo de la oficina.

Para calcularlo, es preciso llevar registro a lo largo de la simulación del valor de la integral: $\int_0^t Q(\tau) \cdot d\tau$. *Para ello se define un acumulador estadístico,* $R(t)$. *Inicialmente* R *vale cero. Cada vez que cambia el número de clientes de la cola, debe actualizarse el valor de* R*: se suma al valor actual de* R *el producto del número de clientes de la cola (antes del cambio) por el tiempo transcurrido desde el anterior cambio. De este modo va calculándose el área bajo la curva* $Q(t)$. *Para ello, es preciso definir otro acumulador estadístico,* t_{evento}*, que almacene el instante en que se produjo el anterior evento.*

Obsérvese que hay dos tipos de eventos que pueden modificar el tamaño de la cola: la llegada de un cliente y el final del servicio a un cliente (el primero de la cola comienza entonces a ser atendido). Si la condición de finalización no coincide con uno de estos dos tipos de eventos, al término de la simulación debe también actualizarse R.

En este modelo hay cuatro tipos de eventos: inicio de la simulación, llegada a la oficina de un nuevo cliente, fin de servicio a un cliente y final de la simulación. Los flujos de acciones asociados a los tres primeros tipos se muestran en la Figura 2.17. □

2.4.3. Elementos del modelo

Aparte de los eventos, un modelo de eventos discretos tiene otros tipos de componentes. En general, cabe distinguir siete tipos de componentes en un modelo de eventos discretos: los componentes del sistema (*entidades*, *recursos* y *colas*), las variables que los describen (*atributos*, *variables* y *contadores estadísticos*) y la interacción entre ellos (*eventos*). A continuación, se describe cada uno de ellos.

1. **Entidades.** Las *entidades* son objetos dinámicos en la simulación, que son creados y se mueven por el sistema, cambiando el valor de sus atributos, afectados por otras entidades y por el estado del sistema. Las entidades pueden abandonar el sistema o bien permanecer indefinidamente circulando en él.

   En el ejemplo anterior de la oficina, existe un único tipo de entidad: "cliente". Cada cliente que se encuentra en la oficina es una *realización* (o *instanciación*, como se prefiera) de este tipo de entidad, que es creada a su llegada, circula a través del sistema y es destruida cuando el cliente abandona la oficina. Si en el modelo se hubieran definido diferentes tipos de clientes, que requirieran

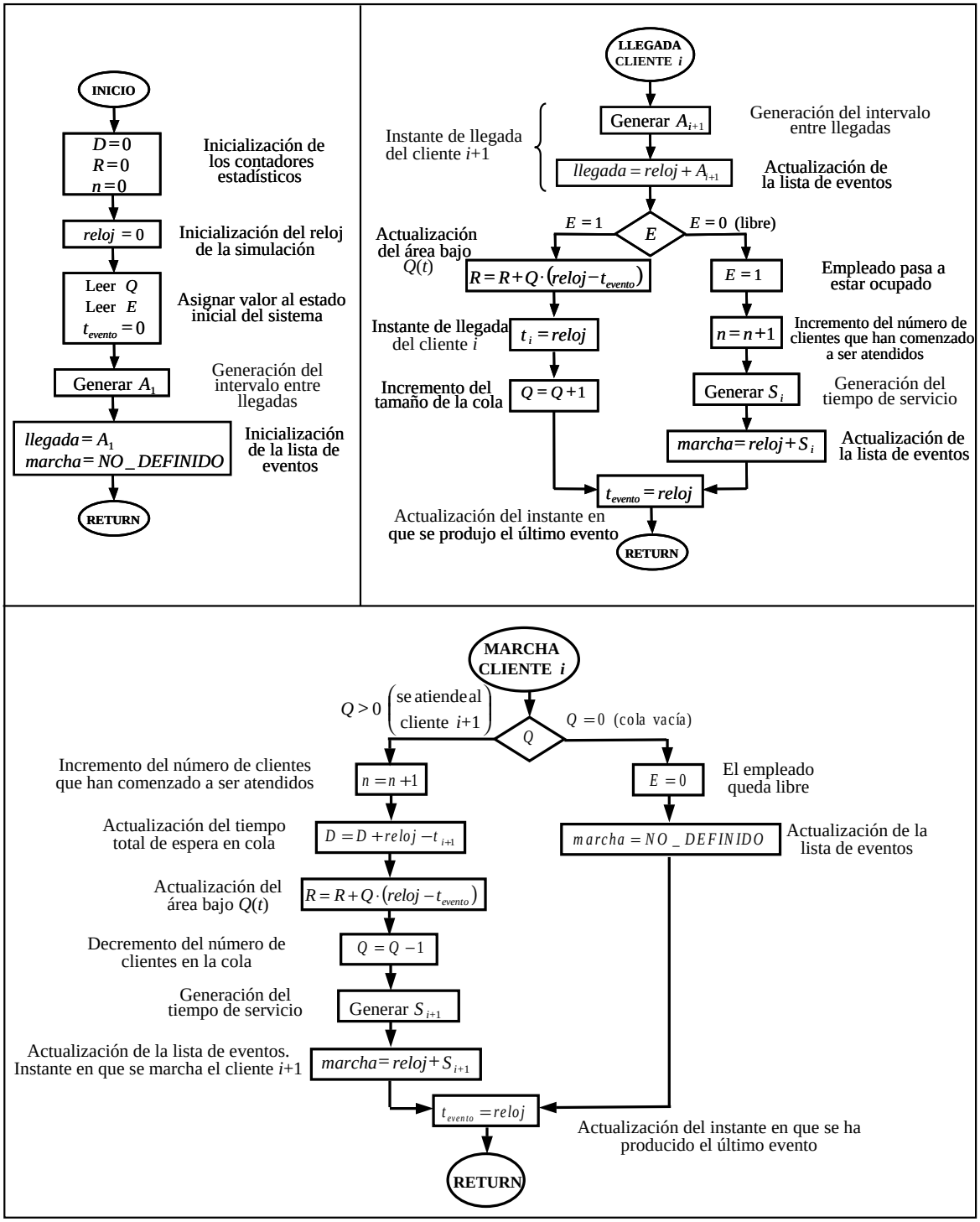

Figura 2.17: Flujos de acciones asociadas a los eventos.

procesos de atención diferentes o a los que se asignara diferente prioridad, cada tipo de cliente podría representarse como un tipo diferente de entidad.

2. **Atributos.** Los *atributos* permiten individualizar cada instanciación de una determinada clase de entidad. Al definir el tipo de entidad, se declaran sus atributos. Se pueden asignar valores diferentes a los atributos de cada instanciación de la clase de entidad, lo cual permite especificar las características particulares de cada uno de ellos.

   Por ejemplo, algunos atributos que podrían definirse para el tipo de entidad "cliente" son: la prioridad con que debe ser atendido o determinados datos personales, como son el nombre y los apellidos, la edad, la nacionalidad, etc. En general, el valor de los atributos diferirá de un cliente a otro y es lo que permite diferenciarlos.

3. **Variables.** Las variables representan características del sistema que son independientes de los tipos de entidades o del número de realizaciones existentes en determinado instante. Por tanto, las variables no están asociadas a entidades en concreto, sino que pertenecen al conjunto del sistema. Son accesibles desde todas las entidades y pueden ser modificadas por todas las entidades. Puede considerarse que cada variable es como una pizarra colgada en la pared, en la que se escribe el valor de la variable. Todas las entidades pueden leer la pizarra, y también pueden borrar el valor escrito y escribir uno nuevo.

   Algunas de las variables son intrínsecas a los elementos del modelo y, por ello surgen en casi todos los modelos de simulación. Algunas de estas son: el número de entidades (en nuestro caso, clientes) que hay en cada instante en cada cola, el número de recursos (en nuestro caso, empleados) ocupados, el estado (ocupado o libre) de cada recurso, el valor del reloj de la simulación (variable que va registrando el tiempo simulado) , etc.

   Por el contrario, otras variables surgen debido a necesidades concretas del modelo en cuestión. Por ejemplo, supóngase que cada cliente tiene que ser atendido consecutivamente por dos empleados diferentes, situados a cierta distancia, y que el tiempo que emplea la entidad en ir de un empleado a otro se considera fijo. Entonces, el valor de este tiempo de "tránsito" sería una variable del modelo.

   Las variables permiten *parametrizar* el modelo, es decir, definir magnitudes que pueden cambiar para adaptar el modelo a sus diferentes aplicaciones. Por ejemplo, mediante la modificación del tiempo de tránsito entre los dos puntos de atención, puede adaptarse un mismo modelo para representar diferentes sistemas.

Asimismo, las variables pueden representar magnitudes cuyo valor cambie durante el curso de la simulación. Por ejemplo: el número total de clientes que esperan, que están siendo atendidos y que se encuentran en tránsito entre los dos puntos de atención.

4. **Recursos.** Los *recursos* pueden ser el personal (en nuestro caso, el empleado), las máquinas (por ejemplo, si las entidades son piezas que deben ser procesadas), el espacio (por ejemplo, en un almacén), etc. Una entidad *captura* un recurso cuando éste está disponible, a fin de obtener un servicio de él, y lo *libera* una vez ha terminado.

   El recurso puede ser individual o estar compuesto por un grupo de elementos individuales, cada uno de los cuales se llama una *unidad del recurso* . Por ejemplo, el caso de varios mostradores paralelos e idénticos de atención al público, puede representarse como un recurso con tantas *unidades* como puntos de atención. El número de unidades disponibles de un recurso puede variar durante el curso de la simulación, representando mostradores que son cerrados o abiertos.

5. **Colas.** Cuando una entidad no puede circular, debido tal vez a que necesita usar una unidad de un recurso que en ese momento no se encuentra disponible, entonces la entidad necesita un sitio donde esperar: este es el propósito de la *cola*.

6. **Acumuladores estadísticos.** A fin de calcular el valor de las variables de salida, es preciso calcular durante el curso de la simulación el valor de determinadas variables intermedias. Estas se llaman *acumuladores estadísticos*. Algunos ejemplos son: el número total de clientes atendidos hasta ese momento, la suma de los tiempos de espera en cola de los clientes hasta ese momento, el número total de clientes que han comenzado a ser atendidos hasta ese momento, el mayor tiempo de espera en cola hasta ese momento, etc. Los contadores son inicializados a cero al comenzar la simulación. Cuando "algo sucede" en la simulación (es decir, se ejecuta un evento), los contadores estadísticos afectados deben ser convenientemente actualizados.

7. **Eventos.** Un *evento* es un suceso que ocurre en un determinado instante de tiempo simulado y, como resultado de la ejecución de sus acciones asociadas, puede cambiar el valor de los atributos, las variables y los acumuladores estadísticos. De hecho, los atributos, las variables y los acumuladores estadísticos sólo pueden cambiar a consecuencia de la ejecución de los eventos. Los cambios en sus valores se producen en los instantes en que son activados

los eventos, manteniéndose constantes durante el intervalo de tiempo entre eventos sucesivos.

Cada evento tiene asociado dos tipos de información. Por una parte, su *condición de activación*, también llamada *condición de disparo*. Es decir, la condición que hace que el evento pase de estar desactivado a estar activado. Por otra, las *acciones* que deben realizarse en el instante en que el evento es activado.

2.4.4. Descripción del funcionamiento del sistema

Llegado este punto, se han descrito los componentes del sistema (entidades, recursos y colas), las variables que los describen (atributos, variables y contadores estadísticos) y la interacción entre ellos (eventos).

No obstante, todavía falta describir en el modelo los detalles acerca del funcionamiento del sistema. Básicamente, puede realizarse desde dos ópticas distintas: la *orientación a los eventos* y la *orientación a los procesos*. A continuación, se describe las características fundamentales de cada una de ellas, usando para ello el modelo de la oficina de atención al cliente.

Modelado orientado a los eventos

Como su nombre indica, el *modelado orientado a los eventos* se centra en la descripción de los eventos. Es decir, en describir:

- Qué tipos de eventos se producen.
- Qué condición de activación tiene cada uno.
- Cuál es el flujo lógico de acciones asociadas a la activación de cada evento.

Anteriormente se describió el modelado orientado a los eventos de la oficina de atención al cliente. En la Figura 2.17 se mostraron los flujos de acciones correspondientes a los eventos *inicialización*, *llegada de un nuevo cliente* y *marcha de un cliente*.

El modelado orientado a los eventos presenta dos ventajas fundamentales. La primera es que permite una flexibilidad total en la descripción del modelo. La segunda es que la realización, empleando un lenguaje de programación, del código

de la simulación a partir de este tipo de descripción del modelo es conceptualmente sencilla.

Sin embargo, la orientación a los eventos presenta también una desventaja importante: la realización de modelos de grandes dimensiones, con diferentes tipos de eventos, entidades y recursos, resulta excesivamente compleja, ya que este enfoque requiere que el programador adquiera el papel de "supervisor omnisapiente", llevando el control de todos los eventos, entidades, atributos, variables y acumuladores estadísticos.

Modelado orientado a los procesos

Una forma alternativa, más natural y sencilla, de describir el modelo consiste en tomar el punto de vista de las entidades y describir su circulación a través del sistema. Este enfoque se centra en los procesos que llevan a cabo las entidades, por ello se llama *modelado orientado a los procesos*. Su práctica es posible gracias al empleo de lenguajes de simulación, que traducen de manera automática la descripción orientada a los procesos a una descripción orientada a los eventos, y ésta en código escrito en algún lenguaje de programación. En última instancia, el código ejecutable de la simulación siempre está orientado a los eventos.

El modelo orientado a los procesos de la oficina de atención al público se realiza tomando el punto de vista de un cliente cualquiera. Como en el caso anterior, las variables de salida son el tiempo medio de espera en la cola y el número medio de clientes que componen la cola. Los pasos en el proceso de atención son:

1. Llego a la oficina.
2. Escribo en mi atributo "Instante de llegada" el valor que tiene en este momento el reloj de la simulación. Así más tarde podré calcular el tiempo que he estado esperando en la cola.
3. Me pongo al final de la cola e incremento en uno el valor de la variable "Número de clientes de la cola".
4. Espero hasta que yo sea el primero de la cola y el empleado esté libre (si tengo suerte, el tiempo de espera será cero).
5. En el instante en que abandono la cola, calculo mi tiempo de espera (restando el valor de mi atributo "Instante de llegada" del valor del reloj de la simulación), decremento en uno el valor de la variable "Número de clientes de la cola",

incremento en uno la variable “Número de clientes que abandonan la cola” y comienzo a ser atendido por el empleado.

6. El empleado me atiende durante el tiempo que requiero.

7. Finaliza mi tiempo de servicio, con lo que dejo libre al empleado y abandono la oficina.

Existen varios entornos de simulación que permiten realizar una descripción orientada al proceso de los modelos. Uno de ellos es Arena (Kelton et al. 2002, Pedgen et al. 1995). Con el fin de ilustrar cómo se realiza la descripción del modelo en este tipo de entornos, en la Figura 2.18 se muestra el modelo en Arena de la oficina atendida por un empleado.

En la Figura 2.18a se muestra el modelo en un instante de su simulación. Está compuesto por tres módulos: el módulo de creación de las entidades, el módulo que describe el proceso y el módulo por el cual las entidades abandonan el sistema. La conexión entre los módulos define el flujo de las entidades por el sistema. Los modelos pueden construirse de manera modular y jerárquica, empleando los diferentes bloques para la descripción del flujo de entidades que proporciona Arena.

Haciendo doble clic en cada uno de los módulos se abre un menú de configuración. En las Figuras 2.18b y 2.18c se muestran los menús de configuración de los módulos de creación de entidades y de proceso.

Arena permite definir el experimento (condición de finalización de cada réplica, número de réplicas, etc.), calcula por defecto determinados estadísticos (tiempo de espera en las colas, tiempo de proceso, ocupación de los recursos, etc.), permite al usuario definir su propios cálculos, y genera automáticamente informes en los que se muestra de manera textual y gráfica esta información.

El entorno de simulación de Arena se distribuye como parte de un conjunto de herramientas que cubren el ciclo típico de un estudio mediante simulación. Una herramienta permite ajustar distribuciones de probabilidad a los datos experimentales, proporcionando información sobre la bondad del ajuste. Otra herramienta permite definir juegos de experimentos (factores experimentales, niveles de cada factor, número de réplicas en cada punto del diseño experimental, etc.) sobre un modelo definido en Arena, ejecuta el conjunto de simulaciones que componen el experimento y muestra el análisis estadístico de los resultados.

A la vista de las capacidades que ofrecen herramientas comerciales del tipo de Arena, cabe plantearse cuál es la ventaja de emplear el formalismo DEVS. La principal ventaja del formalismo DEVS es su total versatilidad, característica que

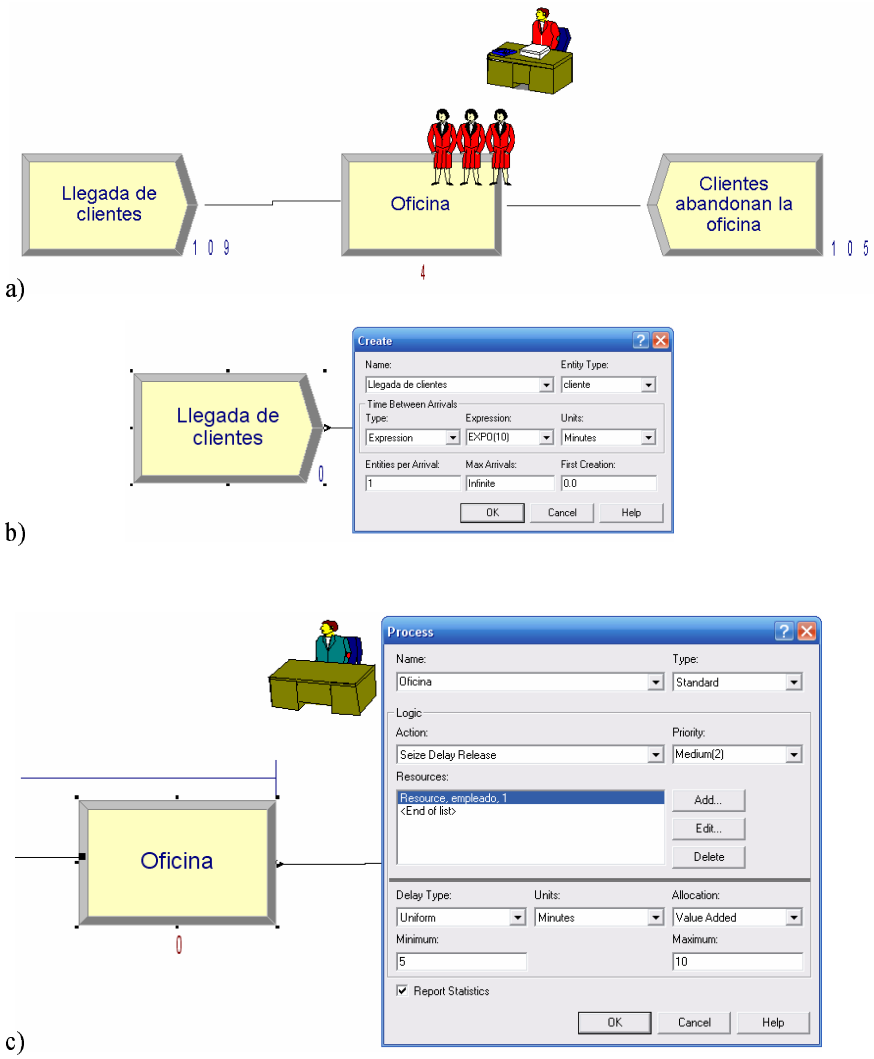

Figura 2.18: Modelo en Arena de la oficina atendida por un empleado.

no comparten los módulos predefinidos, los cuales permiten describir únicamente los comportamientos específicos para los cuáles han sido ideados.

Cuando una herramienta de simulación basadas en módulos predefinidos pretende soportar la descripción de un amplio abanico de comportamientos, el número de módulos que debe proporcionar la herramienta se hace muy grande, así como el número de variables y parámetros asociados, con lo cual su documentación se hace muy extensa (la relación de variables y atributos de Arena es un documento de aproximadamente 70 páginas).

En opinión del autor de este texto, las herramientas basadas en módulos predefinidos son de gran ayuda para la descripción de modelos “convencionales”, pero

su empleo es complejo y tedioso cuando se pretenden modelar comportamientos fuera de los habituales, cosa que frecuentemente sucede cuando se describen modelos complejos. Es en estas situaciones en las cuales el esfuerzo invertido en el aprendizaje del formalismo DEVS se ve recompensado.

2.5. EJERCICIOS DE AUTOCOMPROBACIÓN

Ejercicio 2.1

Ejecute manualmente el algoritmo que realiza la simulación del modelo representado por la *tabla de transición/salidas* mostrada a continuación, para varias trayectorias de entrada y estado inicial. Explique por qué este sistema se denomina *contador binario*.

Estado actual	Entrada actual	Estado siguiente	Salida actual
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Ejercicio 2.2

Escriba un simulador para un autómata celular unidimensional, en el cual pueda configurarse el número de células, el estado inicial y la función de transición de estados. Codifíquelo empleando un lenguaje de programación de su elección y ejecútelo para varios estados iniciales y varias funciones de transición de estado.

Ejercicio 2.3

Reescriba el simulador que planteó al resolver el Ejercicio 2.2. En este caso, en lugar de inspeccionar todas las células en cada paso de tiempo, emplee simulación basada en eventos, es decir, inspeccione en cada paso de tiempo sólo aquellas células cuyo estado tiene posibilidad de cambiar. Compare ambas alternativas en términos de tiempo de ejecución.

Ejercicio 2.4

Lea los documentos *lectura2a.pdf* y *lectura2b.pdf*. A continuación, reproduzca el comportamiento de alguno de los autómatas mostrados en estas lecturas.

Ejercicio 2.5

Plantee el algoritmo de la simulación del modelo que se describe a continuación de un depósito, empleando para ello el método numérico de integración de Euler explícito.

Llamando x al flujo (m^3/s) de entrada o salida de líquido del depósito, e y al volumen (m^3) de líquido almacenado en el depósito, y teniendo en cuenta que la variación respecto al tiempo del volumen de líquido acumulado dentro del depósito es igual al flujo, podemos escribir el modelo del depósito de la forma siguiente:

$$\dot{y} = x \tag{2.119}$$

donde $\dot{y}$ representa la derivada de y respecto al tiempo, es decir, $\frac{dy}{dt}$. En la Figura 2.19 se muestra la analogía entre el modelo del depósito y un integrador.

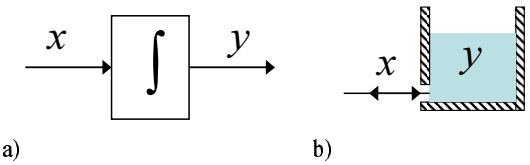

Figura 2.19: Integrador: a) representación como bloque; y b) analogía con un depósito: x es el flujo de entrada o salida (m^3/s), y es el volumen almacenado de líquido (m^3). La relación constitutiva del depósito es $\dot{y} = x$.

Ejercicio 2.6

Plantee un modelo matemático de tiempo continuo de un sistema de su elección. Clasifique las variables del modelo en parámetros, variables de estado y variables algebraicas. Asigne la causalidad computacional del modelo. Plantee el diagrama de flujo del algoritmo para su simulación.

Ejercicio 2.7

En la Figura 2.20 se muestra el esquema de una casa que dispone de un sistema de calefacción, que introduce un flujo de energía Φ_{in} (W). Asimismo, hay un flujo

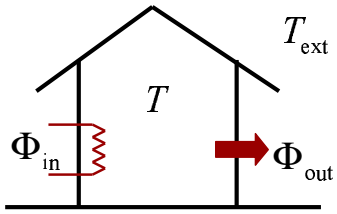

Figura 2.20: Esquema de una casa, con sistema de calefacción y disipación de calor al exterior.

de energía que sale de la casa, Φ_{out} (W), debido a la diferencia entre la temperatura en el interior de la casa T (K) y la temperatura en el exterior T_{ext} (K).

La evolución de la temperatura en el interior de la casa puede calcularse aplicando tres leyes básicas.

1. *Balance de energía.* La diferencia entre los flujos de energía Φ_{in} y Φ_{out} incrementa la energía almacenada en la casa, Q (J).

$$\frac{dQ}{dt} = \Phi_{\text{in}} - \Phi_{\text{out}} \tag{2.120}$$

2. La relación entre Q y la temperatura T depende de la capacidad calorífica de la casa C (J/K).

$$Q = C \cdot T \tag{2.121}$$

3. El flujo de energía que sale al exterior a través de las paredes y ventanas depende de la diferencia entre la temperatura interior de la casa y la temperatura del exterior.

$$\Phi_{\text{out}} = k \cdot (T - T_{\text{ext}}) \tag{2.122}$$

   donde la constante de proporcionalidad, k (W/K), es la conductividad térmica de las paredes y ventanas.

Suponiendo que la temperatura exterior, T_{ext} (K), y el flujo de energía proporcionado por la calefacción, Φ_{in} (W), son funciones conocidas del tiempo, y también que C (J/K) y k (W/K) son parámetros conocidos, realice la asignación de causalidad computacional de las ecuaciones del modelo.

Asigne un valor arbitrario a la evolución temporal de T_{ext} y Φ_{in}, y a los parámetros C y k. A continuación, escriba el algoritmo de la simulación de este modelo. El objetivo de la simulación es calcular la evolución temporal del flujo de energía de salida (Φ_{out}), de la energía acumulada en la casa (Q) y la temperatura en el interior de la casa (T).

Finalmente, codifique el algoritmo anterior empleando un lenguaje de programación de su elección y ejecútelo. Ponga como condición de finalización que el tiempo simulado alcance 24 horas. Represente las dos gráficas siguientes. En la primera gráfica, represente Φ_{in}, Φ_{out} y Q frente al tiempo. En la segunda gráfica, represente T_{ext} y T frente al tiempo.

2.6. SOLUCIONES A LOS EJERCICIOS

Solución al Ejercicio 2.1

Consideremos la trayectoria de entrada siguiente:

x(0)=0	x(1)=1	x(2)=0	x(3)=1	x(4)=1	x(5)=0	x(6)=0	x(7)=1	x(8)=0	x(9)=1

Para esta trayectoria, con un estado inicial q(0)=0, se obtiene las trayectorias de los estados y de salida siguientes:

q(0)=0	q(1)=0	q(2)=1	q(3)=1	q(4)=0	q(5)=1	q(6)=1	q(7)=1	q(8)=0	q(9)=0
x(0)=0	x(1)=1	x(2)=0	x(3)=1	x(4)=1	x(5)=0	x(6)=0	x(7)=1	x(8)=0	x(9)=1
q(1)=0	q(2)=1	q(3)=1	q(4)=0	q(5)=1	q(6)=1	q(7)=1	q(8)=0	q(9)=0	q(10)=1
y(0)=0	y(1)=0	y(2)=0	y(3)=1	y(4)=0	y(5)=0	y(6)=0	y(7)=1	y(8)=0	y(9)=0

Para la misma trayectoria de entrada, pero con estado inicial q(0)=1, se obtiene:

q(0)=1	q(1)=1	q(2)=0	q(3)=0	q(4)=1	q(5)=0	q(6)=0	q(7)=0	q(8)=1	q(9)=1
x(0)=0	x(1)=1	x(2)=0	x(3)=1	x(4)=1	x(5)=0	x(6)=0	x(7)=1	x(8)=0	x(9)=1
q(1)=1	q(2)=0	q(3)=0	q(4)=1	q(5)=0	q(6)=0	q(7)=0	q(8)=1	q(9)=1	q(10)=0
y(0)=0	y(1)=1	y(2)=0	y(3)=0	y(4)=1	y(5)=0	y(6)=0	y(7)=0	y(8)=0	y(9)=1

El estado del sistema almacena el bit menos significativo de la suma binaria de los valores de entrada. La salida es el acarreo.

Solución al Ejercicio 2.2

A continuación se muestra una forma de programar el autómata en Java.

```
public class AutomataCelular {

     public static void main(String[] args) {
        // --------------
        // ESTADO INICIAL
        // --------------
        int estadoActual[] = {0,0,0,1,1,1,0,0,0,1,1,1,0,0,0};
        // ------------------------------
        // TABLA DE TRANSICION DE ESTADOS
        // ------------------------------
        int tabla[] = new int[8];
        // Estado    Entrada     Entrada          Estado
        // actual     izqda       drcha          siguiente
        /*   0          0           0    */    tabla[0]=0;
```

```
        /*   0          0          1    */    tabla[1]=1;
        /*   0          1          0    */    tabla[2]=0;
        /*   0          1          1    */    tabla[3]=1;
        /*   1          0          0    */    tabla[4]=0;
        /*   1          0          1    */    tabla[5]=1;
        /*   1          1          0    */    tabla[6]=0;
        /*   1          1          1    */    tabla[7]=1;
        // ---------------
        // NUMERO REPLICAS
        // ---------------
        int numReplicas = 10;
        // ----------------
        // Genera resultado
        // ----------------
        int numCeldas = estadoActual.length;
        for (int t = 0; t < numReplicas; t++) {
            System.out.print("t="+ t + "\t");
            for(int i=0; i<numCeldas; i++)
                 System.out.print(Integer.toString(estadoActual[i]));
            System.out.println();
            int[] estadoAnterior = (int[])estadoActual.clone();
            for (int i = 1; i < numCeldas - 1; i++)
                 estadoActual[i] = tabla[4*estadoAnterior[i]+2*estadoAnterior[i-1]+estadoAnterior[i+1]];
        }
    }
}
```

Por simplicidad, la configuración del autómata se realiza en el propio programa. Otra posible opción sería que el programa solicitara la entrada de datos por teclado o bien que leyera los datos de un fichero.

La configuración del autómata consiste en especificar el estado inicial y la tabla de transición de estados. Del estado inicial se obtiene el número de células. Asimismo, debe indicarse el número de réplicas de que constará la simulación. A continuación, se muestran algunos ejemplos de ejecución del programa.

Ejemplo 1

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,0,0,1,0,0,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0           0    */    tabla[0]=0;
/*   0          0           1    */    tabla[1]=1;
/*   0          1           0    */    tabla[2]=1;
/*   0          1           1    */    tabla[3]=1;
```

```
/*   1          0          0    */    tabla[4]=0;
/*   1          0          1    */    tabla[5]=0;
/*   1          1          0    */    tabla[6]=0;
/*   1          1          1    */    tabla[7]=0;
```

Resultado de la ejecución:

```
t=0 00000100000
t=1 00001010000
t=2 00010101000
t=3 00101010100
t=4 01010101010
t=5 00101010100
t=6 01010101010
t=7 00101010100
t=8 01010101010
t=9 00101010100
```

Ejemplo 2

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,1,1,1,0,0,0,1,1,1,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0          0    */    tabla[0]=1;
/*   0          0          1    */    tabla[1]=0;
/*   0          1          0    */    tabla[2]=0;
/*   0          1          1    */    tabla[3]=1;
/*   1          0          0    */    tabla[4]=0;
/*   1          0          1    */    tabla[5]=1;
/*   1          1          0    */    tabla[6]=1;
/*   1          1          1    */    tabla[7]=0;
```

Resultado de la ejecución:

```
t=0 000111000111000
t=1 010101010101010
t=2 001010101010100
t=3 000101010101000
t=4 010010101010010
t=5 000001010100000
t=6 011100101001110
t=7 010100010001010
t=8 001001000100100
t=9 000000010000000
```

Ejemplo 3

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,1,1,1,0,0,0,1,1,1,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0           0    */    tabla[0]=0;
/*   0          0           1    */    tabla[1]=1;
/*   0          1           0    */    tabla[2]=0;
/*   0          1           1    */    tabla[3]=1;
/*   1          0           0    */    tabla[4]=0;
/*   1          0           1    */    tabla[5]=1;
/*   1          1           0    */    tabla[6]=0;
/*   1          1           1    */    tabla[7]=1;
```

Resultado de la ejecución:

```
t=0 000111000111000
t=1 001110001110000
t=2 011100011100000
t=3 011000111000000
t=4 010001110000000
t=5 000011100000000
t=6 000111000000000
t=7 001110000000000
t=8 011100000000000
t=9 011000000000000
```

Solución al Ejercicio 2.3

A continuación se muestra una posible forma de programar el autómata en Java.

```
public class AutomataCelular2 {

    public static void main(String[] args) {

        // --------------
        // ESTADO INICIAL
        // --------------
        int estadoActual[] = {0,0,0,0,0,1,0,0,0,0,0};
        // ------------------------------
        // TABLA DE TRANSICION DE ESTADOS
        // ------------------------------
        int tabla[] = new int[8];
        // Estado    Entrada     Entrada          Estado
        // actual     izqda       drcha          siguiente
```

```
        /*   0          0          0    */    tabla[0]=0;
        /*   0          0          1    */    tabla[1]=1;
        /*   0          1          0    */    tabla[2]=1;
        /*   0          1          1    */    tabla[3]=1;
        /*   1          0          0    */    tabla[4]=0;
        /*   1          0          1    */    tabla[5]=0;
        /*   1          1          0    */    tabla[6]=0;
        /*   1          1          1    */    tabla[7]=0;
        //------------------------------------------------------
        // ---------------
        // NUMERO REPLICAS
        // ---------------
        int numReplicas = 10;
        // ----------------
        // Genera resultado
        // ----------------
        int numCeldas = estadoActual.length;

        HashSet hSetSiguiente = new HashSet();
        for (int i=1; i<numCeldas - 1; i++)
            hSetSiguiente.add(new Integer(i));

        for (int t = 0; t < numReplicas; t++) {

            System.out.print("t="+ t + "\t");
            for(int i=0; i<numCeldas; i++)
                 System.out.print(Integer.toString(estadoActual[i]));
            System.out.println("\tCeldas examinadas: " + hSetSiguiente.size());

            int[] estadoAnterior = (int[])estadoActual.clone();

            HashSet hSet = hSetSiguiente;
            Iterator itr = hSet.iterator();

            hSetSiguiente = new HashSet();
            while (itr.hasNext()) {
                int i = ((Integer)itr.next()).intValue();
                estadoActual[i] = tabla[4*estadoAnterior[i]+2*estadoAnterior[i-1]+estadoAnterior[i+1]];
                if (estadoActual[i]!=estadoAnterior[i]) {
                    if (i-1>0) hSetSiguiente.add(new Integer(i-1));
                    hSetSiguiente.add(new Integer(i));
                    if (i+1<numCeldas-1) hSetSiguiente.add(new Integer(i+1));
                }
            }
        }
    }
}
```

Ejemplo 1

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,0,0,1,0,0,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
```

```
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0           0    */    tabla[0]=0;
/*   0          0           1    */    tabla[1]=1;
/*   0          1           0    */    tabla[2]=1;
/*   0          1           1    */    tabla[3]=1;
/*   1          0           0    */    tabla[4]=0;
/*   1          0           1    */    tabla[5]=0;
/*   1          1           0    */    tabla[6]=0;
/*   1          1           1    */    tabla[7]=0;
```

Resultado de la ejecución:

```
t=0 00000100000 Celdas examinadas: 9
t=1 00001010000 Celdas examinadas: 5
t=2 00010101000 Celdas examinadas: 7
t=3 00101010100 Celdas examinadas: 9
t=4 01010101010 Celdas examinadas: 9
t=5 00101010100 Celdas examinadas: 9
t=6 01010101010 Celdas examinadas: 9
t=7 00101010100 Celdas examinadas: 9
t=8 01010101010 Celdas examinadas: 9
t=9 00101010100 Celdas examinadas: 9
```

Ejemplo 2

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,1,1,1,0,0,0,1,1,1,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0           0    */    tabla[0]=0;
/*   0          0           1    */    tabla[1]=1;
/*   0          1           0    */    tabla[2]=0;
/*   0          1           1    */    tabla[3]=1;
/*   1          0           0    */    tabla[4]=0;
/*   1          0           1    */    tabla[5]=1;
/*   1          1           0    */    tabla[6]=0;
/*   1          1           1    */    tabla[7]=1;
```

Resultado de la ejecución:

```
t=0 000111000111000 Celdas examinadas: 13
t=1 001110001110000 Celdas examinadas: 12
t=2 011100011100000 Celdas examinadas: 11
```

```
t=3 011000111000000 Celdas examinadas: 9
t=4 010001110000000 Celdas examinadas: 9
t=5 000011100000000 Celdas examinadas: 8
t=6 000111000000000 Celdas examinadas: 6
t=7 001110000000000 Celdas examinadas: 6
t=8 011100000000000 Celdas examinadas: 5
t=9 011000000000000 Celdas examinadas: 3
```

Solución al Ejercicio 2.4

Reproduzcamos el comportamiento de dos de los autómatas descritos en la *lectura2b.pdf*. El primero de ellos corresponde con la Regla 250 (Repetition).

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
                      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0           0    */    tabla[0]=0;
/*   0          0           1    */    tabla[1]=1;
/*   0          1           0    */    tabla[2]=1;
/*   0          1           1    */    tabla[3]=1;
/*   1          0           0    */    tabla[4]=0;
/*   1          0           1    */    tabla[5]=1;
/*   1          1           0    */    tabla[6]=1;
/*   1          1           1    */    tabla[7]=1;
```

Resultado de la ejecución:

```
t=0     ------------------------------X------------------------------  Celdas examinadas: 59
t=1     -----------------------------X-X-----------------------------  Celdas examinadas: 5
t=2     ----------------------------X-X-X----------------------------  Celdas examinadas: 7
t=3     ---------------------------X-X-X-X---------------------------  Celdas examinadas: 9
t=4     --------------------------X-X-X-X-X--------------------------  Celdas examinadas: 11
t=5     -------------------------X-X-X-X-X-X-------------------------  Celdas examinadas: 13
t=6     ------------------------X-X-X-X-X-X-X------------------------  Celdas examinadas: 15
t=7     -----------------------X-X-X-X-X-X-X-X-----------------------  Celdas examinadas: 17
t=8     ----------------------X-X-X-X-X-X-X-X-X----------------------  Celdas examinadas: 19
t=9     ---------------------X-X-X-X-X-X-X-X-X-X---------------------  Celdas examinadas: 21
t=10    --------------------X-X-X-X-X-X-X-X-X-X-X--------------------  Celdas examinadas: 23
t=11    -------------------X-X-X-X-X-X-X-X-X-X-X-X-------------------  Celdas examinadas: 25
t=12    ------------------X-X-X-X-X-X-X-X-X-X-X-X-X------------------  Celdas examinadas: 27
t=13    -----------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-----------------  Celdas examinadas: 29
t=14    ----------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X----------------  Celdas examinadas: 31
t=15    ---------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X---------------  Celdas examinadas: 33
```

```
t=16    --------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X--------------  Celdas examinadas: 35
t=17    -------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-------------  Celdas examinadas: 37
t=18    ------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X------------  Celdas examinadas: 39
t=19    -----------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-----------  Celdas examinadas: 41
t=20    ----------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X----------  Celdas examinadas: 43
t=21    ---------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X---------  Celdas examinadas: 45
t=22    --------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X--------  Celdas examinadas: 47
t=23    -------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-------  Celdas examinadas: 49
t=24    ------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X------  Celdas examinadas: 51
t=25    -----X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-----  Celdas examinadas: 53
t=26    ----X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X----  Celdas examinadas: 55
t=27    ---X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X---  Celdas examinadas: 57
t=28    --X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X--  Celdas examinadas: 59
t=29    -X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-  Celdas examinadas: 59
```

A continuación, reproduzcamos el comportamiento del autómata que sigue la Regla 90 (Nesting).

```
// --------------
// ESTADO INICIAL
// --------------
int estadoActual[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
                      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
// ------------------------------
// TABLA DE TRANSICION DE ESTADOS
// ------------------------------
int tabla[] = new int[8];
// Estado    Entrada     Entrada          Estado
// actual     izqda       drcha          siguiente
/*   0          0           0    */    tabla[0]=0;
/*   0          0           1    */    tabla[1]=1;
/*   0          1           0    */    tabla[2]=1;
/*   0          1           1    */    tabla[3]=0;
/*   1          0           0    */    tabla[4]=0;
/*   1          0           1    */    tabla[5]=1;
/*   1          1           0    */    tabla[6]=1;
/*   1          1           1    */    tabla[7]=0;
```

Resultado de la ejecución:

```
t=0     ------------------------------X------------------------------  Celdas examinadas: 59
t=1     -----------------------------X-X-----------------------------  Celdas examinadas: 5
t=2     ----------------------------X---X----------------------------  Celdas examinadas: 7
t=3     ---------------------------X-X-X-X---------------------------  Celdas examinadas: 9
t=4     --------------------------X-------X--------------------------  Celdas examinadas: 11
t=5     -------------------------X-X-----X-X-------------------------  Celdas examinadas: 10
t=6     ------------------------X---X---X---X------------------------  Celdas examinadas: 14
t=7     -----------------------X-X-X-X-X-X-X-----------------------  Celdas examinadas: 17
t=8     ----------------------X---------------X----------------------  Celdas examinadas: 19
t=9     ---------------------X-X-------------X-X---------------------  Celdas examinadas: 10
t=10    --------------------X---X-----------X---X--------------------  Celdas examinadas: 14
t=11    -------------------X-X-X-X---------X-X-X-X-------------------  Celdas examinadas: 18
```

```
t=12    -----------------X-------X-------X-------X-----------------  Celdas examinadas: 22
t=13    ----------------X-X-----X-X-----X-X-----X-X----------------  Celdas examinadas: 20
t=14    ----------------X---X---X---X---X---X---X---X---------------  Celdas examinadas: 28
t=15    ---------------X-X-X-X-X-X-X-X-X-X-X-X-X-X-X---------------  Celdas examinadas: 33
t=16    -------------X-----------------------------X-------------  Celdas examinadas: 35
t=17    -------------X-X---------------------------X-X-------------  Celdas examinadas: 10
t=18    ------------X---X-------------------------X---X------------  Celdas examinadas: 14
t=19    -----------X-X-X-X-----------------------X-X-X-X-----------  Celdas examinadas: 18
t=20    ----------X-------X---------------------X-------X----------  Celdas examinadas: 22
t=21    ---------X-X-----X-X-------------------X-X-----X-X---------  Celdas examinadas: 20
t=22    --------X---X---X---X-----------------X---X---X---X--------  Celdas examinadas: 28
t=23    -------X-X-X-X-X-X-X-X---------------X-X-X-X-X-X-X-X-------  Celdas examinadas: 34
t=24    ------X---------------X---------------X---------------X------  Celdas examinadas: 38
t=25    -----X-X-------------X-X-------------X-X-------------X-X-----  Celdas examinadas: 20
t=26    ----X---X-----------X---X-----------X---X-----------X---X----  Celdas examinadas: 28
t=27    ---X-X-X-X---------X-X-X-X---------X-X-X-X---------X-X-X-X---  Celdas examinadas: 36
t=28    --X-------X-------X-------X-------X-------X-------X-------X--  Celdas examinadas: 44
t=29    -X-X-----X-X-----X-X-----X-X-----X-X-----X-X-----X-X-----X-X-  Celdas examinadas: 38
```

Solución al Ejercicio 2.5

La ecuación $\frac{dy}{dt} = x$ describe cómo varía el volumen de líquido almacenado dentro del depósito (y) en función del flujo de líquido (x). Para poder realizar la simulación del modelo, es necesario especificar cómo varía el flujo de líquido en función del tiempo, que de forma genérica puede representarse $x = f(t, y)$. El modelo a simular sería:

$$\frac{dy}{dt} = x \tag{2.123}$$

$$x = f(t, y) \tag{2.124}$$

La variable y aparece derivada, siendo la variable de estado del modelo. La variable x es una variable algebraica.

La función de paso del método de Euler explícito para la ecuación $\frac{dy}{dt} = x$ es:

$$y_{i+1} = y_i + x_i \cdot \Delta t \tag{2.125}$$

El algoritmo para simular este modelo podría ser el siguiente:

1. Debe conocerse el valor inicial de la variable de estado (p. e., $y_0 = 10$). También debe fijarse el tamaño del paso de integración (p. e., $\Delta t = 0.001$). Se fija el valor inicial del tiempo (p. e., $t_0 = 0$). Se asigna valor al índice: $i = 0$.

2. Cálculo del flujo (variable algebraica) en el instante t_i:

$$x_i = f(y_i, t_i) \tag{2.126}$$

3. Se comprueba si se satisface la condición de finalización. Si se satisface, termina la simulación. Si no, se continua en el paso siguiente.

4. Se calcula el valor de la variable de estado en el instante t_{i+1}:

$$y_{i+1} = y_i + x_i \cdot \Delta t \tag{2.127}$$

5. Avance de un paso en el tiempo e incremento del índice i:

$$t_{i+1} = t_i + \Delta t \tag{2.128}$$
$$i = i + 1 \tag{2.129}$$

6. Volver al paso 2.

En la Figura 2.21 se muestra el algoritmo para la simulación del modelo.

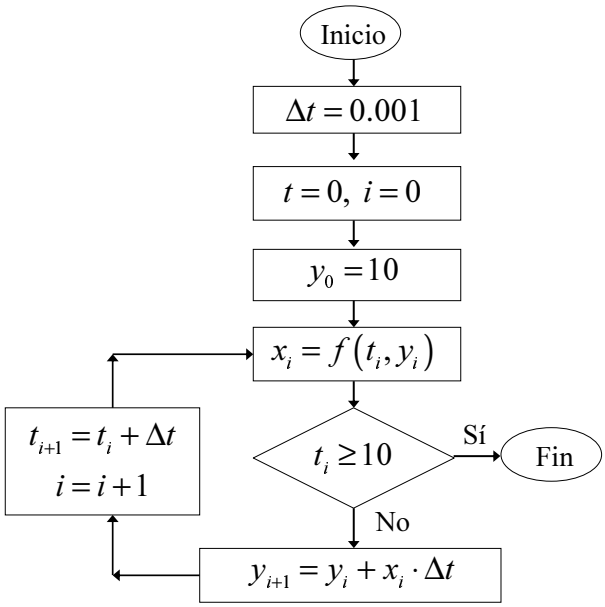

Figura 2.21: Algoritmo para la simulación del modelo del depósito.

Solución al Ejercicio 2.6

Como ejemplo podemos escoger el modelo de Lotka-Volterra, que describe la interacción entre depredadores y presas. El modelo tiene dos variables: densidad de presas (h) y densidad de depredadores (p).

$$\frac{dh}{dt} = r \cdot h - a \cdot h \cdot p \tag{2.130}$$

$$\frac{dp}{dt} = b \cdot h \cdot p - m \cdot p \tag{2.131}$$

El significado de los parámetros r, a, b y m es el siguiente:

- r Velocidad intrínseca de incremento de la población de presas
- a Coeficiente de velocidad de la depredación
- b Velocidad de reproducción de los depredadores por cada presa comida
- m Tasa de mortandad de los depredadores

El modelo posee dos variables de estado: h y p. Sustituyendo las derivadas de las variables de estado por variables mudas, se obtiene el siguiente modelo con la causalidad computacional señalada:

$$[derh] = r \cdot h - a \cdot h \cdot p \tag{2.132}$$

$$[derp] = b \cdot h \cdot p - m \cdot p \tag{2.133}$$

En la Figura 2.22 se muestra el algoritmo para la simulación del modelo. Para ello, se han asignado valores a los parámetros, al tamaño del paso de integración, se ha dado valores iniciales a las variables de estado y se ha fijado la condición de terminación.

Solución al Ejercicio 2.7

Definiendo (arbitrariamente) la forma en que T_{ext} y Φ_{in} varían con el tiempo, se obtiene que el modelo completo está compuesto por las ecuaciones siguientes:

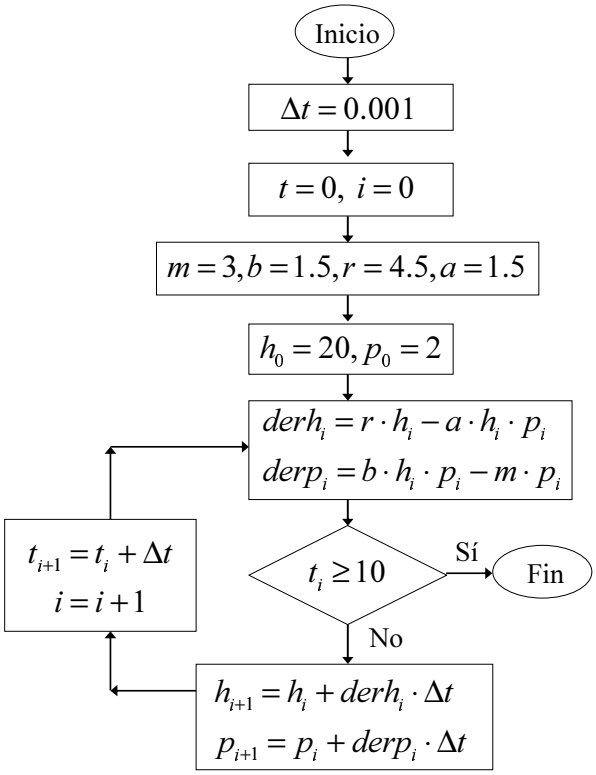

Figura 2.22: Algoritmo para la simulación del modelo de Lotka-Volterra.

$$\frac{dQ}{dt} = \Phi_{\text{in}} - \Phi_{\text{out}} \tag{2.134}$$

$$Q = C \cdot T \tag{2.135}$$

$$\Phi_{\text{out}} = k \cdot (T - T_{\text{ext}}) \tag{2.136}$$

$$T_{\text{ext}} = 275 + 10 \cdot sin(\frac{2 \cdot \pi \cdot t}{3600 \cdot 24}) \tag{2.137}$$

$$\Phi_{\text{in}} = 1e3; \tag{2.138}$$

Para simular el modelo es preciso asignar valores a los parámetros y también especificar el valor inicial de las variables de estado. Supongamos que $C = 1e5$ (J/K), $k = 55$ (W/K) y también que el valor inicial de Q es $3e7$ (J).

La variable Q aparece derivada, siendo la variable de estado. Sustituyendo la derivada de la variable de estado por la variable muda $derQ$ y realizando la asignación de la causalidad computacional, se obtiene el siguiente modelo ordenado y con la causalidad computacional señalada:

$$[\Phi_{\text{in}}] = 1e3; \tag{2.139}$$

$$[T_{\text{ext}}] = 275 + 10 \cdot sin(\frac{2 \cdot \pi \cdot t}{3600 \cdot 24}) \tag{2.140}$$

$$Q = C \cdot [T] \tag{2.141}$$

$$[\Phi_{\text{out}}] = k \cdot (T - T_{\text{ext}}) \tag{2.142}$$

$$[derQ] = \Phi_{\text{in}} - \Phi_{\text{out}} \tag{2.143}$$

En la Figura 2.23 se muestra el algoritmo para la simulación del modelo. Dado que se ha supuesto que Φ_{in} no cambia con el tiempo, se asigna valor a esa variable al inicio de la simulación y no se modifica (es decir, esa variable es tratada como un parámetro).

Finalmente, en la Figura 2.24 puede observarse la evolución de Φ_{in}, Φ_{out}, Q, T_{ext} y T frente al tiempo.

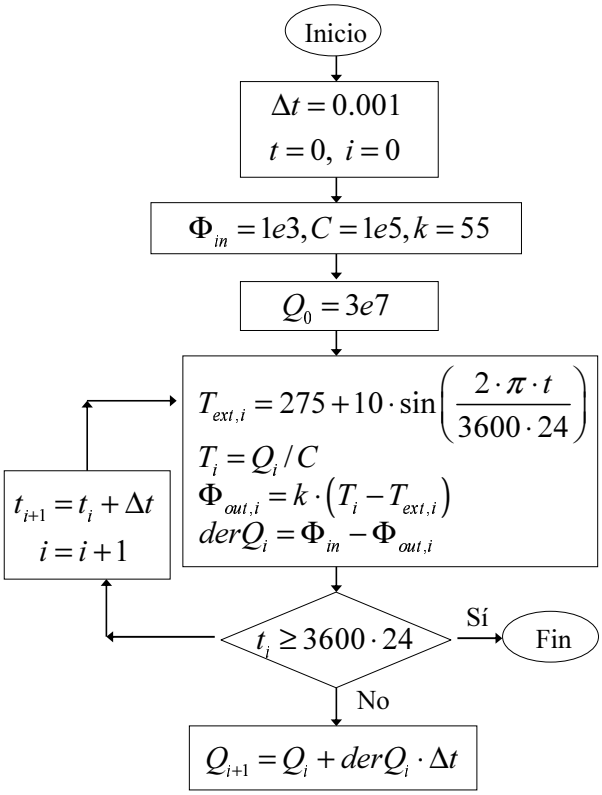

Figura 2.23: Algoritmo para la simulación del modelo de la casa.

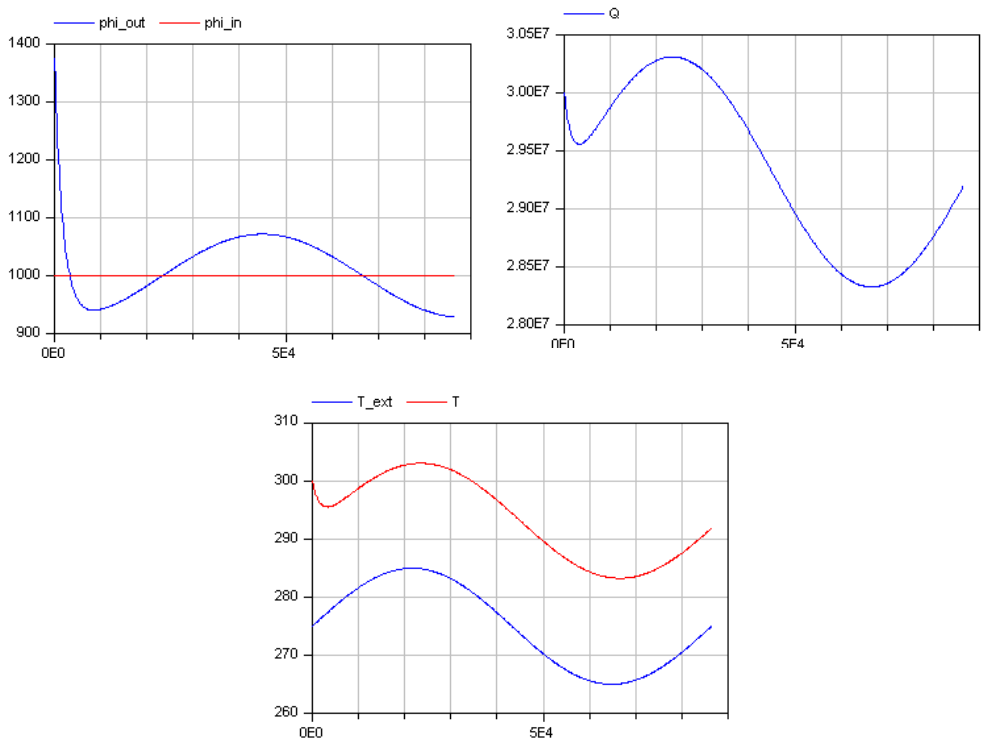

Figura 2.24: Evolución de algunas variables a lo largo de 24 horas (en las gráficas el tiempo se mide en segundos).

Tema 3

DEVS CLÁSICO

OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema debería saber:

- Discutir el formalismo DEVS clásico para modelos atómicos y compuestos.
- Describir modelos de eventos discretos mediante el formalismo DEVS clásico.
- Discutir los fundamentos del simulador abstracto para DEVS y programar, usando un lenguaje de programación, el simulador de modelos DEVS atómicos y modelos DEVS modulares y jerárquicos sencillos.
- Discutir y aplicar el formalismo DEVS multicomponente. Discutir su relación con el formalismo DEVS. Discutir el modelo multiDEVS del Juego de la Vida.
- Discutir y aplicar el procedimiento de modularización.

3.1. INTRODUCCIÓN

En este tema se describe la aplicación del formalismo DEVS (*Discrete Event System Specification*) al modelado de eventos discretos y se aplica en varios ejemplos sencillos. Este formalismo fue el propuesto, a mediados de la década de 1970, por el profesor Bernard Zeigler (Zeigler 1976).

El nombre *DEVS clásico* proviene del hecho de que dos décadas más tarde Chow y Zeigler (Chow 1996) realizaron una revisión de este formalismo, con el fin de explotar las posibilidades que ofrece la computación paralela. En el nuevo formalismo que surgió de esta revisión, denominado *DEVS paralelo*, se suprimieron algunas de las restricciones del DEVS clásico, que se impusieron debido a la forma secuencial en que operaban los ordenadores en la década de 1970.

El formalismo DEVS paralelo, que será explicado en el Tema 4, es el que se emplea comúnmente hoy en día, si bien por motivos didácticos suele explicarse en primer lugar el formalismo DEVS clásico. Esta es la aproximación seguida en este texto.

3.2. MODELOS DEVS ATÓMICOS SISO

Se dice que un modelo DEVS atómico es SISO (*single-input single-output*) cuando tiene un puerto de entrada y un puerto de salida. Un modelo de este tipo procesa una trayectoria de eventos de entrada y, de acuerdo a las condiciones iniciales del modelo y a esta trayectoria de entrada, produce una trayectoria de eventos de salida. En la Figura 3.1 se representa el comportamiento entrada/salida de este tipo de modelo DEVS.

Figura 3.1: Comportamiento E/S de un modelo DEVS SISO.

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

especial, que limita la trayectoria de salida para que sea también una secuencia de eventos.

La especificación de un *modelo DEVS atómico* con un puerto de entrada y un puerto de salida (SISO) es la tupla siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.1}$$

donde:

X	Conjunto de todos los posibles valores que un evento de entrada puede tener.
S	Conjunto de posibles estados secuenciales. La dinámica del modelo DEVS consiste en una secuencia ordenada de estados pertenecientes a S.
Y	Conjunto de posibles valores de los eventos de salida.
$\delta_{int} : S \to S$	Función de transición interna, que describe la transición de un estado al siguiente estado secuencial S. Mediante "$\delta_{int} : S \to S$" únicamente pretende indicarse que la función δ_{int} tiene un único argumento, que es un elemento del conjunto S, y la salida de la función es también un elemento del conjunto S.
$\delta_{ext} : Q \times X \to S$	Función de transición externa. La función tiene dos argumentos de entrada: un elemento del conjunto Q y un elemento del conjunto X. La función devuelve un elemento del conjunto S. El conjunto Q, llamado estado total, se define de la forma siguiente: $Q = \{\ (s, e) \mid s \in S,\ 0 \leq e \leq ta(s)\ \}$ Cada elemento del conjunto Q está formado por dos valores: (s, e), donde s es un elemento del conjunto S y e es un número real positivo: el tiempo transcurrido desde la anterior transición del estado. Como veremos, el valor de e siempre está acotado de la forma siguiente: $0 \leq e \leq ta(s)$.
$\lambda : S \to Y$	Función de salida. El argumento de entrada a la función es un elemento del conjunto S. La función devuelve un elemento del conjunto Y.

$ta : S \rightarrow \mathbb{R}^{+}_{0,\infty}$	Función de avance de tiempo (*time advance function*). El argumento de entrada a esta función es un elemento de S, que llamamos s. La salida devuelve un número real positivo, incluyendo 0 e ∞, es decir, un elemento del conjunto $\mathbb{R}^{+}_{0,\infty}$. Este número, que se denomina "avance de tiempo", es el tiempo que permanecerá el modelo en el estado s en ausencia de eventos de entrada.

La interpretación de estos elementos es la siguiente. Supongamos que en el instante de tiempo t_1 el sistema está en el estado s_1. Si no se produce ningún evento de entrada, el sistema permanecerá en el estado s_1 durante $ta(s_1)$ unidades de tiempo, es decir, hasta el instante $t_1 + ta(s_1)$. Obsérvese que $ta(s_1)$ puede ser un número real positivo, cero o infinito.

- Si $ta(s_1)$ es cero, entonces el sistema está en el estado s_1 un tiempo cero, con lo cual los eventos externos (eventos de entrada) no pueden intervenir. En este caso, se dice que s_1 es un *estado transitorio*.
- Por el contrario, si $ta(s_1)$ vale infinito, el sistema permanecerá en el estado s_1 hasta que suceda un evento externo. En este caso, se dice que s_1 es un *estado pasivo*.

Llegado el instante $t_2 = t_1 + ta(s_1)$, se produce una *transición interna*. Es decir, se realizan las siguientes acciones (en el orden indicado):

1. Se asigna a e el tiempo transcurrido desde la última transición: $e = ta(s_1)$.
2. La salida del sistema cambia al valor $y_1 = \lambda(s_1)$.
3. El estado del sistema pasa de ser s_1 a ser $s_2 = \delta_{int}(s_1)$.
4. Se planifica la siguiente transición interna para el instante $t_2 + ta(s_2)$. Para ello, se añade dicha transición a la *lista de eventos*.

Si antes del instante en que está planificada la siguiente transición interna llega un evento de entrada al sistema, entonces en el instante de llegada del evento de entrada se produce una *transición externa*.

Supongamos que el sistema pasa (mediante una transición interna o externa) al estado s_3 en el instante t_3 y que en el instante t_4 llega un evento de entrada de valor

x_1. Obsérvese que se satisfará $ta(s_3) > e = t_4 - t_3$, ya que en caso contrario se habría producido una transición interna antes de la llegada del evento externo.

En respuesta a la llegada en t_4 del evento externo, se producen las acciones siguientes:

1. El estado del sistema cambia instantáneamente debido al evento de entrada. El nuevo estado viene dado de evaluar la función de transición externa, pasándole los tres argumentos siguientes: el valor del evento de entrada, el estado y el tiempo que ha transcurrido desde la última transición (interna o externa) del estado. El nuevo estado es $s_4 = \delta_{ext}(s_3, e, x_1)$, donde $e = t_4 - t_3$.

2. Se elimina de la *lista de eventos* el evento interno que estaba planificado para el instante $t_3 + ta(s_3)$. En su lugar, se planifica un evento interno para el instante $t_4 + ta(s_4)$. En ausencia de eventos externos, ese será el instante en que se produzca la próxima transición interna.

Hay dos variables que se emplean en todos los modelos DEVS para representar intervalos de tiempo. Estas son:

– La variable e almacena el tiempo transcurrido desde la anterior transición del estado, ya sea interna o externa.

– La variable σ almacena el tiempo que resta hasta el instante en que está planificada la siguiente transición interna. Se satisface la relación: $\sigma = ta(s) - e$.

Es importante darse cuenta de que durante la transición externa no se produce ningún evento de salida. Los eventos de salida sólo se producen justo antes de las transiciones internas.

La explicación anterior acerca del funcionamiento del modelo DEVS sugiere cómo podría ser la operación de un simulador que ejecute este tipo de modelos para generar su comportamiento. A continuación, se muestra un ejemplo del funcionamiento de un modelo DEVS.

Ejemplo 3.2.1. *En la Figura 3.2 se muestra un ejemplo del funcionamiento de un sistema DEVS.*

La trayectoria de entrada consiste en dos eventos, que ocurren en los instantes t_0 *y* t_2*. Entre estos dos instantes de tiempo, en* t_1*, sucede un evento interno. Los tres eventos, producen sendos cambios en el estado del sistema. La salida* y_0 *se produce justo antes de producirse la transición interna del estado.*

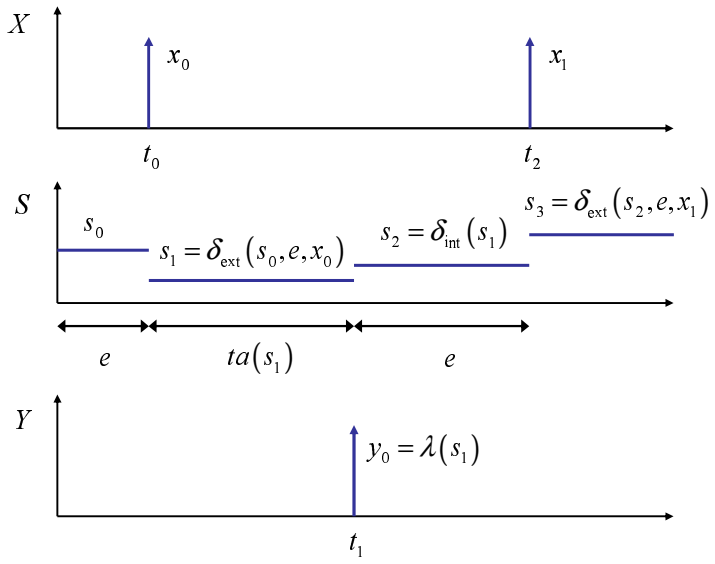

Figura 3.2: Ejemplo de funcionamiento de un modelo DEVS.

Inicialmente, el sistema se encuentra en el estado s_0. La siguiente transición interna se planifica para que ocurra transcurridas $ta(s0)$ unidades de tiempo, pero antes de que llegue ese momento se produce un evento de entrada, en el instante t_0 ($t_0 < ta(s0)$). Por ello, la transición interna planificada para el instante $ta(s0)$ no llega a producirse: se elimina de la lista de eventos.

Como resultado del evento de entrada en t_0, se produce inmediatamente un cambio en el estado. El nuevo estado viene dado por $s_1 = \delta_{ext}(s_0, e, x_0)$. Es decir, el nuevo estado se calcula de la función de transición externa, pasándole como parámetros el estado anterior (s_0), el tiempo que ha permanecido el sistema en dicho estado (e) y el valor del evento de entrada (x_0).

La siguiente transición interna se planifica para dentro de $ta(s_1)$ unidades de tiempo, es decir, para el instante $t_0 + ta(s_1)$. Como no se produce ningún evento externo (evento de entrada) antes de ese instante, en $t_0 + ta(s_1)$ se produce una transición interna. Se genera la salida, cuyo valor se calcula de la función de salida, pasándole como parámetro el valor del estado: $y_0 = \lambda(s_1)$. A continuación, se produce la transición del estado. El nuevo estado se calcula a partir del estado anterior: $s_2 = \delta_{int}(s_1)$. Obsérvese que en las transiciones internas tanto la salida como el nuevo estado se calculan únicamente a partir del valor anterior del estado.

Una vez calculado el nuevo estado, s_2, se planifica la siguiente transición interna para dentro de $ta(s_2)$ unidades de tiempo, es decir, para el instante $t_1 + ta(s_2)$. Sin

embargo, antes de que llegue ese instante se produce un evento externo en t_2, con lo cual se elimina el evento interno de la lista de eventos.

El nuevo estado en t_2 se calcula del estado anterior (s_2), del tiempo durante el cual el sistema ha permanecido en el estado s_2 (e) y del valor del evento de entrada (x_1). El nuevo estado es: $s_3 = \delta_{ext}(s_2, e, x_1)$. Se planifica la siguiente transición interna para el instante $t_2 + ta(s_3)$ y así sucesivamente. □

En la definición del modelo DEVS atómico no se indica qué ocurre si se produce un evento externo precisamente en el mismo instante en que está planificada una transición interna. Habría dos posibles formas de proceder:

1. Primero se ejecuta la transición interna y luego la transición externa.
2. Únicamente se ejecuta la transición externa, eliminándose de la lista de eventos la transición interna, que no llega a producirse.

Ambas alternativas son válidas y puede escogerse una u otra dependiendo del comportamiento del sistema que se desee modelar. Por defecto se escoge la segunda opción: DEVS ignora la transición interna y aplica la función de transición externa. Como veremos, este conflicto se resuelve en el formalismo DEVS paralelo.

A continuación, se describen varios ejemplos de modelos DEVS atómicos, con un puerto de entrada y un puerto de salida, y se muestra cuál es su comportamiento. En la Sección 3.3 se definen los modelos DEVS atómicos con varios puertos de entrada y varios puertos de salida. Finalmente, en la Sección 3.4 se definirá la especificación de los *modelos DEVS acoplados*, en la cual se proporciona información acerca de qué componentes componen el modelo y cómo se realiza el acoplamiento entre ellos.

3.2.1. Modelo de un sistema pasivo

El modelo DEVS más sencillo es aquel que no responde con salidas a ninguna trayectoria de entrada y se encuentra siempre en su único estado, que denominamos "pasivo". La definición de este modelo es la siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.2}$$

donde:

$$
\begin{array}{lcl}
X & = & \mathbb{R} \\
Y & = & \mathbb{R} \\
S & = & \{\text{“pasivo”}\} \\
\delta_{ext}\,(\text{“pasivo”}, e, x) & = & \text{“pasivo”} \\
\delta_{int}\,(\text{“pasivo”}) & = & \text{“pasivo”} \\
\lambda\,(\text{“pasivo”}) & = & \emptyset \\
ta\,(\text{“pasivo”}) & = & \infty
\end{array}
\tag{3.3}
$$

Los conjuntos de entrada y salida son numéricos: los eventos de entrada y de salida toman valores reales. Sólo hay un estado, que es “pasivo”. En este estado, el tiempo que debe transcurrir hasta que se produzca una transición interna es infinito. Eso significa que el sistema permanecerá indefinidamente en ese estado a no ser que sea interrumpido por un evento externo, en cuyo caso el estado dado por δ_{ext} también es “pasivo”. En consecuencia, el estado siempre es “pasivo”. Las funciones de transición interna y externa llevan el sistema desde el estado “pasivo” hasta el estado “pasivo”, con lo cual no modifican el estado del sistema. La función de salida devuelve el conjunto vacío, es decir, el sistema no produce ninguna salida. No obstante, puesto que *ta* vale infinito, las funciones de transición interna y de salida nunca llegan a evaluarse.

Ejemplo 3.2.2. *En la Figura 3.3 se muestra un ejemplo del comportamiento del DEVS pasivo.*

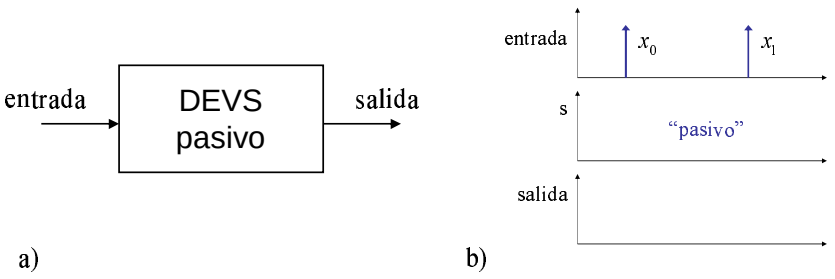

Figura 3.3: DEVS pasivo.

□

3.2.2. Modelo de un sistema acumulador

Este sistema almacena internamente el valor introducido en su entrada, cosa que hace hasta que recibe el siguiente evento de entrada. El sistema dispone de una variable de estado del tipo real, llamada *almacena*, en la cual guarda el valor.

Asimismo, necesitamos disponer de un procedimiento para indicar al sistema que queremos que genere un evento de salida con el valor almacenado. Para ello, el sistema podría disponer de una segunda entrada, de modo que un evento externo en esa entrada indique al sistema que debe producir un evento de salida con el valor que en ese instante tiene almacenado. Sin embargo, por el momento no aplicaremos esta solución, ya que estamos analizando modelos DEVS atómicos con sólo un puerto de entrada.

Otro procedimiento es establecer el convenio siguiente respecto a los eventos de entrada:

- Cuando el evento de entrada tiene valor cero, significa que queremos obtener en la salida del sistema el valor que se encuentra en ese momento almacenado.
- Cuando el evento de entrada tiene valor distinto de cero, significa que queremos que el sistema almacene internamente dicho valor.

Una peculiaridad de este sistema es que, cuando solicitamos que genere un evento de salida con el valor almacenado, no se produce inmediatamente el evento de salida, sino que se produce un retardo desde el instante de la petición (evento de entrada con valor cero) hasta que el sistema produce el evento de salida. Este *tiempo de respuesta* es un parámetro del sistema que llamaremos Δ. Mientras transcurre el *tiempo de respuesta*, es decir, mientras el sistema está procesando la petición, todos los eventos de entrada que se produzcan son ignorados.

El comportamiento del sistema tiene, por tanto, dos fases diferentes: mientras acepta eventos de entrada y mientras está procesando una petición. Definimos una segunda variable de estado, llamada *fase*, que puede tomar sólo dos valores: {“pasivo”, “responde”}.

- Mientras *fase* = “pasivo”, el sistema acepta eventos de entrada. Si el valor de la entrada es diferente de cero, lo almacena. Si es igual a cero, pasa al estado “responde”.
- Mientras el sistema está en el estado “responde”, ignora todos los eventos de entrada. Una vez han transcurrido Δ unidades de tiempo en el estado

"responde", el sistema genera un evento de salida cuyo valor es igual al de la variable *almacena*, pasando seguidamente al estado "pasivo".

Ejemplo 3.2.3. *En la Figura 3.4 se muestra un ejemplo del comportamiento dinámico del DEVS acumulador. En el instante t_0 la variable de estado fase vale "pasivo", con lo cual el sistema almacena el valor del evento de entrada, que es x_1. El valor se guarda en una variable de estado llamada almacena, la cual pasa en el instante t_0 a valer x_1.*

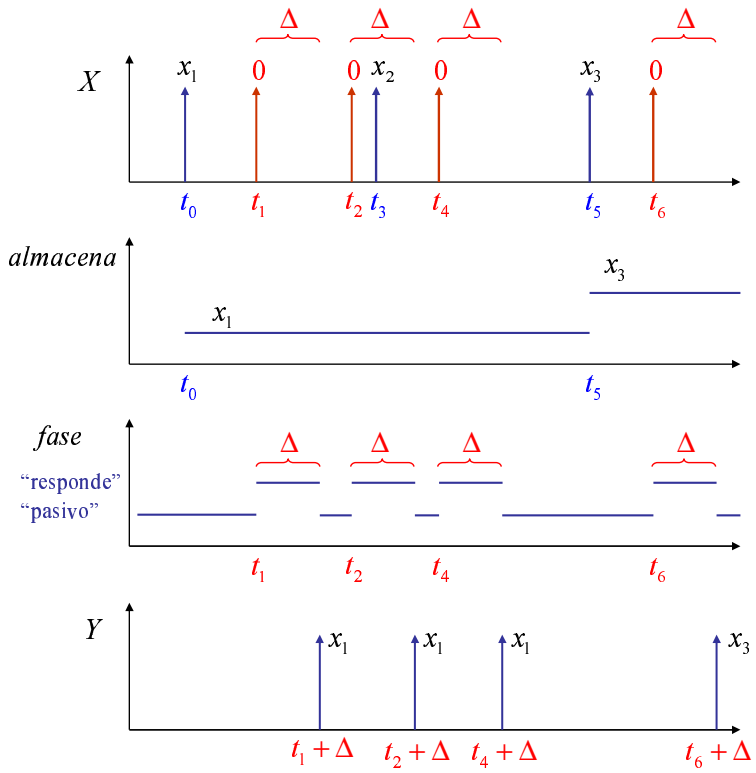

Figura 3.4: Comportamiento del DEVS acumulador.

En el instante t_1 se produce un evento de entrada con valor 0. Como consecuencia, fase pasa a valer "responde" y se programa un evento de salida, de valor x_1, para el instante $t_1 + \Delta$.

Llegado el instante $t_1 + \Delta$, se produce el evento de salida con valor x_1 y fase vuelve a tomar el valor "pasivo", con lo cual el sistema vuelve a estar listo para aceptar una nueva entrada.

En el instante t_2 se produce un evento de entrada de valor cero. Como consecuencia de ello, fase pasa a valer "responde" y se programa un evento de salida, de valor

x_1 (que es el valor guardado en la variable de estado almacena), para el instante $t_2 + \Delta$.

En el instante t_3 se produce un evento de entrada de valor x_2. Sin embargo, como no ha transcurrido todavía el tiempo de respuesta, es decir, como $t_3 < t_2 + \Delta$, el evento de entrada es ignorado.

En el instante $t_2 + \Delta$ se produce el evento de salida y la variable de estado fase vuelve a valer "pasivo".

Posteriormente, en el instante t_4 se produce un evento de entrada de valor cero. Nuevamente, fase pasa a valer "responde" y se programa un evento de salida de valor x_1 para el instante $t_4 + \Delta$. Llegado ese instante, se produce el evento de salida y fase vuelve a valer "pasivo".

En el instante t_5 se produce un evento de entrada de valor $x_3 \neq 0$. Puesto que fase="pasivo", el valor x_3 se almacena inmediatamente en la variable de estado almacena.

Finalmente, en el instante t_6 se produce un evento de entrada de valor 0, con lo cual fase pasa a valer "responde" y se programa un evento de salida de valor x_3 para el instante $t_6 + \Delta$. □

El comportamiento del DEVS acumulador puede expresarse formalmente de la manera siguiente:

$$\text{DEVS}_{\Delta} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.4}$$

Se ha escrito Δ como subíndice de la palabra DEVS para indicar que el sistema tiene un parámetro llamado Δ (el tiempo de respuesta). Un parámetro es una variable cuyo valor debe especificarse antes de iniciar la simulación del modelo y cuyo valor no se modifica posteriormente. Los elementos de la tupla tienen los valores siguientes:

$$
\begin{array}{lcl}
X & = & \mathbb{R} \\
Y & = & \mathbb{R} \\
S & = & \{\text{“pasivo”,“responde”}\} \times \mathbb{R}_0^+ \times \mathbb{R} - \{0\} \\
\delta_{ext}\left(fase, \sigma, almacena, e, x\right) & = & \begin{cases} (\text{“pasivo”}, \infty, x) & \text{si } fase = \text{“pasivo” y } x \neq 0 \\ (\text{“responde”}, \Delta, almacena) & \text{si } fase = \text{“pasivo” y } x = 0 \\ (\text{“responde”}, \sigma - e, almacena) & \text{si } fase = \text{“responde”} \end{cases} \\
\delta_{int}\left(\text{“responde”}, \sigma, almacena\right) & = & (\text{“pasivo”}, \infty, almacena) \\
\lambda\left(\text{“responde”}, \sigma, almacena\right) & = & almacena \\
ta\left(fase, \sigma, almacena\right) & = & \sigma
\end{array}
\tag{3.5}
$$

Como puede verse en la definición anterior, el conjunto de los estados secuenciales del modelo, S, es el producto cartesiano (representado mediante el símbolo $\times$) de tres conjuntos:

- El conjunto de posibles valores de la variable *fase*: {“pasivo”, “responde”}.
- El conjunto de posibles valores de la variable σ, que almacena el tiempo durante el cual el sistema va a permanecer en el estado actual. La variable σ puede tomar valores en el conjunto de los números reales positivos, incluyendo el cero. Este conjunto de representa: $\mathbb{R}_0^+$
- El conjunto de posibles valores de la variable *almacena*, que son los números reales menos el cero: $\mathbb{R} - \{0\}$.

Consecuentemente, el conjunto S se define de la forma:

$$S = \{\text{“pasivo”,“responde”}\} \times \mathbb{R}_0^+ \times \mathbb{R} - \{0\} \tag{3.6}$$

por lo cual, el estado del sistema queda especificado mediante los valores que toman las tres variables: $(fase, \sigma, almacena)$.

De la definición de ta, vemos que la variable de estado σ almacena el tiempo durante el cual el sistema va a permanecer en el estado actual. Es decir, el tiempo que debe transcurrir hasta que se produzca la siguiente transición interna, suponiendo que en ese intervalo de tiempo no se produzcan eventos de entrada.

La función de transición externa, δ_{ext}, determina el nuevo estado del sistema cuando se produce un evento externo:

– Cuando el sistema está en *fase* = "pasivo" y se produce un evento de entrada con un valor $x \neq 0$, entonces *fase* continua valiendo "pasivo" y la variable de estado *almacena* pasa a valer x. No se planifica ninguna transición interna, con lo cual el valor de σ pasa a ser infinito. El nuevo estado es ("pasivo", ∞, x).

– Cuando el sistema está en *fase* = "pasivo" y se produce un evento de entrada de valor $x = 0$ en el instante t, entonces se planifica una transición interna para el instante $t + \Delta$. Así pues, se asigna a σ el valor Δ. La variable de estado *fase* pasa a valer "responde" y el valor de la variable de estado *almacena* no se modifica. El nuevo estado es ("responde", $\Delta, almacena$).

– Cuando el sistema está en *fase* = "responde" y se produce un evento de entrada, éste es ignorado. Es decir, *fase* sigue valiendo "responde", no se modifica el valor de la variable de estado *almacena* y se actualiza el valor de σ. Puesto que e es el tiempo transcurrido desde que se produjo el anterior evento, entonces se reduce el valor de σ, restándole e, para reflejar el tiempo mas pequeño durante el cual el sistema permanece en el estado actual. El nuevo estado es ("responde", $\sigma - e, almacena$).

Finalmente, la función de transición interna, δ_{int}, define el nuevo estado cuando se produce una transición interna: la variable *fase* pasa de valer "responde" a valer "pasivo", no se modifica el valor de *almacena* y, puesto que no se planifica ninguna transición interna, se asigna infinito a σ. Así pues, el nuevo estado es: ("pasivo", $\infty, almacena$).

3.2.3. Modelo de un generador de eventos

Este sistema genera una salida, de valor uno, cada cierto intervalo de tiempo constante. La duración de dicho intervalo es un parámetro del modelo, llamado *periodo*.

El conjunto de valores posibles del estado del modelo es el producto cartesiano de los valores posibles de dos variables:

– La variable de estado *fase* puede tomar dos valores: "activo" y "pasivo".

– La variable de estado σ puede tomar valores reales positivos.

En consecuencia, el conjunto de posibles valores del estado del sistema es:

$$S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}^+ \tag{3.7}$$

en consecuencia, el estado del sistema queda definido mediante el valor de las dos variables siguientes: $(fase, \sigma)$.

El comportamiento del sistema es el siguiente. El generador permanece en la *fase* “activo” durante el intervalo de tiempo *periodo*, transcurrido el cual la función de salida genera un evento de valor uno y la función de transición interna resetea *fase* al valor “activo” y asigna a σ el valor *periodo*. Estrictamente hablando, no sería necesario volver a asignar valor a estas variables, ya que este sistema no posee entradas y, por ello, los valores de esas variables no pueden ser cambiados externamente.

Ejemplo 3.2.4. *En la Figura 3.5 se muestra un ejemplo del comportamiento dinámico del DEVS generador.*

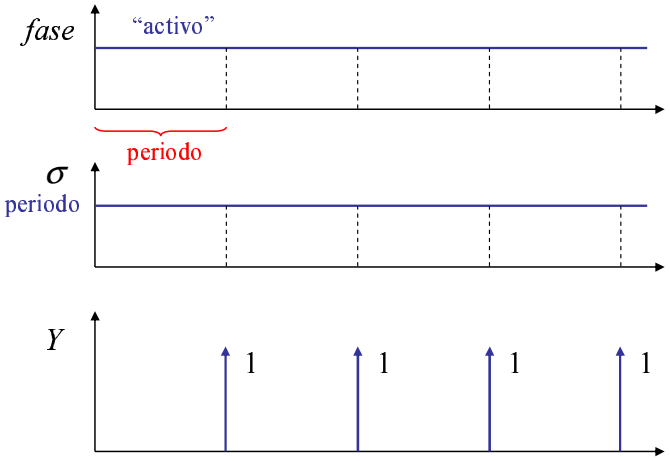

Figura 3.5: Comportamiento del DEVS generador.

□

La definición formal del comportamiento del DEVS generador es la siguiente:

$$\text{DEVS}_{periodo} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.8}$$

donde:

$$\begin{array}{lcl} X & = & \{\} \\ Y & = & \{1\} \\ S & = & \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}^{+} \\ \delta_{int}\left(fase, \sigma\right) & = & (\text{“activo”}, periodo) \\ \lambda\left(\text{“activo”}, \sigma\right) & = & 1 \\ ta\left(fase, \sigma\right) & = & \sigma \end{array} \tag{3.9}$$

3.2.4. Modelo de un contador binario

En este sistema, los eventos de entrada sólo pueden tomar el valor cero o el valor uno. Los eventos de salida sólo pueden tomar un valor: el uno. Así pues, el conjunto de posibles valores de los eventos de entrada y salida son los siguientes:

$$X = \{0, 1\} \tag{3.10}$$

$$Y = \{1\} \tag{3.11}$$

Se genera un evento de salida, de valor uno, cada dos eventos de entrada de valor uno. Para ello, el sistema contiene un contador módulo 2 del número de eventos de entrada de valor uno recibidos hasta el momento. Este contador es la variable de estado *cuenta*.

El conjunto de valores posibles del estado del sistema es el producto cartesiano de los valores posibles de las tres variables siguientes:

- La variable *fase*, que puede tomar dos valores: “pasivo” y “activo”.
- La variable σ, que puede tomar valores reales positivos, incluyendo el cero. Como siempre en los modelos DEVS, esta variable almacena el tiempo restante hasta que se produzca la próxima transición interna.
- La variable *cuenta*, que tiene dos valores posibles: 0 y 1.

Así pues, el conjunto de valores posibles del estado puede representarse de la forma siguiente:

$$S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^{+} \times \{0, 1\} \tag{3.12}$$

en consecuencia, el estado del sistema queda definido mediante el valor de las tres variables siguientes: $(fase, \sigma, cuenta)$.

El comportamiento del sistema es el siguiente. Cuando recibe una entrada de valor uno y con ella el número de unos recibidos hasta ese momento es par, entonces entra en una fase transitoria, "activo", para generar inmediatamente (se asigna el valor cero a σ) la salida, de valor uno. La salida es generada por la función de salida justo antes de que se produzca la transición interna. El comportamiento del sistema puede describirse de la manera siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.13}$$

donde:

$$\begin{array}{lcl}
X & = & \{0,1\} \\
Y & = & \{1\} \\
S & = & \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \{0,1\} \\
\delta_{ext}\left(\text{“pasivo”}, \sigma, cuenta, e, x\right) & = & \begin{cases} (\text{“pasivo”}, \sigma - e, cuenta + x) & \text{si } cuenta + x < 2 \\ (\text{“activo”}, 0, 0) & \text{en cualquier otro caso} \end{cases} \\
\delta_{int}\left(fase, \sigma, cuenta\right) & = & (\text{“pasivo”}, \infty, cuenta) \\
\lambda\left(\text{“activo”}, \sigma, cuenta\right) & = & 1 \\
ta\left(fase, \sigma, cuenta\right) & = & \sigma
\end{array} \tag{3.14}$$

3.2.5. Modelo de un proceso

En la Sección 2.4.2 se describió el modelo de una oficina de atención al cliente atendida por un empleado. Este modelo tiene un *proceso*, la oficina, el cual a su vez tiene un *recurso*: el empleado que trabaja en ella. En esta sección, veremos cómo formular el modelo de un proceso con un único recurso, empleando la metodología DEVS clásico.

Una vez hayamos desarrollado el modelo DEVS del proceso, sería posible componer el modelo de la oficina, sin la cola FIFO, simplemente conectando el modelo DEVS del generador al modelo DEVS del proceso. Las salidas del generador serían las entidades (en el caso de la oficina, los clientes). Un evento de salida del generador se interpreta como la llegada de un nuevo cliente a la oficina.

El conjunto de posibles valores del estado es el producto cartesiano de los posibles valores de las tres variables de estado siguientes:

- La *fase* del sistema puede ser "pasivo" (el empleado no está atendiendo a ningún cliente) o "activo" (está atendiendo a un cliente).
- La variable σ.
- La variable *entidad* almacena un valor real, que es un atributo característico de la entidad que tiene capturado en ese instante el recurso. Este atributo puede tomar valor en el conjunto de los números reales.

El conjunto de posibles valores del estado es:

$$S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \mathbb{R} \tag{3.15}$$

y, en consecuencia, el estado queda definido por el valor de las tres variables siguientes: $(fase, \sigma, entidad)$.

El funcionamiento del sistema es el siguiente. Mientras el recurso está en la *fase* "pasivo", el empleado no está atendiendo a ningún cliente. Si se produce un evento externo (la llegada de un cliente), entonces el recurso pasa a la *fase* "activo", es decir, el empleado comienza a atender al cliente. El tiempo que tarda en procesar la entidad (en atender al cliente) es un parámetro, denominado *tiempo de proceso*, que representamos como Δ.

Mientras el recurso está en la *fase* "activo" ignora los eventos de entrada. Esto es equivalente a suponer que si cuando llega un cliente el empleado está ocupado, entonces el cliente que llega debe marcharse sin ser atendido.

El valor de cada evento de entrada es un número real característico de la entidad. Mientras la entidad tiene capturado el recurso, este número se almacena en una variable de estado del proceso, que llamaremos *entidad*. Cuando finaliza el *tiempo de proceso* y la entidad libera el recurso, el proceso genera un evento de salida, cuyo valor es el número de la entidad, es decir, el valor almacenado en la variable de estado *entidad*.

El modelo DEVS del proceso con un recurso y sin cola puede describirse de la forma siguiente:

$$\text{DEVS}_\Delta = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.16}$$

donde:

$$
\begin{array}{lcl}
X & = & \mathbb{R} \\
Y & = & \mathbb{R} \\
S & = & \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \mathbb{R} \\
\delta_{ext}\left(fase, \sigma, entidad, e, x\right) & = & \begin{cases} (\text{“activo”}, \Delta, x) & \text{si } fase = \text{“pasivo”} \\ (\text{“activo”}, \sigma - e, entidad) & \text{si } fase = \text{“activo”} \end{cases} \\
\delta_{int}\left(fase, \sigma, entidad\right) & = & (\text{“pasivo”}, \infty, \emptyset) \\
\lambda\left(\text{“activo”}, \sigma, entidad\right) & = & entidad \\
ta\left(fase, \sigma, entidad\right) & = & \sigma
\end{array}
\tag{3.17}
$$

Ejemplo 3.2.5. *En la Figura 3.6 se muestra un ejemplo del comportamiento dinámico del DEVS que modela un proceso con un recurso.*

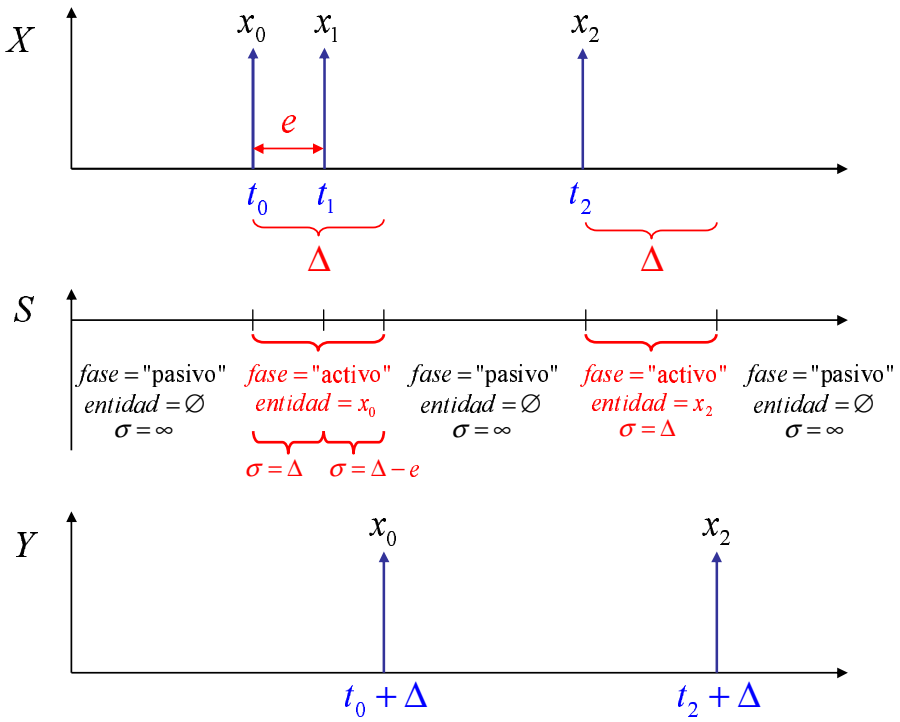

Figura 3.6: Comportamiento del DEVS de un proceso con un recurso.

En el instante t_0 se produce un evento de entrada de valor x_0. El nuevo estado se calcula de la función de transición externa, teniendo en cuenta que el estado antes del evento es : $(\text{“pasivo”}, \infty, \emptyset)$. El nuevo estado es: $(\text{“activo”}, \Delta, x_0)$. El tiempo que el sistema permanecerá en ese estado viene dado por la función ta y vale Δ.

En el instante t_1 se produce un evento externo, de valor x_1. El nuevo estado, que se calcula de la función de transición externa, es ("activo", $\Delta - e, x_0$), donde e es el tiempo transcurrido desde la última transición, es decir, $e = t_2 - t_1$. El tiempo que el sistema permanecerá en este estado viene dado por la función ta y vale $\Delta - e$.

En el instante $t_0 + \Delta$ se produce una transición interna, generándose un evento de salida cuyo valor viene dado por la función de salida, λ, y es igual al valor almacenado en la variable de estado entidad, es decir, x_0. El nuevo estado es ("pasivo", $\infty, \emptyset$). En este estado permanecerá un tiempo igual a σ, es decir, infinito.

En el instante t_2 se produce un evento externo de valor x_2. El nuevo estado, obtenido de la función de transición externa es ("activo", Δ, x_2) y el tiempo que permanecerá en él, en ausencia de eventos externos, es σ, es decir, Δ. Trascurrido el tiempo Δ, es decir, en el instante $t_2 + \Delta$, se produce una transición interna, generándose un evento de salida de valor x_2 y un cambio al estado ("pasivo", $\infty, \emptyset$). □

3.3. MODELOS DEVS ATÓMICOS MIMO

La posibilidad de definir modelos DEVS con varios puertos de entrada y de salida simplifica considerablemente la tarea de modelado. Por ejemplo, la definición del modelo DEVS del componente acumulador descrito en la Sección 3.2.2 se simplifica si se definen dos puertos: uno para introducir los valores que deben almacenarse y otro para solicitar que el sistema genere un evento de salida con el valor almacenado.

La restricción que impone el formalismo DEVS clásico es que en cada instante de tiempo sólo uno de los puertos puede recibir un evento de entrada. Es decir, no está permitido que varios puertos reciban eventos de entrada en un mismo instante de tiempo. Como veremos, esta restricción no aplica en el formalismo DEVS paralelo.

La especificación de un *modelo DEVS con varios puertos* es la tupla siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.18}$$

donde:

$X = \{ (p, v) \mid p \in InPorts,\ v \in X_p \}$	Se denomina $InPorts$ al conjunto de puertos de entrada del sistema. Para un puerto p, perteneciente a $InPorts$, se llama X_p al conjunto de posibles valores de los eventos de entrada en el puerto p. El conjunto de entrada X está compuesto por las parejas (p, v), donde p en el nombre del puerto de entrada y v es el valor del evento. Este valor pertenecerá al conjunto de valores admisibles X_p.
S	Conjunto de posibles estados secuenciales del sistema.
$Y = \{ (p, v) \mid p \in OutPorts,\ v \in Y_p \}$	Análogamente al conjunto de entrada, el conjunto de salida está compuesto por las parejas (puerto de salida, valor del evento).
$\delta_{int} : S \to S$	Función de transición interna.
$\delta_{ext} : Q \times X \to S$	Función de transición externa. $Q = \{ (s, e) \mid s \in S,\ 0 \leq e \leq ta(s) \}$ es el estado total y e es el tiempo transcurrido desde la anterior transición.
$\lambda : S \to Y$	Función de salida.
$ta : S \to \mathbb{R}^+_{0,\infty}$	Función de avance de tiempo.

Puede comprobarse que la definición del modelo DEVS con varios puertos es similar a la del modelo DEVS SISO, con la excepción de la especificación de los conjuntos de entrada (X) y salida (Y).

3.3.1. Modelo de un conmutador

En la Figura 3.7 se muestra la interfaz del modelo de un conmutador. Tiene dos puertos de entrada, $InPorts = \{in, in1\}$, y dos puertos de salida, $OutPorts = \{out, out1\}$.

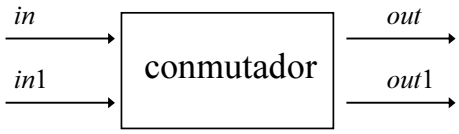

Figura 3.7: Modelo DEVS de un conmutador.

El comportamiento del conmutador está caracterizado por el valor de la variable de estado Sw, que puede ser $\{true, false\}$. Cada vez que se produce un evento de entrada, cambia el valor de la variable Sw, el cual determina el funcionamiento del conmutador:

- Mientras $Sw = true$, las entidades que llegan al puerto in son enviadas al puerto out. Similarmente, las entidades que llegan al puerto $in1$ son enviadas al puerto $out1$.
- Mientras $Sw = false$, las entidades que llegan al puerto in son enviadas al puerto $out1$ y las que llegan al puerto $in1$ son enviadas al puerto out.

En ambos casos se produce un retardo desde la llegada del evento de entrada y la generación del evento de salida. Este tiempo de proceso es un parámetro del modelo, que llamaremos Δ. Durante este tiempo de proceso, el sistema no responde a los eventos de entrada.

El estado del sistema está caracterizado por las cinco variables de estado siguientes:

- La variable *fase*, que puede valer {"pasivo", "ocupado"}.
- La variable σ, que puede tomar valores pertenecientes al conjunto $\mathbb{R}_0^+$.
- La variable *inport* almacena el puerto de entrada en el cual se produjo el evento de entrada. Tiene dos posibles valores: $\{in, in1\}$.
- La variable *almacena* guarda en valor del evento de entrada. Puede tomar valores reales.
- La variable Sw, que puede tomar valores $\{true, false\}$.

Así pues, el estado del sistema está caracterizado por las siguientes cinco variables: $(fase, \sigma, inport, almacena, Sw)$. El conjunto de posibles estados viene definido

por el producto cartesiano de los posibles valores de cada una de estas variables: $S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \{in, in1\} \times \mathbb{R} \times \{true, false\}$. La descripción del sistema es la siguiente:

$$\text{DEVS}_\Delta = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.19}$$

donde los conjuntos de entrada y salida son los siguientes:

$$\begin{array}{rcll} X & = & \{\ (p,v) \mid p \in InPorts = \{in, in1\},\ v \in X_p\ \}, & \text{con } X_{in} = X_{in1} = \mathbb{R} \\ Y & = & \{\ (p,v) \mid p \in OutPorts = \{out, out1\},\ v \in Y_p\ \}, & \text{con } Y_{out} = Y_{out1} = \mathbb{R} \end{array} \tag{3.20}$$

Las funciones de transición son:

$$\begin{array}{rcl} \delta_{ext}\,(S, e, (p,v)) & = & \begin{cases} (\text{“activo”}, \Delta, p, v, !Sw) & \text{si } fase = \text{“pasivo”} \\ (fase, \sigma - e, inport, almacena, Sw) & \text{en caso contrario} \end{cases} \\ \delta_{int}\,(S) & = & (\text{“pasivo”}, \infty, inport, almacena, Sw) \end{array} \tag{3.21}$$

La función de salida es:

$$\lambda\,(S) = \begin{cases} (out, almacena) & \text{si } fase = \text{“activo”} \ \text{y}\ Sw = true \ \text{e}\ inport = in \\ (out1, almacena) & \text{si } fase = \text{“activo”} \ \text{y}\ Sw = true \ \text{e}\ inport = in1 \\ (out1, almacena) & \text{si } fase = \text{“activo”} \ \text{y}\ Sw = false \ \text{e}\ inport = in \\ (out, almacena) & \text{si } fase = \text{“activo”} \ \text{y}\ Sw = false \ \text{e}\ inport = in1 \end{cases} \tag{3.22}$$

Finalmente, la función de avance del tiempo es:

$$ta\,(S) = \sigma \tag{3.23}$$

3.4. MODELOS DEVS ACOPLADOS MODULARMENTE

El formalismo DEVS permite definir modelos compuestos. Están formados por otros modelos DEVS (atómicos o compuestos), acoplados entre sí mediante la cone-

xión de sus puertos. La especificación de los modelos DEVS compuestos es la tupla siguiente:

$$N = \langle X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC,\ select \rangle \tag{3.24}$$

donde:

- Los conjuntos de entrada y salida, X e Y, definen la interfaz del modelo compuesto.
- D y M_d definen los componentes.
- EIC, EOC, IC definen la conexión entre los componentes y también la conexión de estos con la interfaz del modelo compuesto.
- La función *select* establece la prioridad en caso de que varios componentes tengan planificada una transición interna para el mismo instante.

De forma más rigurosa, esto mismo puede expresarse como sigue:

$X = \{\ (p, v) \mid p \in InPorts,\ v \in X_p\ \}$	Conjunto de entrada del modelo compuesto.
$Y = \{\ (p, v) \mid p \in OutPorts,\ v \in Y_p\ \}$	Conjunto de salida del modelo compuesto.
D	Conjunto de nombres de los componentes.

Los componentes son modelos DEVS. Para cada $d \in D$ se verifica:

$M_d = \langle X_d, S, Y_d, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$	Es un modelo DEVS clásico, que en general tendrá varios puertos de entrada y salida. $X_d = \{\ (p, v) \mid p \in InPorts_d,\ v \in X_p\ \}$ $Y_d = \{\ (p, v) \mid p \in OutPorts_d,\ v \in Y_p\ \}$

Se distinguen tres tipos de conexión entre puertos, que se denominan EIC, EOC e IC. Como ejemplo ilustrativo de los tres tipos de conexión, en la Figura 3.8 se muestran dos componentes, D_1 y D_2, que están conectados entre sí y conectados a la interfaz del modelo compuesto que los engloba. Cada uno de los componentes, así

como el modelo compuesto, tiene un puerto de entrada y un puerto de salida. En este ejemplo, los puertos de los componentes se representan mediante triángulos de color azul y los puertos de modelo compuesto mediante triángulos de color negro.

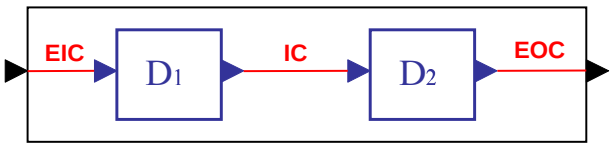

Figura 3.8: Tipos de conexión entre modelos DEVS con puertos.

1. *Entrada externa - Entrada de un componente.* Una de las entradas al modelo compuesto se conecta a una de las entradas de los componentes que lo forman. Este tipo de conexión, que se denomina EIC (*External Input Coupling*), puede definirse de la forma siguiente:

$$\text{EIC} \subseteq \{\ (\ (N, ip_N),\ (d, ip_d)\ )\ \mid ip_N \in InPorts,\ d \in D,\ ip_d \in InPorts_d\ \}$$

donde

(N, ip_N)	representa el puerto de entrada ip_N del modelo compuesto
(d, ip_d)	representa el puerto de entrada ip_d del componente d
$(\ (N, ip_N),\ (d, ip_d)\ )$	representa la conexión entre ambos puertos

2. *Salida de un componente - Salida externa.* Una de las salidas de los componentes se conecta a una de las salidas del modelo compuesto. Este tipo de conexión se llama EOC (*External Output Coupling*) y puede definirse de la forma siguiente:

$$\text{EOC} \subseteq \{\ (\ (d, op_d),\ (N, op_N)\ )\ \mid op_N \in OutPorts,\ d \in D,\ op_d \in OutPorts_d\ \}$$

donde

(d, op_d)	representa el puerto de salida op_d del componente d
(N, op_N)	representa el puerto de salida op_N del modelo compuesto
$(\ (d, op_d),\ (N, op_N)\ )$	representa la conexión entre ambos puertos

3. *Salida de un componente - Entrada de un componente.* Una de las salidas de un componente se conecta a una de las entradas de otro componente diferente. Este tipo de conexión se llama IC (*Internal Coupling*) y puede definirse de la forma siguiente:

$$\text{IC} \subseteq \{\ (\ (a, op_a),\ (b, ip_b)\ )\ \mid a, b \in D \text{ con } a \neq b,\ op_a \in OutPorts_a,\ ip_b \in InPorts_b\ \}$$

donde

(a, op_a)	representa el puerto de salida op_a del componente a
(b, ip_b)	representa el puerto de entrada ip_b del componente b
$(\ (a, op_a),\ (b, ip_b)\ )$	representa la conexión entre ambos puertos

La condición "$a, b \in D$ con $a \neq b$" indica que en el formalismo DEVS no está permitido conectar una salida de un componente con una entrada del mismo componente. Por ello, en la definición anterior de la conexión interna se impone que los componentes a y b no sean el mismo: $a \neq b$.

Finalmente, *select* es una función del tipo:

$select : 2^D \rightarrow D$	Función select, que establece la prioridad en caso de que varios componentes tengan un evento interno planificado para el mismo instante.

Cuando dos o más componentes tienen prevista una transición interna para el mismo instante de tiempo, el funcionamiento global del sistema puede variar dependiendo del orden en que se disparen estas transiciones internas, ya que las transiciones internas provocan eventos de salida, que a su vez provocan eventos externos en otros componentes.

La función *select* establece prioridades para las transiciones internas de diferentes componentes. Cuando un grupo de componentes tiene planificada para un determinado instante una transición interna, la función *select* aplicada a dicho subconjunto devuelve aquel componente del subconjunto que realizará la transición en primer lugar.

3.4.1. Modelo de una secuencia de etapas

Supongamos una secuencia de unidades funcionales (también denominadas *etapas*) que realizan una tarea en varios pasos, como por ejemplo, una línea de ensamblaje en una fábrica. En concreto, supongamos que la línea o *pipeline* consta de tres etapas, tal como se muestra en la Figura 3.9, cada una de las cuales es un proceso realizado por un recurso.

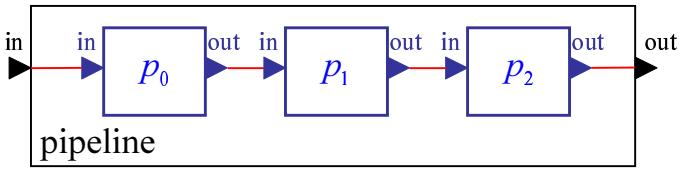

Figura 3.9: Pipeline compuesta por tres etapas.

La conexión en serie de los componentes se realiza conectando:

- El puerto de salida del primer proceso (p_0) con el puerto de entrada del segundo proceso (p_1).
- El puerto de salida del segundo proceso (p_1) con el puerto de entrada del tercer proceso (p_2).

Este tipo de conexión entre componentes al mismo nivel jerárquico se denomina *conexión interna* o *acoplamiento interno* (IC), y se realiza siempre entre un puerto de salida y un puerto de entrada.

Puesto que los modelos compuestos pueden usarse a su vez como componentes para construir otros modelos de un nivel jerárquico superior, los modelos compuestos tienen también puertos de entrada y de salida. Por ejemplo, la pipeline mostrada en la Figura 3.9 tiene dos puertos, señalados mediante triángulos de color negro, uno de entrada, que está conectado al puerto de entrada del primer componente, y otro de salida, que está conectado al puerto de salida del tercer componente. En este caso, la conexión se realiza entre un puerto de un modelo de nivel jerárquico superior (el modelo compuesto) y el puerto de uno de sus componentes, estableciéndose siempre, o bien entre dos puertos de entrada (EIC), o bien entre dos puertos de salida (EOC).

La especificación del modelo DEVS compuesto mostrado en la Figura 3.9 es la siguiente:

$$N = \langle\ X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC,\ select\ \rangle \tag{3.25}$$

donde:

$InPorts = \{\text{“in”}\}$	Puerto de entrada del modelo compuesto.
$X_{in} = \mathbb{R}$	Posibles valores de los eventos en el puerto “in” del modelo compuesto.
$X = \{\ (\text{“in”}, v) \mid v \in \mathbb{R}\ \}$	Puerto de entrada del modelo compuesto y posibles valores de los eventos en ese puerto.
$OutPorts = \{\text{“out”}\}$	Puerto de salida del modelo compuesto.
$Y_{out} = \mathbb{R}$	Posibles valores de los eventos en el puerto “out” del modelo compuesto.
$Y = \{\ (\text{“out”}, v) \mid v \in \mathbb{R}\ \}$	Puerto de salida del modelo compuesto y posibles valores de los eventos en ese puerto.
$D = \{p_0, p_1, p_2\}$	Conjunto de nombres de los componentes.
$M_{p_0} = M_{p_1} = M_{p_2} = M_{\text{recurso}}$	Cada componente es un modelo DEVS del tipo recurso, como los descritos en la Sección 3.2.5.
$\text{EIC} = \{\ ((N, \text{“in”}), (p_0, \text{“in”}))\ \}$	Conexión del puerto de entrada del modelo compuesto, N, al puerto de entrada del componente p_0.
$\text{EOC} = \{\ ((p_2, \text{“out”}), (N, \text{“out”}))\ \}$	Conexión del puerto de salida del componente p_2 al puerto de salida del modelo compuesto.
$\text{IC} = \{\ ((p_0, \text{“out”}), (p_1, \text{“in”})),\ ((p_1, \text{“out”}), (p_2, \text{“in”}))\ \}$	Conexiones internas.
$select(D') = \text{proceso con mayor índice}$	Función *select*, que dado un conjunto de componentes devuelve aquel cuyo índice es mayor.

Las conexiones transmiten instantáneamente los eventos entre los puertos. Es decir:

- Si un evento externo llega al puerto de entrada del modelo compuesto, éste es transmitido instantáneamente al puerto de entrada del componente p_0.
- Si el componente p_0 genera un evento de salida, éste es transmitido instantáneamente al puerto de entrada del componente p_1. Análogamente, si el componente p_1 genera un evento de salida, éste es transmitido instantáneamente al puerto de entrada del componente p_2.
- Si el componente p_2 genera un evento de salida, éste es transmitido instantáneamente al puerto de salida del modelo compuesto.

El siguiente ejemplo muestra la aplicación de la función *select* al establecimiento del orden de disparo en caso de que varios eventos estén programados para el mismo instante.

Ejemplo 3.4.1. *Supongamos que el intervalo de tiempo que transcurre entre la llegada de eventos sucesivos al modelo compuesto es igual al tiempo de proceso de los tres componentes, Δ. En este caso, las salidas de los componentes p_0 y p_1 están planificadas para el mismo instante, apareciendo además la salida de p_0 como entrada a p_1, con lo cual el componente p_1 tiene planificadas para el mismo instante una transición externa, debida al evento de entrada, y una transición interna, en la cual genera un evento de salida. El diseñador del modelo debe decidir qué hacer respecto al orden en que se disparan estos dos eventos:*

1. *Si se dispara primero el evento interno, el componente p_1 pasa de "activo" a "pasivo", con lo cual, cuando a continuación se dispare el evento externo, está en disposición de aceptar la entidad proveniente de p_0.*
2. *Si se dispara primero el evento externo, la entidad que llega a p_1 encuentra que éste está en la fase "activo", con lo cual el evento de entrada es ignorado. A continuación, se dispara el evento interno de p_1, pasando este componente de "activo" a "pasivo" y quedando, por tanto, en la fase "pasivo".*

La función select especifica que si hay dos eventos planificados para el mismo instante en los componentes p_0 y p_1, entonces el evento asociado al componente de mayor índice, que en este caso es p_1, se dispara en primer lugar. Así pues, se verificaría la primera de las dos opciones anteriores. □

3.4.2. Modelo de una red de conmutación

En el modelo compuesto mostrado en la Figura 3.10, un conmutador (s_0) envía entidades a dos procesos (p_0, p_1). Obsérvese que la salida de cada proceso está conectada a la salida del modelo compuesto, con lo cual los eventos generados por ambos procesos aparecen en la salida del modelo compuesto.

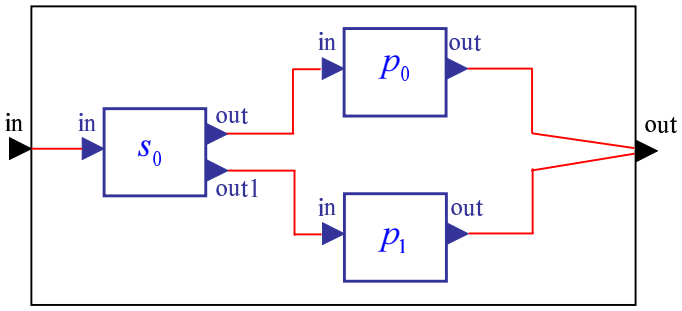

Figura 3.10: Red de conmutación.

El modelo compuesto puede especificarse siguiendo el formalismo DEVS de la forma siguiente:

$$N = \langle\ X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC,\ select\ \rangle \tag{3.26}$$

donde:

$InPorts = \{\text{“in”}\}$	Puerto de entrada del modelo compuesto.
$X_{in} = \mathbb{R}$	Posibles valores de los eventos en el puerto “in” del modelo compuesto.
$X = \{\ (\text{“in”}, v) \mid v \in \mathbb{R}\ \}$	Puerto de entrada del modelo compuesto y posibles valores de los eventos en ese puerto.
$OutPorts = \{\text{“out”}\}$	Puerto de salida del modelo compuesto.
$Y_{out} = \mathbb{R}$	Posibles valores de los eventos en el puerto “out” del modelo compuesto.
$Y = \{\ (\text{“out”}, v) \mid v \in \mathbb{R}\ \}$	Puerto de salida del modelo compuesto y posibles valores de los eventos en ese puerto.

$D = \{s_0, p_0, p_1\}$	Conjunto de nombres de los componentes.
$M_{s_0} = M_{\text{conmutador}}$ $M_{p_0} = M_{p_1} = M_{\text{recurso}}$	Tipo de los componentes.
EIC = { $((N, \text{“in”}), (s_0, \text{“in”}))$ }	Conexión externa de entrada.
EOC = { $((p_0, \text{“out”}), (N, \text{“out”}))$, $((p_1, \text{“out”}), (N, \text{“out”}))$ }	Conexiones externas de salida.
IC = { $((s_0, \text{“out”}), (p_0, \text{“in”}))$, $((s_0, \text{“out1”}), (p_1, \text{“in”}))$ }	Conexiones internas.

3.5. SIMULADOR ABSTRACTO PARA DEVS

Como vimos en la Sección 1.8, la simulación es una transformación desde un nivel superior en la descripción del sistema, a la descripción de entrada/salida. Típicamente, dada una especificación del sistema, por ejemplo, siguiendo la metodología DEVS, y dados también el valor inicial de todas las variables de estado del modelo y las trayectorias de entrada, la tarea del simulador es obtener la trayectoria del estado (evolución de las variables de estado) y las trayectorias de salida.

En esta sección va a describirse un procedimiento para la simulación de modelos DEVS, en el cual se establece de manera sistemática una relación entre los componentes atómicos y acoplados de cualquier modelo DEVS, y los algoritmos para su simulación (Zeigler et al. 2000).

3.5.1. Estructura del simulador abstracto

La estructura jerárquica del modelo DEVS es reproducida por una estructura jerárquica de simuladores abstractos que, funcionando de manera cooperativa, realiza la simulación del modelo DEVS. La estructura del simulador se obtiene a partir de la estructura del modelo de la forma siguiente:

- Se asocia, a cada *modelo atómico DEVS*, un algoritmo llamado *Devs-simulator*, que realiza la simulación del modelo DEVS atómico.
- Se asocia, a cada modelo DEVS acoplado, un algoritmo llamado *Devs-coordinator*, que realiza la simulación del modelo DEVS acoplado.

– En la parte superior de la jerarquía del simulador abstracto se sitúa un algoritmo denominado *Devs-root-coordinator*.

Ejemplo 3.5.1. *Para ilustrar la construcción del simulador abstracto, en la Figura 3.11 se muestra la correspondencia entre un modelo DEVS compuesto, con varios niveles jerárquicos, y su simulador abstracto. Los algoritmos Devs-simulator están asociados con las hojas del árbol del modelo: los modelos DEVS atómicos. Los algoritmos Devs-coordinator están asociados con los modelos DEVS acoplados, que están situados en los nodos internos de la jerarquía del modelo DEVS.* □

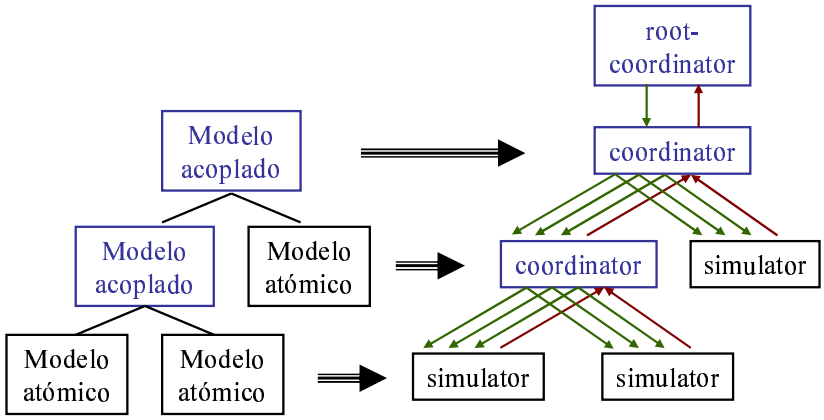

Figura 3.11: Correspondencia entre un modelo DEVS compuesto y jerárquico (izquierda), y su simulador abstracto (derecha).

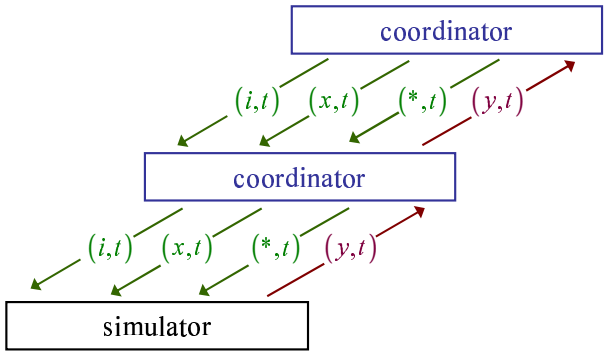

Figura 3.12: Protocolo de comunicación.

Los algoritmos *Devs-simulator* y *Devs-coordinator* son *simuladores abstractos*, por ese motivo en ocasiones se empleará el término *simulador* para referirse indistintamente a cualquiera de los dos algoritmos.

3.5.2. Intercambio de mensajes

Los algoritmos *Devs-simulator* y *Devs-coordinator* siguen un determinado protocolo de intercambio de *mensajes*, que les permite coordinarse entre sí durante la simulación y mantener el *calendario de eventos*.

En la Sección 2.4 se explicó que en la simulación de eventos discretos se mantiene una lista de eventos, denominada *calendario de eventos*, que está ordenada atendiendo a la proximidad del instante en que está planificado el disparo de cada evento. Los eventos son ejecutados extrayéndolos en orden del calendario de eventos y realizando las correspondientes transiciones en el estado, es decir, los correspondientes cambios en el valor de las variables de estado del modelo. La ejecución de los eventos resulta en la planificación de nuevos eventos, que son insertados en el calendario de eventos, así como en la cancelación de algunos eventos previamente planificados, que son eliminados del calendario de eventos. Como veremos, el calendario de eventos del simulador abstracto se construye mediante paso de mensajes a través de la estructura jerárquica del simulador.

La comunicación de los algoritmos *Devs-coordinator* entre sí y con los algoritmos *Devs-simulator* se realiza mediante cuatro tipos de mensaje (véase la Figura 3.12): (i,t), (x,t), $(*,t)$ y (y,t).

Un simulador envía a sus *hijos* (los simuladores de inferior nivel jerárquico que están directamente en comunicación con él) tres tipos de mensaje, mediante los cuales puede inicializar el hijo, planificar un evento interno en el hijo y producir un evento de entrada en el hijo. Los tres tipos de mensaje son los siguientes:

$(i,t) \swarrow$ El mensaje (i,t), llamado *mensaje-i*, inicializa el hijo, es decir, fuerza que determinadas variables internas del algoritmo hijo adquieran un valor inicial.

$(x,t) \swarrow$ El mensaje (x,t), llamado *mensaje-x*, dispara la ejecución de un evento de entrada de valor v, en el puerto de entrada p del hijo, en el instante t. Se verifica: $x = (p,v)$.

$(*,t) \swarrow$ El mensaje $(*,t)$, llamado *mensaje-**, dispara la ejecución de un evento interno en el hijo, en el instante t.

Asimismo, un simulador envía a su *padre* (el algoritmo de superior nivel jerárquico con el que se comunica) un único tipo de mensaje:

$(y, t) \nearrow$ El mensaje (y, t), llamado *mensaje-y*, tiene la finalidad de notificar al padre que se ha producido un evento de salida de valor v, por el puerto de salida p del simulador, en el instante t. Se verifica: $y = (p, v)$.

En los cuatro tipos de mensaje, el valor de t en el mensaje es igual al valor del reloj de la simulación en el instante en que se produce la comunicación del mensaje.

3.5.3. Devs-simulator

El algoritmo *Devs-simulator* es un simulador abstracto de un modelo DEVS atómico. El algoritmo emplea las tres variables siguientes, que tienen unidades de tiempo:

t	Reloj de la simulación: almacena el tiempo simulado actual.
tl	Almacena el instante en que ocurrió el último evento ("*l*" proviene de la palabra inglesa "*last*").
tn	Almacena el instante de tiempo para el cual está planificado el próximo evento ("*n*" proviene de a palabra inglesa "*next*").

De la definición de la *función de avance en el tiempo*, ta, se verifica la relación siguiente, que permite calcular tn:

$$tn = tl + ta(s) \tag{3.27}$$

Conocidos t, tl y tn, pueden calcularse las variables e y σ, las cuales intervienen en la definición del modelo DEVS:

– El tiempo transcurrido desde el último evento, e, puede calcularse de la expresión siguiente:

$$e = t - tl \tag{3.28}$$

– El tiempo que falta hasta el próximo evento, σ, puede calcularse de la expresión siguiente:

$$\sigma = tn - t = ta(s) - e \tag{3.29}$$

A continuación, se muestra el algoritmo del simulador abstracto *Devs-simulator*, que realiza la simulación de un modelo DEVS atómico:

`Devs-simulator`

```
  Variables:
    parent // Padre de este algoritmo Devs-simulator
    tl     // Instante de tiempo en que se produjo el último evento
    tn     // Instante de tiempo en que está planificado el siguiente evento
    (s,e)  // Estado total del modelo DEVS asociado a este Devs-simulator
    y      // y = (p,v), evento de valor v en el puerto de salida p del modelo
    x      // x = (p,v), evento de valor v en el puerto de entrada p del modelo
  when ( recibe mensaje-i (i,t) en el instante t ) {
    tl = t - e
    tn = tl + ta(s)
    envía {tl,tn} a parent
    }
  when ( recibe mensaje-* (*,t) en el instante t ) {
    if t != tn then error: mala sincronización
    y =  lambda(s)
    envia mensaje-y (y,t) a parent
    s = delta_int(s)
    tl = t
    tn = tl + ta(s)
    envía tn a parent
    }
  when ( recibe mensaje-x (x,t) en el instante t ) {
    if not ( tl <= t <= tn ) then error: mala sincronización
    e = t - tl
    s = delta_ext(s,e,x)
    tl = t
    tn = tl + ta(s)
    envía tn a parent
    }
end Devs-simulator
```

Obsérvese que el algoritmo del simulador abstracto *Devs-simulator* está compuesto por la declaración de las variables y tres cláusulas when. La sintaxis de una *cláusula when* es la siguiente:

```
  when ( evento_condición ) {
    sentencias
    }
```

donde las sentencias del cuerpo de la cláusula when se ejecutan cuando se produce el evento descrito en la condición de la cláusula. A continuación, se describen con detalle las tres cláusulas when del algoritmo *Devs-simulator*.

Mensaje de inicialización

En el instante inicial t_i el modelo DEVS atómico se encuentra en el estado s, en el cual ha estado durante e unidades de tiempo. El estado total del modelo, (s, e), en el instante inicial t_i, se supone conocido.

En ese instante inicial, t_i, el algoritmo *Devs-simulator* debe recibir un mensaje-i, (i, t), que hará que el algoritmo se inicialice correctamente. Debe recibir el mensaje de inicialización al comienzo de cada réplica de la simulación, y antes de recibir ningún otro tipo de mensaje.

La cláusula when de inicialización es la siguiente:

```
when ( recibe mensaje-i (i,t) en el instante t ) {
  tl = t - e
  tn = tl + ta(s)
  envía {tl,tn} a parent
  }
```

es decir, cuando se recibe el mensaje (i, t):

1. El algoritmo asigna valor al instante de tiempo desde el evento anterior, tl, restando el tiempo transcurrido, e, del valor del reloj de la simulación, t, recibido en el mensaje. Es decir: $tl = t - e$.

2. Calcula el instante de tiempo en que se planifica el disparo del siguiente evento interno, tn. Para ello, suma el resultado de evaluar la función de avance en el tiempo, $ta(s)$, con el instante en que se produjo el último evento, tl, que ha sido calculado previamente.

3. El valor de tn calculado se envía al coordinador padre (parent), para indicarle en qué instante está planificado el primer evento interno en este algoritmo *Devs-simulator*. También se envía el valor de tl.

Mensaje de evento interno

Cuando el algoritmo *Devs-simulator* recibe un mensaje de transición interna (*mensaje-**), se dispara la ejecución en el algoritmo de un evento interno. El código de la cláusula when asociada a la recepción de este tipo de mensaje es el siguiente:

```
when ( recibe mensaje-* (*,t) en el instante t ) {
  if t != tn then error: mala sincronización
  y = lambda(s)
  envia mensaje-y (y,t) a parent
  s = delta_int(s)
  tl = t
  tn = tl + ta(s)
  envía tn a parent
  }
```

donde las acciones de la cláusula when son la siguientes:

1. Se comprueba si el instante en que se recibe el mensaje coincide con el instante en que estaba planificada la transición interna. Si no coincide, se ha producido un error de sincronización.

2. Se calcula el valor del evento de salida, v, y el puerto en el que se produce, p. Para ello, se aplica la función de transición interna, $y = \lambda(s)$, donde $y = (p, v)$.

3. Se envía un mensaje (y, t) al simulador padre. En el mensaje-y se especifica y, es decir, el valor calculado del evento de salida (v) y el puerto de salida (p), y el valor actual del reloj de la simulación, t.

4. Se calcula el nuevo estado aplicando la función de transición interna: $s = \delta_{int}(s)$.

5. Se actualiza el instante del último evento al valor actual del reloj de la simulación: $tl = t$.

6. Se calcula el instante para el cual se planifica el disparo del siguiente evento interno, sumándole al valor actual del reloj de la simulación el tiempo durante el cual el sistema permanecerá en el estado actual en ausencia de eventos externos. Es decir, se calcula tn de la expresión siguiente: $tn = tl + ta(s)$.

7. El valor de tn calculado se envía al coordinador padre (parent), para indicarle en qué instante está planificado el siguiente evento interno en este objeto Devs-simulator.

Mensaje de evento externo

Cuando el algoritmo *Devs-simulator* recibe un mensaje de transición externa, (x, t), el algoritmo ejecuta un evento de valor v, en el puerto de entrada p, en el

instante de tiempo actual, t. Se verifica: $x = (p, v)$. El código de la cláusula when es el siguiente:

```
when ( recibe mensaje-x (x,t) en el instante t ) {
  if not ( tl <= t <= tn ) then error: mala sincronización
  e = t - tl
  s = delta_ext(s,e,x)
  tl = t
  tn = tl + ta(s)
  envía tn a parent
  }
```

donde la cláusula when contiene las sentencias siguientes:

1. Se comprueba que el mensaje (x, t) ha llegado en un instante de tiempo anterior o igual al instante en que está planificado el próximo evento interno. Es decir, se comprueba que el valor de t recibido en el mensaje verifica $tl <= t <= tn$. Si no es así, se ha producido un error de sincronización entre los algoritmos simuladores.

2. Se calcula el tiempo transcurrido desde la anterior transición: $e = t - tl$.

3. Se calcula el nuevo estado, evaluando para ello la función de transición externa, pasándole como argumentos el estado total (s, e), que es una variable local del objeto Devs-simulator, y el valor de x recibido en el mensaje: $s = \delta_{ext}(s, e, x)$.

4. Se actualiza el tiempo del último evento al valor actual del reloj de la simulación: $tl = t$.

5. Se calcula el instante de tiempo en que se producirá la siguiente transición interna (en ausencia de eventos externos): $tn = tl + ta(s)$.

6. El valor de tn calculado se envía al coordinador padre (parent), para indicarle en qué instante está planificado el siguiente evento interno en este objeto Devs-simulator.

3.5.4. Devs-coordinator

Como se ha explicado anteriormente, para simular un modelo DEVS modular y jerárquico se asocia un algoritmo *Devs-simulator* a cada modelo DEVS atómico y

un algoritmo *Devs-coordinator* a cada modelo acoplado. Con cada algoritmo *Devs-coordinator* se comunican, desde un nivel jerárquico inmediatamente inferior, sus simuladores hijos: uno por cada componente del modelo acoplado. Estos simuladores son algoritmos *Devs-simulator* o *Devs-coordinator*, dependiendo de si el componente es atómico o compuesto, respectivamente.

Cada simulador *Devs-coordinator* es responsable de la correcta sincronización de sus simuladores hijos, ya sean simuladores *Devs-simulator* u otros simuladores *Devs-coordinator*. Para ello, el simulador *Devs-coordinator* intercambia mensajes con sus hijos y con su padre, y gestiona una lista ordenada de eventos, en la que va almacenando el instante en el que está planificado el próximo evento interno en cada uno de sus simuladores hijo. Esta lista de eventos está ordenada en función de los valores de tn. Cuando hay varios valores de tn iguales, los eventos se ordenan empleando la función *select* del componente acoplado al que está asociado el simulador *Devs-coordinator*.

El primer evento de la lista de eventos es aquel cuyo instante de disparo está más cercano en el tiempo. Si se llama D al conjunto de hijos del algoritmo, el primer evento de la lista puede calcularse de la expresión siguiente:

$$tn = \text{mín}\{\ tn_d \mid d \in D\ \} \tag{3.30}$$

Asimismo, el algoritmo *Devs-coordinator* calcula el instante de tiempo en que se produjo el último evento. Este cálculo se realiza a partir de los instantes de tiempo en que se produjo el último evento en cada uno de sus simuladores hijo:

$$tl = \text{máx}\{\ tl_d \mid d \in D\ \} \tag{3.31}$$

Adicionalmente, el algoritmo *Devs-coordinator* debe gestionar los eventos de entrada que llegan a la interfaz del modelo DEVS acoplado que simula. El algoritmo del simulador abstracto *Devs-coordinator* es el siguiente:

```
Devs-coordinator
  variables:
    parent          // Simulador coordinador padre
    tl              // Instante de tiempo del último evento ejecutado
    tn              // Instante de tiempo del próximo evento planificado
    event-list      // Lista de eventos, (d,tn_d), ordenada por tn_d y select
    d*              // Hijo con evento interno planificado más próximo
  when ( recibe mensaje-i (i,t) en el instante t ) {
    envía mensaje-i (i,t) a cada uno de los simuladores hijo d
```

```
    recibe {tn_d, tl_d} de cada hijo d
    ordena la lista de eventos
    tl = max{ tl_d | d en D }
    tn = min{ tn_d | d en D }
    envía {tn,tl} a parent
    }
  when ( recibe mensaje-* (*,t) en el instante t ) {
    if t != tn then error: mala sincronización
    d* = primer elemento en la lista de eventos
    envia mensaje-* (*,t) a d*
    recibe tn_d* de d*
    actualiza la lista de eventos
    tl = t
    tn = min{ tn_d | d en D }
    envía tn a parent
    }
  when ( recibe mensaje-x (x,t) en el instante t ) {
    if not ( tl <= t <= tn ) then error: mala sincronización
    para cada conexión [ (N,p) , (d*,p_d*) ] {
      envía el mensaje-x ((p_d*,v),t) a d*
      recibe tn_d* de d*
      }
    actualiza la lista de eventos
    tl = t
    tn = min{ tn_d | d en D }
    envía tn a parent
    }
  when ( recibe mensaje-y ((p_d*,v_d*),t) en el instante t de d* ) {
     para cada conexión [ (d*,p_d*) , (N,q) ] {
        envía el mensaje-y ((q,v_d*),t) a parent
        }
     para cada conexión [ (d*,p_d*) , (d,p_d) ]   {
        envía un mensaje-x ((p_d,v_d*),t) al hijo d
        recibe tn_d del hijo d
        }
    actualiza la lista de eventos
    tl = t
    tn = min{ tn_d | d en D }
    envía tn a parent
    }
end Devs-coordinator
```

Como se mencionó anteriormente, el algoritmo *Devs-coordinator* envía y recibe el mismo tipo de mensajes que el algoritmo *Devs-simulator*. Puede recibir, desde su simulador padre, los siguientes mensajes:

- Un mensaje-i de inicialización, (i, t),
- Mensajes-* de transición interna, $(*, t)$ que disparan la ejecución de eventos internos.
- Mensajes-x de transición externa, (x, t), correspondientes a eventos externos de entrada al modelo DEVS acoplado, que disparan la ejecución de eventos externos.

El simulador *Devs-coordinator* envía a su simulador padre mensajes-y, ((v,p),t), que indican que el modelo DEVS compuesto ha generado un evento de salida de valor v, en el puerto p, en el instante t.

Finalmente, el simulador *Devs-coordinator* gestiona los mensajes-y recibidos desde sus simuladores hijo. Un evento de salida de un simulador hijo puede suponer un evento de entrada para otro simulador hijo y también un evento externo de salida del modelo DEVS acoplado. Por ejemplo, en el modelo acoplado mostrado en la Figura 3.9, un evento de salida del componente p_1 supone un evento de entrada en el componente p_2. Asimismo, un evento de salida en el componente p_2 supone un evento externo de salida del modelo DEVS acoplado "pipeline".

A continuación, se explican detalladamente las acciones que realiza el *Devs-coordinator* cuando recibe cada tipo de mensaje.

Mensaje de inicialización

Se supone que se conoce el estado total de cada componente en el instante inicial t_i. En otras palabras, se supone que se conoce (s_d, e_d), para todo $d \in D$. Esto implica que el componente d ha permanecido en el estado s_d durante un tiempo e_d, es decir, desde el instante $t_i - e_d$ hasta el instante inicial t_i.

Para garantizar el correcto funcionamiento del simulador jerárquico, el mensaje de inicialización debe llegar a cada algoritmo *Devs-coordinator* y *Devs-simulator* antes que ningún otro tipo de mensaje.

El algoritmo *Devs-coordinator* contiene una cláusula when con las acciones a ejecutar cuando recibe un mensaje-i de su simulador padre. Esta cláusula when es la siguiente:

```
when ( recibe mensaje-i (i,t) en el instante t ) {
    envía mensaje-i (i,t) a cada uno de los simuladores hijo d
```

```
    recibe {tn_d, tl_d} de cada hijo d
    ordena la lista de eventos
    tl = max{ tl_d | d en D }
    tn = min{ tn_d | d en D }
    envía {tn,tl} a parent
  }
```

Como puede verse en el código anterior, la acciones que realiza cuando recibe el mensaje-i son las siguientes:

1. Reenvía a sus simuladores hijo el mensaje-i.
2. Cada uno de los simuladores hijo procesa el mensaje-i y, una vez hecho esto, cada uno de ellos devuelve una pareja de valores $\{tn_d, tl_d\}$ al *Devs-coordinator*. Así pues, el *Devs-coordinator* recibe parejas de valores $\{tn_d, tl_d\}$ para todo $d \in D$, donde D es el conjunto de simuladores hijo de este *Devs-coordinator*.
3. Los instantes en que están planificados los eventos internos de los hijos se ordenan cronológicamente, de más próximo a más lejano en el tiempo, en una lista o el calendario de eventos. En caso de coincidencia en el instante de disparo del evento interno de varios hijos, se usa la función *Select* del modelo DEVS acoplado para decidir cómo deben ordenarse.
4. Se escoge el mayor valor de tl_d y se asigna a tl.
5. Se escoge el menor valor de tn_d y se asigna a tn. Así pues, tn contiene el más reciente de los eventos internos planificados en los simuladores hijo.
6. Finalmente, el *Devs-coordinator* envía los valores de tn y tl obtenidos anteriormente a su simulador padre.

Mensaje de evento interno

La siguiente cláusula when realiza las acciones que se ejecutan cuando el *Devs-coordinator* recibe un mensaje-* de su simulador padre. La cláusula when es la siguiente:

```
when ( recibe mensaje-* (*,t) en el instante t ) {
  if t != tn then error: mala sincronización
  d* = primer elemento en la lista de eventos
  envia mensaje-* (*,t) a d*
```

```
recibe tn_d* de d*
actualiza la lista de eventos
tl = t
tn = min{ tn_d | d en D }
envía tn a parent
}
```

Las acciones que se realizan son las siguientes:

1. En primer lugar, se comprueba que el valor de la variable tiempo del mensaje recibido, t, coincide con el instante en que está planificado el evento interno más próximo. Si no coincidiera, se ha producido un error de coordinación.

2. El simulador *Devs-coordinator* sabe a cual de sus simuladores hijo corresponde el evento interno del mensaje: a aquel que ocupa el primer puesto en la lista de eventos. Llamemos d^* al simulador hijo que ocupa el primer puesto en la lista de eventos.

3. El simulador *Devs-coordinator* envía el mensaje-* al simulador hijo d^*.

4. El simulador d^* ejecuta la transición interna de su estado y, si como resultado de la transición interna genera un evento de salida, además envía un mensaje-y, (y_{d^*}, t), a su simulador padre, es decir, al simulador *Devs-coordinator*. La recepción de este mensaje-y proveniente del simulador hijo d^* desencadena una serie de acciones, como veremos en el siguiente epígrafe.

5. Una vez el simulador d^* ha ejecutado el evento interno, envía el valor tn_{d^*} al simulador *Devs-coordinator*, en el que le indica el instante para el cual está planificado su siguiente evento interno.

6. El simulador *Devs-coordinator* elimina de la lista de eventos el evento que acaba de pasar a su simulador hijo d^*, e inserta en la lista de eventos el próximo evento planificado en este simulador hijo.

7. El simulador *Devs-coordinator* actualiza el valor de sus variables tn y tl. Asigna a tl el valor actual del reloj de la simulación, $tl = t$, y asigna a tn el instante de tiempo del primer evento de la lista de eventos: $tn = \text{mín}\{tn_d | d \in D\}$.

8. Finalmente, envía el valor de tn a su simulador padre (parent).

Mensaje de salida de un simulador hijo

Cuando el simulador *Devs-coordinator* recibe un mensaje-y del tipo (y_{d^*}, t), significa que en el instante t, el simulador hijo d^* ha generado un evento de salida de valor v_{d^*} a través de su puerto de salida p_{d^*}. Se verifica que: $y_{d^*} = (p_{d^*}, v_{d^*})$. Por ello, en el algoritmo el mensaje-y se escribe de la forma siguiente: $((p_{d^*}, v_{d^*}), t)$. La siguiente cláusula when describe las acciones asociadas a la recepción de dicho mensaje:

```
when ( recibe mensaje-y ((p_d*,v_d*),t) en el instante t de d* ) {
   para cada conexión [ (d*,p_d*) , (N,q) ] {
      envía el mensaje-y ((q,v_d*),t) a parent
      }
   para cada conexión [ (d*,p_d*) , (d,p_d) ]   {
      envía un mensaje-x ((p_d,v_d*),t) al hijo d
      recibe tn_d del hijo d
      }
  actualiza la lista de eventos
  tl = t
  tn = min{ tn_d | d en D }
  envía tn a parent
  }
```

Las acciones ejecutadas como consecuencia de recibir el mensaje (y_{d^*}, t) son las siguientes:

1. El simulador *Devs-coordinator* inspecciona las conexiones entre el puerto de salida p_{d^*} y los puertos de salida del modelo acoplado. Representamos mediante $[(d^*, p_{d^*}), (N, q)]$ este tipo de conexiones, donde N representa el modelo DEVS acoplado y q el puerto de salida del modelo DEVS acoplado al cual está conectado el puerto de salida p_{d^*} del componente d^*.

   Para cada una de estas conexiones, se envía el siguiente mensaje-y a parent (el simulador padre de *Devs-coordinator*): $((q, v_{d^*}), t)$, que indica que se ha generado un evento de salida de valor v_{d^*} en el puerto de salida q del DEVS acoplado, en el instante t.

2. El simulador *Devs-coordinator* inspecciona las conexiones entre el puerto de salida p_{d^*} de d^* y los puertos de entrada de los demás componentes d, con $d \in D$. Este tipo de conexiones se representan de la forma siguiente: $[(d^*, p_{d^*}), (d, p_d)]$.

   Para cada una de estas conexiones entre d^* y un componente d, *Devs-coordinator* envía el siguiente mensaje-x al componente d: $((p_d, v_{d^*}), t)$. Este mensaje dis-

para la ejecución de un evento externo de valor v_{d^*}, en el puerto de entrada p_d, en el instante t.

Una vez ejecutadas las transiciones externas, los componentes d envían al *Devs-coordinator* los instantes para los que están planificadas sus próximas transiciones internas, tn_d.

3. El *Devs-coordinator* elimina de la lista los eventos que estaban programados para estos componentes, que ya no llegarán a producirse debido a la transiciones externas que acaban de ejecutarse, e inserta la nueva planificación de eventos internos que acaba de recibir.

4. Finalmente, el algoritmo *Devs-coordinator* actualiza los valores de sus variables tl y tn, y envía el valor de tn a parent.

Mensaje de evento externo

Un mensaje-x, (x, t), enviado al algoritmo *Devs-coordinator* desde su padre, produce el disparo de un evento externo de valor v, en el puerto de entrada p del modelo DEVS acoplado asociado al *Devs-coordinator*, en el instante t. Obsérvese que el valor del evento externo, v, y el puerto de entrada, p, son transmitidos en el mensaje, ya que x es la pareja de valores (p, v). Es decir: $x = (p, v)$.

Cuando el algoritmo *Devs-coordinator* recibe un mensaje-x de su padre, entonces ejecuta las acciones descritas en la cláusula when siguiente:

```
when ( recibe mensaje-x (x,t) en el instante t ) {
  if not ( tl <= t <= tn ) then error: mala sincronización
  para cada conexión [ (N,p) , (d*,p_d*) ] {
    envía el mensaje-x ((p_d*,v),t) a d*
    recibe tn_d* de d*
    actualiza la lista de eventos
    }
  tl = t
  tn = min{ tn_d | d en D }
  envía tn a parent
  }
```

El simulador *Devs-coordinator* ejecuta las acciones siguientes cuando recibe un mensaje (x, t) de su padre:

1. En primer lugar, comprueba que el evento externo no se ha recibido posteriormente al instante en que está planificado el siguiente evento interno, tn. Si

no se satisface la condición $tl \leq t \leq tn$, entonces se ha producido un error de sincronización entre los algoritmos.

2. El algoritmo *Devs-coordinator* analiza las conexiones entre los componentes, con el fin de identificar qué puertos de entrada de los componentes están directamente conectados al puerto p del modelo compuesto, que es donde se ha producido el evento externo de entrada. Este tipo de conexiones, entre el puerto de entrada p del modelo compuesto N y el puerto de entrada p_{d^*} del componente d^*, se representan de la forma siguiente: $[(N, p), (d^*, p_{d^*})]$.

3. Para cada una de las conexiones anteriores entre el modelo compuesto y uno de los submodelos, que llamamos d^*, el algoritmo *Devs-coordinator* envía un mensaje-x al submodelo d^*. El mensaje-x fuerza el disparo de un evento de entrada de valor v, en el puerto de entrada p_{d^*} del componente d, en el instante t.

   Tras ejecutar su evento externo de entrada, cada uno de los componentes d^* envía al algoritmo *Devs-coordinator* el instante en el cual tiene planificado su siguiente evento interno, tn_{d^*}. El algoritmo *Devs-coordinator* actualiza su lista de eventos, eliminando el evento interno que estaba planificado para cada componente d^* e insertando en la lista de eventos los nuevos instantes de disparo tn_{d^*}, para todos los componentes d^*.

4. Finalmente, el algoritmo *Devs-coordinator* recalcula el valor de sus variables tl y tn, y envía el valor calculado de tn a su simulador padre.

3.5.5. Devs-root-coordinator

El algoritmo *Devs-root-coordinator*, situado en el nivel jerárquico superior del simulador abstracto (véase, por ejemplo, la Figura 3.11), gestiona el avance del reloj de la simulación, t.

Realiza su tarea enviando mensajes al algoritmo del nivel jerárquico inmediatamente inferior, que es el simulador asociado al modelo DEVS completo. Si el modelo DEVS completo es acoplado, el algoritmo hijo de *Devs-root-coordinator* será un algoritmo *Devs-coordinator*. Si el modelo DEVS completo es atómico, el algoritmo hijo será *Devs-simulator*. En cualquier caso, *Devs-root-coordinator* tiene un único simulador hijo. El algoritmo *Devs-root-coordinator* es el siguiente:

```
Devs-root-coordinator
  variables:
```

```
    t     // Reloj de la simulación
    hijo  // Algoritmo de nivel jerárquico inmediatamente inferior
  t = t0
  envía mensaje-i (i,t) a hijo
  recibe tn de hijo
  t = tn
  loop
    envía mensaje-* (*,t) a hijo
    recibe tn de hijo
    t = tn
  until final de simulación
end Devs-root-coordinator
```

Como puede verse en el código anterior, las acciones que realiza el algoritmo son las siguientes:

1. Inicialmente, *Devs-root-coordinator* envía un mensaje-i a su simulador hijo, que es propagado a través de todo el árbol jerárquico del simulador abstracto, inicializando todos los simuladores. En respuesta, el algoritmo hijo responde enviando a *Devs-root-coordinator* el instante de tiempo para el que está planificada la próxima transición interna, tn.

2. El algoritmo *Devs-root-coordinator* avanza el reloj de la simulación, t, hasta el valor tn que ha recibido de su hijo. Es decir, asigna: $t = tn$.

3. El algoritmo *Devs-root-coordinator* envía a su hijo un mensaje-*, $(*, t)$, que fuerza la ejecución del evento interno más reciente. En respuesta el hijo envía a *Devs-root-coordinator* el instante para el cual está planificado el siguiente evento interno, tn.

4. El algoritmo *Devs-root-coordinator* avanza el reloj de la simulación, t, hasta el valor tn que ha recibido de su hijo.

5. *Devs-root-coordinator* comprueba si se satisface la condición de finalización de la simulación. En caso afirmativo, finaliza la simulación. En caso contrario, vuelve al Paso 3.

3.6. MODELOS DEVS ACOPLADOS NO MODULARMENTE

En la Sección 3.4 se describió la construcción modular de modelos de eventos discretos. Los modelos DEVS con *acoplamiento modular* están formados de compo-

nentes, que son sistemas DEVS, que interactúan sólo a través de la conexión entre los puertos de sus interfaces.

En los modelos DEVS con *acoplamiento no modular*, los componentes influyen sobre los otros componentes a través de la función de transición de estados. Los eventos que ocurren en un componente pueden producir cambios en el estado, así como planificación o cancelación de eventos en otros componentes. Este tipo de modelos acoplados no modularmente, en los cuales los componentes son modelos DEVS, se denominan *DEVS multicomponente*.

3.6.1. DEVS multicomponente

Un modelo DEVS multicomponente es la tupla siguiente:

$$\text{multiDEVS} = \langle X, Y, D, \{M_d\}, Select \rangle \tag{3.32}$$

donde:

X	Conjunto de eventos de entrada.
Y	Conjunto de eventos de salida.
D	Conjunto de referencias a los componentes.
$Select : 2^D \rightarrow D$	Función select. Establece la prioridad en caso de que varios componentes tengan planificado un evento interno para el mismo instante.

además, cada componente $d \in D$ se define de la forma siguiente:

$$M_d = \langle S_d, I_d, E_d, \delta_{ext,d}, \delta_{int,d}, \lambda_d, ta_d \rangle \tag{3.33}$$

donde

S_d	Conjunto de estados secuenciales de d.

$Q_d = \{ \ (s_d, e_d) \mid s_d \in S_d, \ e_d \in \mathbb{R}_0^+ \ \}$	Estados totales de d. El estado total es el estado del sistema, s_d, y el tiempo transcurrido e_d desde la última transición (interna o externa) del estado de d.
$I_d \subseteq D$	Conjunto de componentes cuyo estado influye en el estado de d.
$E_d \subseteq D$	Conjunto de componentes cuyo estado es influido por el estado de d.
$\delta_{ext,d} : \underset{i \in I_d}{\times} Q_i \times X \rightarrow \underset{j \in E_d}{\times} Q_j$	Función de transición externa del estado. Los argumentos de entrada de la función son el estado total de cada uno de los componentes que influyen sobre d y el evento de entrada (X). Se representa $\underset{i \in I_d}{\times} Q_i$ el producto cartesiano del estado total de todos los componentes que influyen sobre d. La función devuelve el estado total de todos los componentes que son influenciados por d. Análogamente, $\underset{j \in E_d}{\times} Q_j$ representa el producto cartesiano del estado total de todos los componentes que son influenciados por d.
$\delta_{int,d} : \underset{i \in I_d}{\times} Q_i \rightarrow \underset{j \in E_d}{\times} Q_j$	Función de transición interna del estado. Los argumentos de entrada de la función son el estado total de cada uno de los componentes que influyen sobre d. La función devuelve el estado total de todos los componentes que son influenciados por d.
$\lambda_d : \underset{i \in I_d}{\times} Q_i \rightarrow Y$	Función de salida. Los argumentos de la función son el estado total de todos los componentes que influyen en d. La función devuelve un elemento del conjunto de salida del multiDEVS.

$ta_d : \underset{i \in I_d}{\times} Q_i \rightarrow \mathbb{R}^+_{0,\infty}$	Función de avance en el tiempo. Los argumentos de entrada a la función son el estado total de todos los componentes que influyen en d. Devuelve un valor real positivo, incluyendo el cero y el infinito.

El funcionamiento de un modelo DEVS multicomponente, también llamado multiDEVS, es el siguiente.

- Los eventos internos de cada componente son planificados individualmente por cada componente $d \in D$, empleando para ello su función de avance en el tiempo ta_d.
- Cuando se dispara un evento interno en uno de los componentes $d \in D$, el evento es ejecutado, resultando en:
  - Cambios en el estado, que vienen dados por la función de transición interna $\delta_{int,d}$. A partir del estado total de todos los componentes que influyen sobre d, la función de transición interna calcula el nuevo estado total de todos los componentes que son influidos por d.
  - La generación de eventos de salida en el multiDEVS, que viene dada por la función de salida λ_d, pasando como argumentos a la función el estado total de todos los componentes que influyen en d.
- Cuando están planificados para un mismo instante eventos internos en diferentes componentes, se emplea la función *Select* para decidir cuál de ellos es ejecutado.
- Cuando se produce un evento externo en la interfaz de entrada del multiDEVS, entonces reaccionan ante él los componentes d que tienen definida una función de transición externa $\delta_{ext,d}$. No reaccionan ante los eventos externos aquellos componentes que no tienen definida la función de transición externa.

3.6.2. MultiDEVS definido como DEVS

Tal como se describe a continuación, un modelo multiDEVS puede interpretarse como un modelo DEVS. Sea el modelo DEVS:

$$\text{DEVS} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle \tag{3.34}$$

El conjunto de estados síncronos del modelo DEVS, S, es el producto cartesiano de los conjuntos de posibles estados totales de los componentes del multiDEVS.

$$S = \underset{d \in D}{\times} Q_d \tag{3.35}$$

La expresión anterior indica que el estado del modelo DEVS queda definido si se conoce el estado total de cada uno de los componentes del modelo multiDEVS.

El tiempo hasta el siguiente evento interno en el modelo DEVS se calcula de la forma siguiente:

$$ta(s) = \text{mín}\{\ \sigma_d \mid d \in D\ \} \tag{3.36}$$

donde σ_d es el tiempo hasta el siguiente evento interno planificado en el componente d del modelo multiDEVS. El valor de σ_d puede calcularse evaluando ta_d y restando e_d al valor obtenido. Es decir:

$$\sigma_d = ta_d(\dots, q_i, \dots) - e_d \qquad \text{con } i \in I_d \tag{3.37}$$

donde la notación "$ta_d(\dots, q_i, \dots)$ con $i \in I_d$" representa que la función de avance en el tiempo del componente d tiene como argumentos el estado total de todos los componentes que influyen sobre d.

El modelo d del multiDEVS que sufre la siguiente transición en el estado es aquel que tiene el menor valor σ_d. En caso de que este valor coincida para varios modelos del multiDEVS, entonces se usa la función *Select* del multiDEVS para decidir cuál de ellos sufre la transición interna.

Llamemos d^* al modelo del multiDEVS que sufrirá la siguiente transición interna. La función de transición interna del modelo DEVS puede definirse de la forma siguiente:

$$\delta_{int}\left((s_1, e_1), \dots, (s_n, e_n)\right) = \left((s'_1, e'_1), \dots, (s'_n, e'_n)\right) \tag{3.38}$$

donde:

$$(s'_j, e'_j) = \begin{cases} (s_j, e_j + ta(s)) & \text{si } j \notin E_{d^*} \\ \delta_{int,d^*}(\dots, (s_i, e_i + ta(s)), \dots) & \text{si } j \in E_{d^*}, \text{ con } i \in I_{d^*} \end{cases} \tag{3.39}$$

es decir, la función de transición interna del modelo DEVS está definida por la función de transición interna del componente d^* del multiDEVS. Igualmente, la salida del modelo DEVS está definida por la función de salida del componente d^* del multiDEVS:

$$\lambda(s) = \lambda_{d^*}(\dots, q_i, \dots) \quad \text{con } i \in I_{d^*} \tag{3.40}$$

Análogamente, la función de transición externa del modelo DEVS se define, para un evento externo x, como el resultado de la ejecución de las funciones de transición externa de todos los componentes del multiDEVS.

3.6.3. Modelo multiDEVS del Juego de la Vida

El autómata celular del Juego de la Vida, descrito en la Sección 2.4.1, puede modelarse como un multiDEVS de componentes idénticos, distribuidos uniformemente y con acoplos entre ellos uniformes. El modelo multiDEVS de este autómata celular tiene las características siguientes:

- El conjunto de componentes que influyen al componente (i, j), $I_{i,j}$, así como el conjunto de componentes que son influidos por él, $E_{i,j}$, son la propia célula (i, j) más las células vecinas.
- La función de avance en el tiempo de cada célula debe describir su nacimiento y muerte, basándose para ello en su valor actual del factor ambiental y en el estado de las células vecinas. La función de avance toma el valor infinito en caso de que no se planifique ningún evento.
- Un evento en el estado en una célula puede modificar los eventos planificados en sus células vecinas.

El estado de la célula $M_{i,j}$ puede definirse de la forma siguiente:

$$S_{i,j} = \{fase_{i,j}, factorAmbiental_{i,j}\} \tag{3.41}$$

donde la variable de estado $fase$ puede tomar los valores {“viva”, “muerta”} y la variable de estado $factorAmbiental$ representa el valor actual del factor ambiental de la célula.

El tiempo hasta la siguiente transición interna de la célula (i, j) depende de su estado, del estado de las células vecinas y del valor actual de su factor ambiental. Llamando $vecVivas_{i,j}$ al número de células vecinas de (i, j) que están vivas, puede definirse la variable $coef_{i,j}$, que representa el aumento o reducción del factor ambiental de la célula (i, j), de la forma siguiente:

$$coef_{i,j} = \begin{cases} 1 & \text{si } vecVivas_{i,j} = 3 \\ -1 & \text{si } fase_{i,j} = \text{“viva” and } (vecVivas_{i,j} < 2 \text{ or } vecVivas_{i,j} > 3) \\ 0 & \text{en cualquier otro caso} \end{cases} \tag{3.42}$$

Esto significa que cuando el número de células vecinas vivas es igual a 3, el factor ambiental aumenta con pendiente 1. Cuando la célula está viva y el número de células vecinas es menor que 2 ó mayor que 3, entonces el factor ambiental aumenta con pendiente -1, es decir, decrece. Obsérvese que sólo han de planificarse eventos en los dos casos siguientes:

- Cuando la célula (i, j) está muerta y $coef_{i,j} = 1$, en cuyo caso debe planificarse el nacimiento. Sabiendo que el factor ambiental de una célula muerta es -2 y que la pendiente con que aumenta el factor ambiental es 1, el evento “nacimiento” se planifica para dentro de 2 unidades de tiempo.
- Cuando la célula (i, j) está viva y $coef_{i,j} = -1$, en cuyo caso debe planificarse el evento de la muerte de la célula. Conocido el valor actual del factor ambiental y el hecho de que éste evoluciona con pendiente -1, puede calcularse fácilmente el tiempo hasta el evento.

La función de transición interna de la célula (i, j) debe describir el nacimiento o muerte de dicha célula. Asimismo, debe recalcular la variable $coef$ de las células vecinas y el instante en que se producirá el siguiente evento en cada una de ellas.

3.6.4. Modularización

En los sistemas multicomponente con acoplamiento no modular, los componentes pueden acceder y escribir directamente en las variables de estado de los demás componentes. Desde el punto de vista del acoplamiento modular, esto supone una violación del *principio de modularidad*, según el cual la interacción entre los componentes debe producirse únicamente a través de sus interfaces.

A continuación, se describe un procedimiento para traducir el acoplamiento no modular en acoplamiento modular, demostrándose asimismo que esta transformación siempre puede realizarse. El procedimiento consiste en identificar la dependencia entre los componentes y, en función de dicha dependencia, definir las interfaces y la conexión entre ellas. Este proceso de introducción de interfaces para describir el acoplamiento se denomina *modularización*.

Pueden distinguirse dos tipos de acoplamiento no modular, que deben ser tratados de manera diferente:

1. Un componente A tiene acceso para escribir en una variable v de un componente B. Esta comunicación debe transformarse en una conexión entre las interfaces de A y B: se define un puerto de salida en A, *vout*, un puerto de entrada en B, *vin*, y se conectan ambos. Cada vez que A quiera escribir en la variable v de B, en la versión modular A debe generar un evento de salida, a través del puerto *vout*, cuyo valor sea igual al que se desea escribir en v. A través de la conexión, esto resulta en un evento de entrada en el puerto *vin* de B. La función de transición externa de B debe reaccionar escribiendo el valor recibido en la variable v. En la Figura 3.13 se representa esquemáticamente el proceso de modularización.

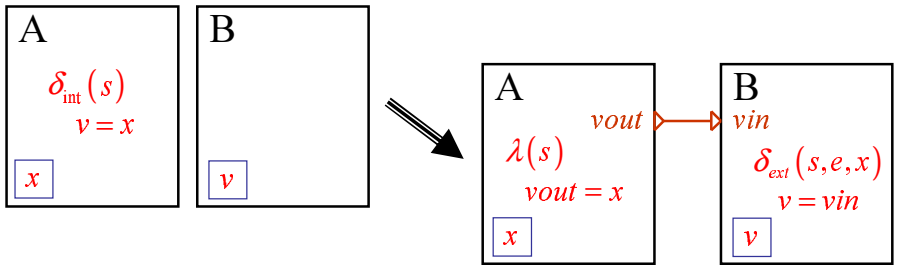

Figura 3.13: El componente A tiene acceso para escribir en la variable v del componente B.

2. Un componente B tiene acceso de lectura a la variable u del componente A. Esta situación es más complicada, puesto que B debe tener acceso en todo momento al valor actual de la variable. Para resolver la situación, B guarda una copia del valor de la variable u en su propia memoria. Denominaremos a esta copia *ucopy*. Ahora bien, cada vez que A cambia el valor de u, B debe ser informado del cambio. Esto se consigue conectando los puertos *uout* y *uin*, de A y B respectivamente, de manera simular a como se hizo en el caso anterior. La función de transición externa de B debe actualizar el valor de *ucopy*. En la Figura 3.14 se representa esquemáticamente el proceso de modularización.

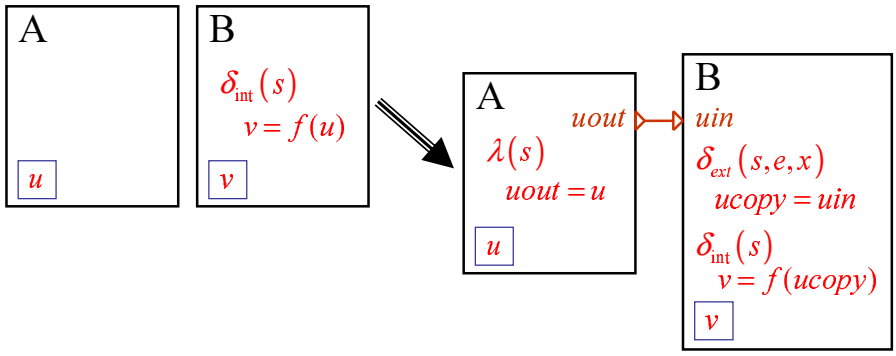

Figura 3.14: El componente B tiene acceso para leer la variable u del componente A.

3.7. EJERCICIOS DE AUTOCOMPROBACIÓN

Ejercicio 3.1

Dibuje, para una trayectoria de entrada de su elección, el estado y la trayectoria de salida del modelo DEVS del contador binario.

Ejercicio 3.2

Defina un contador DEVS SISO que cuente el número de entradas con valor diferente de cero que ha recibido desde su inicialización y que devuelva este valor cuando reciba una entrada de valor cero.

Ejercicio 3.3

Describa el modelo de un sistema, empleando el formalismo DEVS, cuyo comportamiento es el siguiente. Cuando el sistema recibe un evento de entrada de valor real positivo x, genera, tras x unidades de tiempo, una salida de valor $x/2$. Durante este *tiempo de proceso* de duración x, el sistema ignora los eventos de entrada. Además, represente gráficamente una secuencia de eventos de entrada de su elección y la correspondiente evolución del estado y de la trayectoria de salida.

Ejercicio 3.4

Consideremos un multiplexor 2:1 que tiene dos entradas de datos (X_0, X_1), una entrada de selección (S) y una salida (Y). Las señales de entrada y salida son binarias, pudiendo tomar dos posibles valores: $\{0, 1\}$.

Es posible realizar dos tipos de modelo de este circuito: un modelo de tiempo continuo y un modelo de eventos discretos.

En el *modelo de tiempo continuo*, las trayectorias de entrada y salida son señales constantes a tramos. En la Figura 3.15a se muestra un ejemplo del comportamiento de tiempo continuo del multiplexor. Mientras S vale 0, la salida Y es igual al valor de la entrada X_0. Asimismo, mientras S vale 1, la salida Y es igual al valor de la

entrada X_1. En la Figura 3.15a se ha señalado en color rojo los segmentos de las trayectorias de entrada de datos (X_0, X_1) que el circuito transmite a la salida.

En el *modelo de eventos discretos* del multiplexor las trayectorias de entrada y salida son eventos, los cuales pueden tomar dos valores: $\{0, 1\}$. La descripción de eventos discretos de una señal binaria consiste en una secuencia de eventos, que se producen en el instante en que la señal cambia de valor. El valor del evento es igual al nuevo valor que toma la señal. Es decir, el instante en que la señal cambia de 0 a 1 está definido por un evento de valor 1. Análogamente, el instante en que la señal cambia de 1 a 0 está definido por un evento de valor 0.

En la Figura 3.15b se muestra un ejemplo del comportamiento de eventos discretos del multiplexor. Los eventos de entrada señalan los cambios en las señales de entrada. Sólo se producen eventos de salida cuando cambia el valor de la señal de salida.

Describa, empleando el formalismo DEVS clásico, el modelo de eventos discretos del multiplexor 2:1.

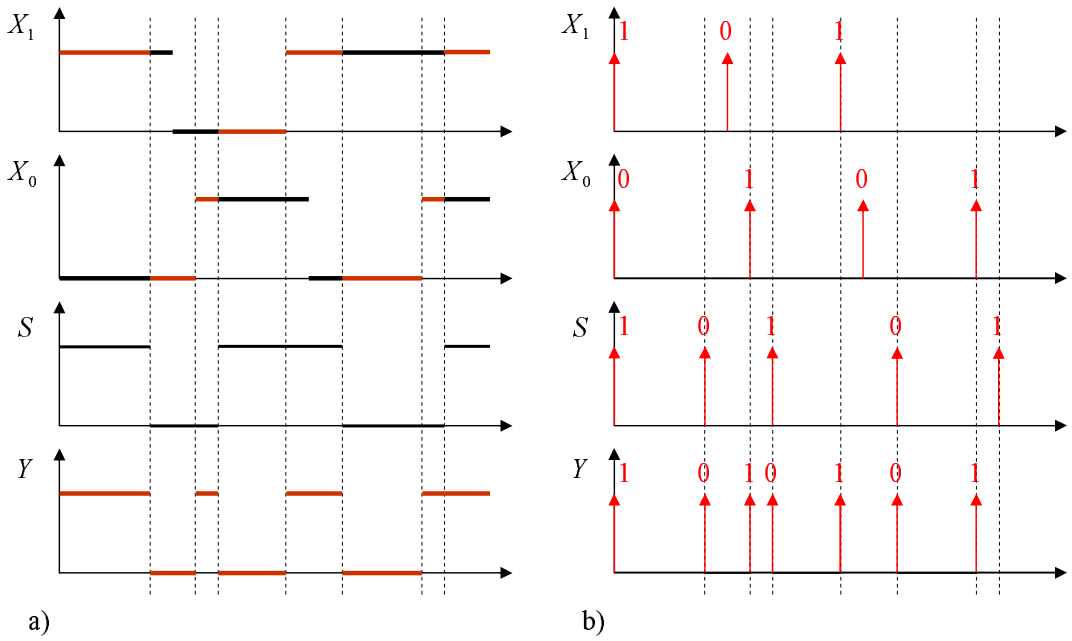

Figura 3.15: MUX 2:1. Ejemplo de comportamiento: a) del modelo de tiempo continuo; y b) del modelo eventos discretos.

Ejercicio 3.5

Describa, empleando el formalismo DEVS clásico, el modelo de un sistema que introduce un retardo aleatorio a las entidades. El sistema tiene una entrada y una salida. Las entidades llegan de una en una al sistema. Cada entidad tiene un número que la identifica de forma unívoca. La llegada de una entidad corresponde con un evento de entrada cuyo valor es igual al número identificativo de la entidad recibida.

Cuando llega una entidad, en el sistema se obtiene una observación aleatoria de la distribución U(1,2) minutos, que es el tiempo durante el cual la entidad permanecerá en el sistema. Transcurrido ese tiempo, la entidad abandona el sistema: se genera un evento de salida cuyo valor es el número identificativo de la entidad.

Obsérvese que puesto que el tiempo que permanece cada entidad en el sistema es aleatorio, puede suceder que haya varias entidades "esperando" en el sistema y que las entidades no abandonen el sistema en el mismo orden en que han llegado.

Ejercicio 3.6

Describa el modelo DEVS multicomponente del autómata celular unidimensional descrito a continuación.

Consideremos un objeto que se mueve a lo largo de una línea recta. La velocidad con la que se mueve el objeto se modifica instantáneamente mediante eventos externos, cuyo valor representa el nuevo valor de la velocidad del objeto. Cuando la velocidad es positiva, el objeto de mueve hacia la derecha. Cuando es negativa, el objeto de mueve hacia la izquierda. Cuando la velocidad es cero, el objeto se queda parado.

Suponga que la longitud de la recta, que es igual a 20 metros, se divide en 20 células de 1 metro de longitud cada una, y que la primera y la última células se encuentran unidas entre sí.

El modelo debe describir el movimiento del objeto en el espacio celular. Cada una de las células puede estar en dos estados, $\{0, 1\}$, dependiendo de que el objeto se encuentre (estado=1) o no (estado=0) en ella.

Tenga en cuenta que el tiempo que transcurre desde que el objeto entra en una célula, hasta que la abandona para entrar en la siguiente, depende de la velocidad del objeto.

Ejercicio 3.7

Describa el modelo de un sistema de bifurcación aleatoria de las entidades (denominado comúnmente *bloque de decisión*), empleando el formalismo DEVS clásico. El sistema tiene una única entrada, a través de la cual las entidades llegan de una en una. Cada entidad tiene asociado un número que la caracteriza de forma unívoca. La llegada de una entidad corresponde con un evento de entrada, cuyo valor coincide con el número identificativo de la entidad.

El sistema tiene dos salidas. Un parámetro del sistema, p, define la probabilidad de que la entidad abandone el sistema por la *salida 1*. Si la entidad no abandona el sistema por la *salida 1*, lo abandona por la *salida 2*. Así pues, cuando llega una entidad, se genera una observación, u, de la distribución $U(0,1)$. Si $u < p$, entonces la entidad abandona el sistema por la *salida 1*. En caso contrario, lo abandona por la *salida 2*. En cualquier caso, el tiempo de residencia de la entidad es el sistema es cero.

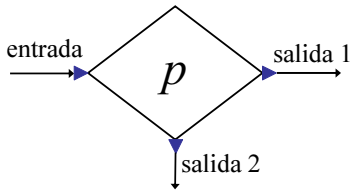

Figura 3.16: Bloque de decisión.

Ejercicio 3.8

Describa el modelo compuesto mostrado en la Figura 3.17 mediante el formalismo DEVS clásico.

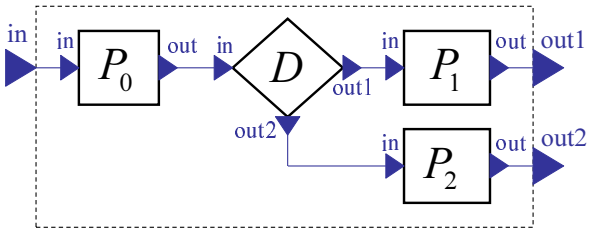

Figura 3.17: Modelo compuesto.

Ejercicio 3.9

Empleando el formalismo DEVS clásico, describa el modelo de un biestable tipo D con dos entradas (D y Ck) y dos salidas (Q y $\bar{Q}$).

3.8. SOLUCIONES A LOS EJERCICIOS

Solución al Ejercicio 3.1

El funcionamiento del contador binario está descrito en el Tema 3 del texto de teoría. Los eventos de entrada pueden tomar valor cero o uno. Los eventos de salida toman valor uno. Se genera un evento de salida cada dos eventos de entrada de valor uno. Para ello, el sistema contiene un contador módulo 2 del número de eventos de entrada de valor uno recibidos hasta el momento. Este contador es la variable de estado *cuenta*.

En la Figura 3.18 se muestra la trayectoria de salida y la evolución de la variable de estado *cuenta* para una determinada trayectoria de entrada.

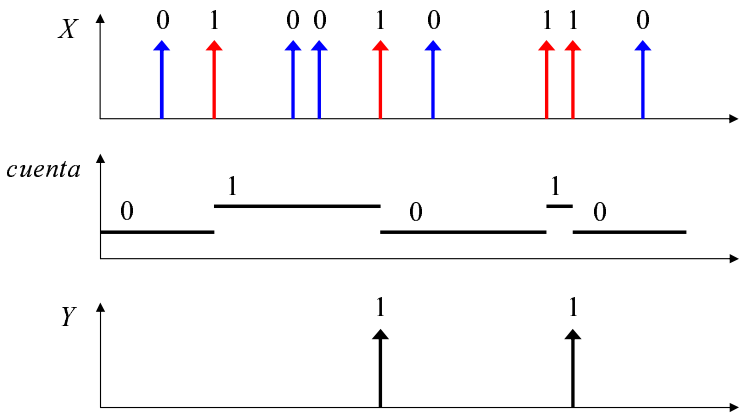

Figura 3.18: Trayectoria de salida y evolución del estado del contador binario para una trayectoria de entrada.

Solución al Ejercicio 3.2

El estado del contador está compuesto por tres variables de estado, $(fase, \sigma, cuenta)$. Los valores posibles del estado son:

$$S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \mathbb{N} \tag{3.43}$$

El comportamiento del sistema puede describirse de la manera siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.44}$$

donde:

$$\begin{array}{lcl}
X & = & \mathbb{R} \\
Y & = & \mathbb{N} \\
S & = & \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \mathbb{N} \\
\delta_{ext}\left(\text{“pasivo”}, \sigma, cuenta, e, x\right) & = & \begin{cases} (\text{“pasivo”}, \sigma - e, cuenta + 1) & \text{si } x \neq 0 \\ (\text{“activo”}, 0, cuenta) & \text{si } x = 0 \end{cases} \\
\delta_{int}\left(fase, \sigma, cuenta\right) & = & (\text{“pasivo”}, \infty, cuenta) \\
\lambda\left(\text{“activo”}, \sigma, cuenta\right) & = & cuenta \\
ta\left(fase, \sigma, cuenta\right) & = & \sigma
\end{array} \tag{3.45}$$

La inicialización del modelo consistiría en asignar el siguiente valor al estado:

$$(fase, \sigma, cuenta) = (\text{“pasivo”}, \infty, 0) \tag{3.46}$$

Para ilustrar el comportamiento del modelo, en la Figura 3.19 se muestra la evolución del estado y la trayectoria de salida del contador, para una determinada trayectoria de entrada.

Obsérvese que en los instantes t_1 y t_4 se producen sendos eventos de entrada de valor cero. Al ejecutar la función de transición externa, el sistema pasa al estado transitorio (“activo”, 0, *cuenta*). Puesto que en este estado $\sigma = 0$, a continuación, en el mismo instante de tiempo, tiene lugar una transición interna. Se genera un evento de salida de valor igual al valor que ese instante tiene *cuenta* y se produce una transición de estado, definida por la función de transición interna. El estado resultante de la transición interna es (“pasivo”, ∞, *cuenta*).

Solución al Ejercicio 3.3

El estado del sistema está compuesto por las tres variables de estado siguientes: $(fase, \sigma, almacena)$.

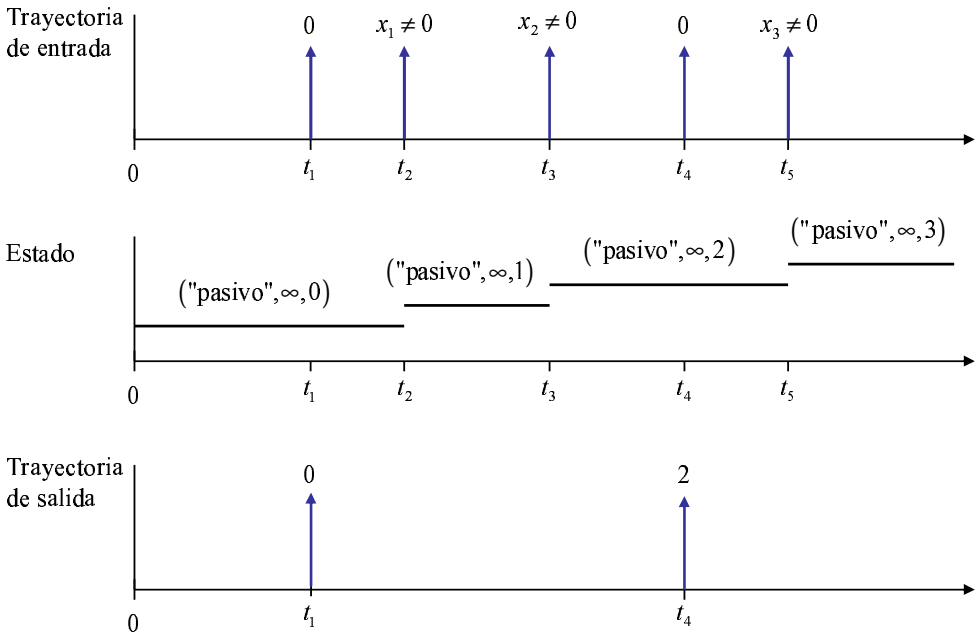

Figura 3.19: Ejemplo de evolución del contador SISO.

La variable de estado $fase$ puede tomar los valores {"pasivo", "responde"}. Se guarda en la variable *almacena* el valor del evento de entrada almacenado en el sistema. Este valor puede ser cualquier número real.

El comportamiento del sistema puede describirse de la manera siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.47}$$

donde:

$$\begin{array}{lcl}
X & = & \mathbb{R}_0^+ \\
Y & = & \mathbb{R}_0^+ \\
S & = & \{\text{“pasivo”}, \text{“responde”}\} \times \mathbb{R}_0^+ \times \mathbb{R} \\
\delta_{ext}\,(S, e, x) & = & \begin{cases} (\text{“responde”}, x, x) & \text{si } fase = \text{“pasivo”} \\ (\text{“responde”}, \sigma - e, almacena) & \text{si } fase = \text{“responde”} \end{cases} \\
\delta_{int}\,(S) & = & (\text{“pasivo”}, \infty, almacena) \\
\lambda\,(S) & = & almacena/2 \\
ta\,(S) & = & \sigma
\end{array} \tag{3.48}$$

La inicialización del modelo podría consistir en asignar el siguiente valor al estado:

$$(fase, \sigma, almacena) = (\text{“pasivo”}, \infty, 0) \tag{3.49}$$

Solución al Ejercicio 3.4

Obsérvese que el multiplexor sólo genera un evento de salida cuando se produce un evento de entrada y el valor de la salida correspondiente es diferente del anterior evento de salida. Esto es equivalente a decir que los eventos de salida se producen únicamente en los instantes en que la señal de salida cambia su valor.

El multiplexor de tiempo continuo es un sistema sin memoria: un circuito combinacional cuya función lógica es $Y = X_0 \cdot \bar{S} + X_1 \cdot S$. Por el contrario, el multiplexor de tiempo discreto es un sistema con memoria. Cuando llega un evento de entrada a alguno de sus puertos, el sistema debe “recordar” el valor de los últimos eventos que se produjeron en los otros puertos, con el fin de evaluar la nueva salida, y debe comparar esta nueva salida con el valor del último evento de salida, para decidir si debe o no generarse un evento de salida. El sistema sólo genera un evento de salida cuando cambia el valor de la señal de salida.

Una forma de modelar este comportamiento es definir tres variables de estado, que almacenen el valor de las entradas. Llamaremos s_{X0}, s_{X1} y s_S a estas tres variables de estado, cada una de las cuales puede tomar los valores $\{0, 1\}$.

Aunque esto no es necesario, definiremos otra variable de estado, s_Y, que almacene el último valor del evento de salida. De esta forma, cuando se produzca un evento de entrada, puede compararse el nuevo valor de la salida con el valor de s_Y, generándose un evento de salida sólo en el caso de que ambos valores sean diferentes.

Así pues, el estado del sistema está compuesto por las variables de estado siguientes: $(fase, \sigma, s_{X0}, s_{X1}, s_S, s_Y)$.

En el enunciado se indica que el sistema tiene 3 entradas (X_0, X_1, S) y una salida (Y). Llamaremos a estos puertos: p_{X0}, p_{X1}, p_S y p_Y.

Para simplificar la descripción del sistema, interpretaremos los valores $\{0, 1\}$ como ’0’ lógico y ’1’ lógico, o equivalentemente, como $false$ y $true$, respectivamente. Llamaremos v_{X0}, v_{X1}, v_S y v_Y al valor de los eventos producidos en cada uno de los puertos. El comportamiento del sistema puede describirse de la manera siguiente:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.50}$$

donde:

$$\begin{array}{lcl} X & = & \{(p_{X0}, \{0,1\}), (p_{X1}, \{0,1\}), (p_S, \{0,1\})\} \\ Y & = & \{(p_Y, \{0,1\})\} \\ S & = & \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \end{array} \tag{3.51}$$

$$\begin{array}{lcl} \delta_{ext}\,(S, e, (p, v)) & = & \begin{cases} (\text{“activo”}, 0, v_{X0}, s_{X1}, s_S, (v_{X0} \cdot \bar{s_S} + s_{X1} \cdot s_S)) \\ \qquad \text{si } (p = p_{X0}) \text{ and } (v_{X0} \cdot \bar{s_S} + s_{X1} \cdot s_S \neq s_Y) \\ (\text{“activo”}, 0, s_{X0}, v_{X1}, s_S, (s_{X0} \cdot \bar{s_S} + v_{X1} \cdot s_S)) \\ \qquad \text{si } (p = p_{X1}) \text{ and } (s_{X0} \cdot \bar{s_S} + v_{X1} \cdot s_S \neq s_Y) \\ (\text{“activo”}, 0, s_{X0}, s_{X1}, v_S, (s_{X0} \cdot \bar{v_S} + s_{X1} \cdot v_S)) \\ \qquad \text{si } (p = p_S) \text{ and } (s_{X0} \cdot \bar{v_S} + s_{X1} \cdot v_S \neq s_Y) \\ (\text{“pasivo”}, \sigma - e, s_{X0}, s_{X1}, s_S, s_Y) \\ \qquad \text{en cualquier otro caso} \end{cases} \\ \delta_{int}\,(S) & = & (\text{“pasivo”}, \infty, s_{X0}, s_{X1}, s_S, s_Y) \\ \lambda\,(S) & = & s_Y \\ ta\,(S) & = & \sigma \end{array} \tag{3.52}$$

Solución al Ejercicio 3.5

El comportamiento del sistema puede describirse de la forma siguiente. Una de las variables de estado del sistema (q) es una cola donde se guarda ordenadamente la información de las entidades que esperan en el sistema. La información de cada entidad es el número que la identifica y el instante de tiempo en que dicha entidad abandonará el sistema. Las entidades están dispuestas en la cola siguiendo el orden en que deberán abandonar el sistema, siendo la primera entidad de la cola la primera en abandonar el sistema.

Cuando llega una nueva entidad, la función de transición externa inserta la nueva entidad en la posición correspondiente de la cola.

La transición interna debe producirse en el instante en que la primera entidad de la cola debe abandonar el sistema. La función de salida genera un evento cuyo valor

coincide con el número que identifica a la primera entidad de la cola y la función de transición interna extrae la primera entidad de la cola.

Una vez descrito a grandes rasgos el comportamiento del modelo, veamos cómo puede definirse cada uno de los elementos de la tupla del modelo DEVS clásico de este sistema SISO:

$$\text{DEVS} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.53}$$

El conjunto de valores posibles de los **eventos de entrada y de salida** son los números reales:

$$X = Y = \mathbb{R} \tag{3.54}$$

El **estado del sistema** puede ser descrito mediante las variables de estado siguientes:

- *fase* puede tomar dos valores: "activo" y "pasivo". El sistema se encuentra en la fase "activo" cuando hay alguna entidad esperando dentro de él. Se encuentra en la fase "pasivo" cuando no hay ninguna entidad esperando dentro de él.
- σ puede tomar valores reales incluido el cero ($\mathbb{R}_0^+$). Su valor representa el tiempo durante el cual el sistema permanecerá en el estado actual en ausencia de eventos externos.
- $q = \{(t_1, x_1), \ldots, (t_n, x_n)\}$ es la cola de entidades que esperan en el sistema. La cola se encuentra ordenada de modo que se satisface $t_1 \leq t_2 \leq \cdots \leq t_n$. Cada pareja de valores corresponde con una entidad que está esperando en el sistema. En el caso de la entidad i-ésima de la cola, el valor t_i es el instante de tiempo en el cual la entidad debe abandonar el sistema y el valor x_i es el número que identifica a la entidad de forma unívoca.
- t_{last} almacena el instante de tiempo en el cual se produjo el último evento. El valor inicial de esta variable es cero. El conjunto de valores que puede tomar es $\mathbb{R}_0^+$. Esta variable de estado se emplea para calcular el valor del reloj de la simulación en los instantes en que se producen las transiciones internas y externas.

La **función de transición externa**, cada vez que es invocada, obtiene una observación independiente de la distribución de probabilidad U(1,2), que llamaremos τ,

y que representa el tiempo durante el cual la entidad que acaba de llegar permanecerá en el sistema. El instante de tiempo en el cual la entidad recién llegada abandonará el sistema se obtiene sumando τ al valor que tenga el reloj de la simulación en el instante en que se produce la llegada. Este valor es $t_{last} + e$. Es decir, la suma del instante de tiempo en que se produjo el último evento (t_{last}) más el tiempo transcurrido (e) desde ese instante hasta el instante actual.

El cambio en el estado producido por la función de transición externa puede interpretarse de la forma siguiente:

- Si cuando se produce el evento externo *fase* vale "pasivo", significa que no hay ninguna entidad esperando en el sistema y, por tanto, q no tiene ningún elemento ($q = \emptyset$). La variable de estado *fase* pasará a valer "activo", ya que la entidad recién llegada se quedará esperando en el sistema.

  La variable de estado σ pasará a valer τ.

  Se añade la entidad recién llegada a la cola q, que estaba vacía. El nuevo valor de la cola, que contiene un único elemento, es: $q = \{(t_{last} + e + \tau, x)\}$.

- Si cuando se produce el evento externo *fase* vale "activo", significa que hay $n \geq 1$ entidades esperando en el sistema. Con la llegada de una nueva entidad, *fase* sigue valiendo "activo".

  El nuevo valor de la cola se obtiene insertando $(t_{last} + e + \tau, x)$ en la posición correspondiente de la cola, de modo que las parejas queden ordenadas por orden creciente de instante en el cual deben abandonar el sistema. Llamando I a la función que inserta la entidad recién llegada en la cola, el nuevo valor de la cola es: $q = I((t_{last} + e + \tau, x), q)$.

  El tiempo restante σ hasta el instante en que se produce la próxima transición interna es el menor de dos valores: $(\sigma - e)$ y τ. Es decir, se comprueba cuál de las dos entidades siguientes abandona antes el sistema: la entidad que se encontraba primera en la cola o la entidad que acaba de llegar.

En ambos casos, el nuevo valor de t_{last} es $t_{last} + e$. Es decir, el nuevo valor es igual a su valor anterior más el tiempo transcurrido desde que se produjo el anterior evento (e).

Formalmente, la función de transición externa puede escribirse de la forma siguiente:

$$\delta_{ext} = \begin{cases} (\text{“activo”}, \tau, \{(t_{last} + e + \tau, x)\}, t_{last} + e) & \text{si } fase = \text{“pasivo”} \\ (\text{“activo”}, \text{mín}(\sigma - e, \tau), I((t_{last} + e + \tau, x), q), t_{last} + e) & \text{si } fase = \text{“activo”} \end{cases} \tag{3.55}$$

La **función de transición interna** describe el cambio en el estado del sistema cuando la entidad que está en la primera posición de la cola abandona el sistema. El nuevo estado del sistema depende del número de entidades que se encuentren en la cola una vez extraída de la cola la primera entidad. Si la cola se representa como $q = (t_1, x_1) \cdot q^*$, donde (t_1, x_1) es el primer elemento de la cola y q^* es la cola una vez extraído el primer elemento, entonces pueden distinguirse la dos situaciones siguientes:

- Si q sólo tiene una entidad, entonces q^* es el conjunto vacío. En este caso, el nuevo estado será: $fase$=“pasivo”, $\sigma = \infty$, $q = \emptyset$, $t_{last} = t_{last} + e$.
- Si q tiene más de una entidad, entonces $q^* = \{(t_2, x_2), \cdots\}$. El nuevo estado del sistema será: $fase$=“activo”, $\sigma = t_2 - (t_{last} + e)$, $q = q^*$, $t_{last} = t_{last} + e$.

El nuevo estado puede representarse de la forma siguiente:

$$\delta_{int}(fase, \sigma, (t_1, x_1) \cdot q^*, t_{last}) = \begin{cases} (\text{“pasivo”}, \infty, q^*, t_{last} + e) & \text{si } q^* = \emptyset \\ (\text{“activo”}, t_2 - (t_{last} + e), q^*, t_{last} + e) & \text{si } q^* \neq \emptyset \end{cases} \tag{3.56}$$

La **función de salida** devuelve el valor x_1. Es decir, el número que identifica a la primera entidad de la cola.

$$\lambda(fase, \sigma, q = \{(t_1, x_1), \cdots\}, t_{last}) = x_1 \tag{3.57}$$

La **función de avance en el tiempo** vale: $t_a = \sigma$.

Solución al Ejercicio 3.6

Un modelo DEVS multicomponente es la tupla siguiente:

$$\text{multiDEVS} = \langle X, Y, D, \{M_d\}, Select \rangle \tag{3.58}$$

Los eventos de entrada pueden tomar cualquier valor real: $X = \mathbb{R}$. La llegada del evento de entrada indica el instante en el que se produce un cambio en la velocidad del objeto y el valor del evento representa este nuevo valor de la velocidad.

En el enunciado no se indica que el sistema genere ningún evento de salida, por tanto: $Y = \emptyset$.

Si el sistema está formado por N componentes, estos se referenciarán numerándolos consecutivamente: $D = \{1, \ldots, N\}$. Cada componente, d, tiene un componente situado a la izquierda y otro a la derecha. Dado que la primera y la última células están unidas entre sí, la célula N está situada a la izquierda de la célula 1.

Puesto que el objeto sólo puede estar en una de las células y los eventos internos se producen cuando el objeto pasa de una célula a la contigua, en este sistema no es posible que varios componentes tengan planificadas transiciones internas para el mismo instante, por ello no es preciso definir la función *Select*.

Cada uno de los componentes, $d \in D$, se define de la forma siguiente:

$$M_d = \langle S_d, I_d, E_d, \delta_{ext,d}, \delta_{int,d}, \lambda_d, ta_d \rangle \tag{3.59}$$

El **estado** del componente d, S_d, puede representarse mediante las cuatro variables de estado siguientes: $(fase, \sigma, posicion, velocidad)$.

- La variable *fase* puede tomar dos valores, {*"ocupado"*, *"libre"*}, dependiendo de que el objeto se encuentre o no en el componente.
- La variable σ almacena el tiempo hasta la siguiente transición interna en ausencia de eventos externos.
- Las variables *posición* y *velocidad* almacenan la posición y velocidad del objeto en el instante en que se produjo el último evento. Si el objeto no se encuentra en el componente, tienen el valor cero. Llamando L a la longitud de una de las células (en este caso, $L = 1$ m), la variable *posicion* puede tomar valores entre 0 y L. El valor 0 representa el extremo izquierdo de la célula y el valor L el extremo derecho.

Cada célula influye sobre sus células adyacentes y es influenciada por éstas.

Supongamos que el objeto se encuentra en la célula i. En el instante en que el objeto sale de la célula i, se produce un evento interno. La **función de transición interna** de la célula i modifica el estado de la célula i y de una de las adyacentes

(aquella en la cual entra el objeto). El nuevo estado de la célula i es: (*fase* = *"libre"*, ∞, 0, 0). Además:

- Si *velocidad* >0, entonces el objeto entra en la célula situada a la derecha de i. El nuevo estado de la célula situada a la derecha de i es: (*fase* = *"ocupado"*, $L/velocidad$, 0, velocidad). Obsérvese que $L/velocidad$ es el tiempo que tardará el objeto en atravesar la célula si no hay cambios en su velocidad (es decir, en ausencia de eventos externos).
- Si *velocidad* <0, entonces el objeto entra en la célula situada a la izquierda de i. El nuevo estado de la célula situada a la izquierda de i es: (*fase* = *"ocupado"*, $-L/velocidad$, L, velocidad). El signo menos en $-L/velocidad$ es debido a que en este caso la velocidad tiene signo negativo. La posición del objeto es L, ya que el objeto entra por la parte derecha de la célula.

Los eventos externos producen cambios en la velocidad del objeto. El cambio en el estado de las células viene definido por la **función de transición externa**. El estado de las células en las que no se encuentra el objeto no se ve alterado por el evento externo. Suponiendo que el objeto se encuentre en la célula i, el nuevo valor del estado de la célula i al producirse un evento de valor x es el siguiente:

- Si $x = 0$: (*"ocupado"*, ∞, $posicion + velocidad \cdot e$, 0)
- Si $x > 0$: (*"ocupado"*, $\frac{L-(posicion+velocidad \cdot e)}{x}$, $posicion + velocidad \cdot e$, x)
- Si $x < 0$: (*"ocupado"*, $-\frac{posicion+velocidad \cdot e}{x}$, $posicion + velocidad \cdot e$, x)

donde e es el tiempo transcurrido desde que se produjo la anterior transición.

La **función de avance en el tiempo** de cada célula devolverá el valor de la variable de estado σ de la célula.

Solución al Ejercicio 3.7

El estado del sistema puede definirse mediante cuatro variables de estado: (*fase*, σ, *entidad*, u).

- La variable *fase* indica si el sistema se encuentra activo (respondiendo a un evento de entrada) o pasivo.

– La variable σ almacena el tiempo que transcurrirá hasta la siguiente transición interna en ausencia de eventos externos.

– La variable *entidad* almacena el número identificativo de la entidad.

– La variable u almacena la observación de la distribución U(0,1) generada en la última transición externa.

El comportamiento del sistema puede describirse de la manera siguiente:

$$\text{DEVS}_p = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{3.60}$$

donde:

$$\begin{array}{lcl} X & = & \{(entrada, \mathbb{R})\} \\ Y & = & \{(salida1, \mathbb{R}), (salida2, \mathbb{R})\} \\ S & = & \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \mathbb{R} \times [0,1] \\ \delta_{ext}\,(S, e, (entrada, x)) & = & (\text{“activo”}, 0, x, \text{genera observación U(0,1)}) \\ \delta_{int}\,(S) & = & (\text{“pasivo”}, \infty, entidad, u) \\ \lambda\,(S) & = & \begin{cases} (salida1, entidad) & \text{si } u < p \\ (salida2, entidad) & \text{en caso contrario} \end{cases} \\ ta\,(S) & = & \sigma \end{array} \tag{3.61}$$

Solución al Ejercicio 3.8

La especificación del modelo DEVS compuesto mostrado en la Figura 3.17 es la siguiente:

$$N = \langle\ X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC,\ select\ \rangle \tag{3.62}$$

donde:

$X = \{(\text{“in”}, \mathbb{R})\}$	Puerto de entrada del modelo compuesto y posibles valores de los eventos.
$Y = \{\ (\text{“out1”}, \mathbb{R}), (\text{“out2”}, \mathbb{R})\}$	Puerto de salida del modelo compuesto y posibles valores de los eventos.

$D = \{P_0, D, P_1, P_2\}$	Conjunto de nombres de los componentes.
$M_{P_0}, M_{P_1}, M_{P_2}, D$	Cada componente es un modelo DEVS.
$\text{EIC} = \{\ ((N, \text{“in”}), (P_0, \text{“in”}))\ \}$	Conexión del puerto de entrada del modelo compuesto, N, al puerto de entrada del componente P_0.
$\text{EOC} = \left\{ \begin{array}{l} ((P_1, \text{“out”}), (N, \text{“out1”}))\,, \\ ((P_2, \text{“out”}), (N, \text{“out2”})) \end{array} \right\}$	Conexión del puerto de salida de los componentes P_1 y P_2 a los puertos de salida del modelo compuesto.
$\text{IC} = \left\{ \begin{array}{l} ((P_0, \text{“out”}), (D, \text{“in”}))\,, \\ ((D, \text{“out1”}), (P_1, \text{“in”}))\,, \\ ((D, \text{“out2”}), (P_2, \text{“in”})) \end{array} \right\}$	Conexiones internas.
$select()$	Debería establecerse atendiendo al funcionamiento del sistema, que no se especifica en el enunciado.

Solución al Ejercicio 3.9

Los eventos en el puerto de entrada D, y en los puertos de salida Q y $notQ$ pueden tomar los valores {0,1}. Los eventos en el puerto de entrada Ck sólo pueden tomar un valor: {1}. Al realizar el modelo, suponemos que en este sistema no pueden producirse dos eventos de entrada simultáneos.

El funcionamiento del sistema es el siguiente. Cada vez que se produce un evento de entrada en el puerto Ck, se producen dos eventos de salida:

- Un evento en el puerto Q, cuyo valor es igual al del último evento recibido en el puerto de entrada D.
- Un evento en el puerto $notQ$, cuyo valor es igual al complementario del valor del último evento recibido en el puerto de entrada D.

El modelo debe almacenar el valor del último evento recibido en el puerto de entrada D. Este valor se almacena en la variable de estado s_D, que puede tomar los valores {0,1}. El estado del sistema es: $(fase, \sigma, s_D)$. La variable de estado $fase$ puede tomar dos valores: {“pasivo”, “respuesta”}.

El comportamiento del sistema puede describirse de la manera siguiente:

$$\begin{array}{lcl} X & = & \{(D,\{0,1\}),(Ck,\{1\})\} \\ Y & = & \{(Q,\{0,1\}),(notQ,\{0,1\})\} \\ S & = & \{\text{“pasivo”},\text{“respuesta”}\}\times\mathbb{R}_0^+\times\{0,1\} \\ \delta_{ext}\left(S,e,(p,v)\right) & = & \begin{cases} (\text{“pasivo”},\sigma-e,v) & \text{si } p=D \\ (\text{“respuesta”},0,s_D) & \text{si } p=Ck \end{cases} \\ \delta_{int}\left(fase,\sigma,s_Y\right) & = & (\text{“pasivo”},\infty,s_D) \\ \lambda\left(fase,\sigma,s_Y\right) & = & \{(Q,\{s_D\}),(notQ,\{\bar{s_D}\})\} \\ ta\left(fase,\sigma,s_Y\right) & = & \sigma \end{array} \tag{3.63}$$

Mediante $\bar{s_D}$, empleado en la definición de la función de salida, se representa el valor complementario del almacenado en s_D.

Tema 4

DEVS PARALELO

OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema debería saber:

- Aplicar el formalismo DEVS paralelo a la descripción de modelos atómicos y compuestos de eventos discretos.
- Discutir las diferencias entre DEVS paralelo y DEVS clásico.

4.1. INTRODUCCIÓN

El formalismo *DEVS paralelo* difiere del formalismo *DEVS clásico* en que DEVS paralelo permite el disparo simultáneo de eventos en diferentes componentes del modelo y la generación simultánea de las correspondientes salidas. Asimismo, el formalismo DEVS paralelo permite la llegada simultánea de varios eventos a un mismo puerto de entrada y la generación simultánea de varios eventos en un mismo puerto de salida. En todos los casos, los componentes receptores de estos eventos son los encargados de examinar el evento de entrada y de procesarlo adecuadamente.

4.2. MODELOS ATÓMICOS

Un modelo atómico DEVS paralelo, que en general tendrá varios puertos de entrada y varios puertos de salida, se define mediante la tupla siguiente:

$$\text{DEVS} = \langle X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle \tag{4.1}$$

donde:

$X_M = \{\ (p, v) \mid p \in InPorts,\ v \in X_p\ \}$	Puertos de entrada y valores.
S	Conjunto de estados secuenciales.
$Y_M = \{\ (p, v) \mid p \in OutPorts,\ v \in Y_p\ \}$	Puertos de salida y valores.
$\delta_{int} : S \to S$	Función de transición interna.
$\delta_{ext} : Q \times X_M^b \to S$	Función de transición externa.
$\delta_{con} : Q \times X_M^b \to S$	Función de transición confluyente.
$\lambda : S \to Y^b$	Función de salida.
$ta : S \to \mathbb{R}_{0,\infty}^+$	Función de avance de tiempo.
$Q = \{\ (s, e) \mid s \in S,\ 0 \leq e \leq ta(s)\ \}$	Estado total.

Obsérvese que el formalismo DEVS paralelo presenta dos diferencias fundamentales respecto al DEVS clásico:

1. DEVS paralelo permite que varios eventos se produzcan simultáneamente en uno o varios puertos de entrada y de salida. Por ello, la función de transición externa y la función de salida de DEVS paralelo son definidas haciendo uso del concepto de "bolsa de eventos". El superíndice b representa la palabra inglesa "bag".

Se denomina *bolsa* a un conjunto de elementos, en el cual es posible que algunos de los elementos estén repetidos. Por ejemplo, $\{a, b, a, c, b, b\}$ es una bolsa. Al igual que sucede en los conjuntos, los elementos de una *bolsa* no están ordenados.

En la definición de la función de transición externa de DEVS paralelo, X_M^b representa una bolsa de parejas (puerto, evento). Al usar *bolsas* para agrupar los eventos de entrada estamos reconociendo el hecho de que las entradas pueden llegar simultáneamente a un mismo puerto de entrada, en un orden indeterminado, o a varios puertos de entrada.

Igualmente, la función de salida λ de DEVS paralelo genera una bolsa de eventos de salida, Y^b. Es decir, el sistema puede generar simultáneamente varios eventos en uno o en varios de sus puertos de salida.

2. Una segunda diferencia respecto a DEVS clásico es que la especificación DEVS paralelo contiene una función adicional: la *función de transición confluyente*, δ_{con}. La finalidad de esta función es definir el nuevo estado del sistema cuando en un mismo instante llega una bolsa de eventos de entrada y está planificada una transición interna del estado.

   Obsérvese que en este caso no se trata de decidir si se ejecuta antes la transición interna o la externa, sino que la función de transición confluyente permite especificar directamente cuál es el nuevo estado cuando se produce la confluencia de una transición interna y una externa.

   Mediante la notación "$\delta_{con} : Q \times X_M^b \to S$" se representa el tipo de argumentos de la función y el tipo de resultado que devuelve. Los argumentos de entrada a la función son el estado total del sistema (Q) y la bolsa de eventos de entrada (X_M^b). La función devuelve el nuevo estado del sistema.

Ejemplo 4.2.1. *En la Figura 4.1 se muestra un ejemplo del comportamiento de un modelo DEVS paralelo con una entrada (X) y una salida (Y). Para simplificar las explicaciones, los eventos llegan al puerto de entrada de uno en uno (x_2 en el instante t_2, x_4 en t_4, etc.), y la función de salida genera cada vez un único evento, que es enviado a través del puerto de salida (y_1 en el instante t_1, y_2 en t_2, etc.)*

En los instantes t_2 y t_5, la llegada de un evento externo coincide con el instante en que está planificada una transición interna. En estos instantes se ejecuta la función de salida, generándose un evento de salida. A continuación, se calcula el nuevo estado, empleando la función de transición confluyente.

□

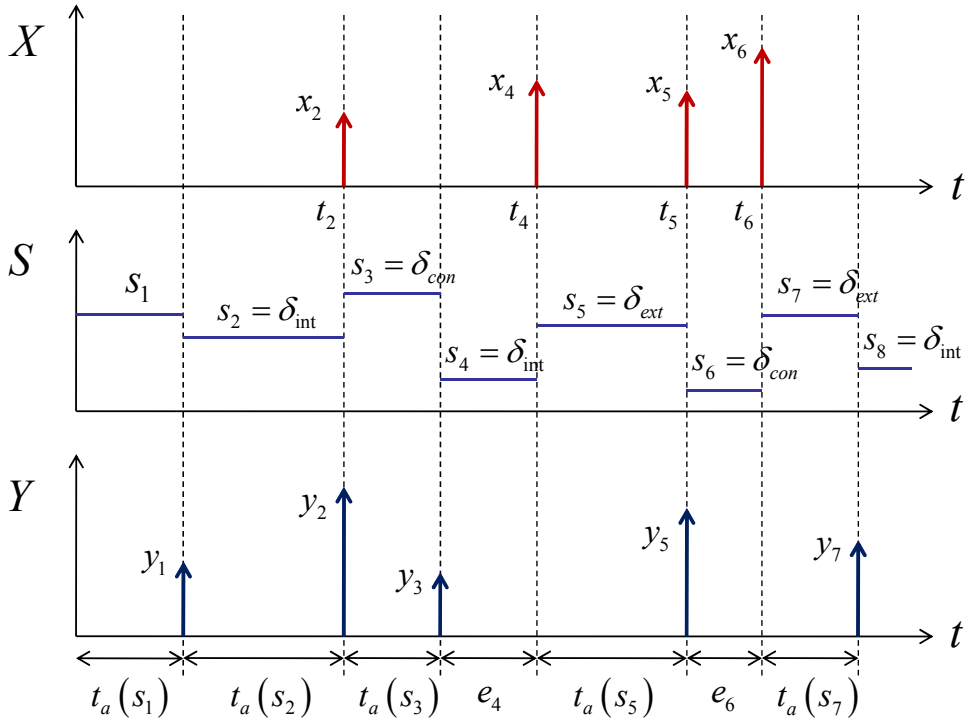

Figura 4.1: Comportamiento de un modelo DEVS paralelo.

4.2.1. Modelo de un proceso con una cola

En la Sección 3.2.5 se describió en modelo DEVS clásico de un proceso con un recurso. En aquel caso, el proceso no disponía de una cola en la cual pudieran esperar las entidades. Consecuentemente, si cuando llegaba una entidad el recurso se encontraba ocupado, dicha entidad era ignorada.

En esta sección complicaremos un poco el modelo descrito en la Sección 3.2.5, suponiendo que el proceso dispone de una cola con disciplina FIFO en la cual pueden esperar las entidades a que llegue su turno. La especificación DEVS paralelo de este sistema es la siguiente:

$$\text{DEVS}_{\Delta} = \langle X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle \tag{4.2}$$

A continuación se describe cada uno de los elementos de la tupla.

Conjuntos de entrada y salida

El modelo tiene un único puerto de entrada, llamado *In*. Los eventos de entrada a través de ese puerto pueden tomar valores de entre un conjunto de valores posibles, que representamos mediante V. Por ejemplo, V podría ser el conjunto de los números reales. El valor de cada evento de entrada corresponde con un atributo o identificador, que caracteriza la entidad que llega al proceso. Así pues:

$$\begin{array}{lcl} X_M & = & \{\ (p,v) \mid p \in InPorts,\ v \in X_p\ \} \\ InPorts & = & \{\text{“}In\text{”}\} \\ X_{in} & = & V \end{array} \tag{4.3}$$

El modelo tiene un único puerto de salida, llamado *Out*. El conjunto de valores que pueden tomar los eventos de salida es también V. Así pues:

$$\begin{array}{lcl} Y_M & = & \{\ (p,v) \mid p \in OutPorts,\ v \in Y_p\ \} \\ OutPorts & = & \{\text{“}Out\text{”}\} \\ Y_{out} & = & V \end{array} \tag{4.4}$$

Estado secuencial

El estado del modelo, S, está definido mediante las tres variables de estado siguientes:

- La variable *fase*, que puede valer “pasivo” o “activo”.
- La variable σ, que almacena el tiempo hasta el instante en que está planificada la siguiente transición interna. Puede tomar valores reales positivos, incluyendo el cero: $\mathbb{R}_0^+$.
- La variable q, que almacena la información sobre la secuencia de entidades que se encuentran en cola y la entidad que se encuentra en proceso. Puesto que cada entidad está caracterizada por un valor perteneciente al conjunto V, la variable q almacena una secuencia finita de valores pertenecientes al conjunto V. Dicha secuencia finita de valores se representa mediante V^+.

En consecuencia, el conjunto de los posibles valores del estado secuencial del sistema es el producto cartesiano de los posibles valores de estas tres variables:

$$S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times V^+ \tag{4.5}$$

Consecuentemente, el estado del sistema queda definido especificando el valor de las tres variables de estado: $(fase, \sigma, q)$.

Función de transición externa

En el modelo DEVS paralelo pueden producirse varios eventos de entrada simultáneamente en un puerto, lo que equivaldría a la llegada simultánea de varias entidades al proceso. Cuando se produce esta situación, el modelo debe situar las entidades ordenadamente en la cola y, en caso de que el proceso se encuentre en la fase “pasivo”, comenzar a trabajar con la entidad que se encuentre situada en primer lugar. Obsérvese que las *bolsas* son conjuntos no ordenados de elementos. Por tanto, en la bolsa de eventos de entrada los elementos (eventos de entrada) no están ordenados.

La función de transición externa es la siguiente:

$$\begin{array}{l}\delta_{ext}\left(fase, \sigma, q, e, ((\text{“}In\text{”}, x1), (\text{“}In\text{”}, x2), \ldots, (\text{“}In\text{”}, xn))\right) = \\ \left\{\begin{array}{ll}(\text{“activo”}, \Delta, \{x1, x2, \ldots, xn\}) & \text{si } fase = \text{“pasivo”} \\ (\text{“activo”}, \sigma - e, q \bullet \{x1, x2, \ldots, xn\}) & \text{si } fase = \text{“activo”}\end{array}\right.\end{array} \tag{4.6}$$

Obsérvese que los argumentos de entrada de la función son el estado total del sistema, $\{fase, \sigma, q, e\}$, y la bolsa de los eventos de entrada, que está compuesta por pares (puerto, evento de entrada). En este caso, hay un único puerto de entrada, *“In”*, al cual se supone que llegan n eventos simultáneamente, de valor $x1$, …, xn. En consecuencia, la bolsa de eventos de entrada se representa:

$$X_M^b = ((\text{“}In\text{”}, x1), (\text{“}In\text{”}, x2), \ldots, (\text{“}In\text{”}, xn)) \tag{4.7}$$

El nuevo estado tras la transición externa depende del valor que tuviera *fase* en el instante de llegada de los n eventos externos simultáneos:

– Si el proceso se encuentra en la fase “pasivo”, entonces no hay ninguna entidad en q, ya que q almacena la entidad en proceso y las entidades en cola.

  En la Figura 4.2a se representa el sistema en la fase “pasivo”. La circunferencia representa el recurso. El elemento de q que está dentro de la circunferencia

representa la entidad que ha capturado el recurso, es decir, la entidad que está siendo procesada.

Al llegar las n nuevas entidades, éstas se almacenan en q. El nuevo valor de q es entonces $\{x1, x2, \ldots, xn\}$. Se supone que el primer elemento de q (en este caso, $x1$) es el que captura el recurso, el cual pasa a la fase "activo" (véase la Figura 4.2b). Se asigna el valor Δ a la variable σ, que almacena el tiempo hasta la próxima transición interna en ausencia de eventos externos.

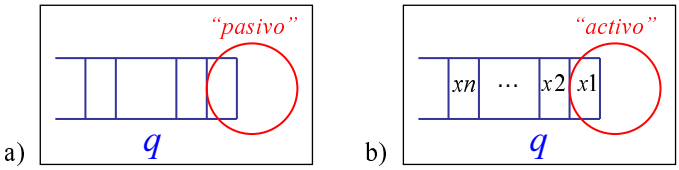

Figura 4.2: Entidades en el proceso: a) antes; y b) después de la llegada de los eventos $x1, \ldots, xn$.

– Si el proceso se encuentra en la fase "activo", entonces q contiene al menos un elemento: la entidad que tiene el recurso capturado. Además, contendrá las entidades que se encuentran esperando en la cola.

En la Figura 4.3a se muestra un ejemplo de esta situación, en el cual q contiene los elementos $\{y1, \ldots, ym\}$, y la entidad $y1$ tiene el recurso capturado.

Las nuevas entidades, $\{x1, x2, \ldots, xn\}$, deben añadirse al final de la cola (véase la Figura 4.3b). Esta concatenación de los nuevos elementos tras los ya existentes en q se representa: $q \bullet \{x1, x2, \ldots, xn\}$. El símbolo $\bullet$ representa la concatenación.

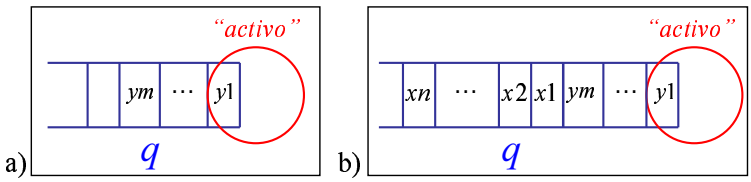

Figura 4.3: Entidades en el proceso: a) antes; y b) después de la llegada de los eventos $x1, \ldots, xn$.

Función de transición interna

La *función de transición interna* es invocada cuando se termina de procesar una entidad. Se define de la forma siguiente:

$$\delta_{int}\left(\text{“activo”}, \sigma, v \bullet q^*\right) = \begin{cases} (\text{“pasivo”}, \infty, \emptyset) & \text{si } q^* = \emptyset \\ (\text{“activo”}, \Delta, q^*) & \text{si } q^* \neq \emptyset \end{cases} \tag{4.8}$$

donde se supone que la variable de estado q, que se pasa como argumento a la función δ_{int}, está compuesta por la concatenación del elemento v, que es aquel cuyo proceso acaba de finalizar, y de la cola q^*. Es decir, $q = v \bullet q^*$.

El estado del sistema tras la transición interna depende de si q^* contiene o no elementos, es decir, de si hay o no otras entidades en el proceso aparte de la que tiene el recurso capturado (que es v).

- Si q^* no contiene ningún elemento, significa que el proceso sólo contiene una entidad: aquella que acaba de terminar de ser procesada. Esta entidad abandona el proceso, con lo cual el proceso queda vacío de entidades: $q = \emptyset$. El recurso queda tras la transición interna en la fase “pasivo” y el tiempo hasta la siguiente transición interna, en ausencia de eventos externos, es infinito: $\sigma = \infty$. En este caso, el nuevo estado es: $(\text{“pasivo”}, \infty, \emptyset)$.
- Si q^* contiene algún elemento, significa que hay entidades en la cola del proceso esperando a que el recurso quede libre para capturarlo. En este caso, la entidad que tenía el recurso capturado lo libera y abandona el proceso: q pasa de valer $v \bullet q^*$ a valer q^*, es decir, la entidad v abandona el proceso. La primera entidad de la cola captura el recurso, que sigue en la fase “activo”. El tiempo hasta la siguiente transición interna, en ausencia de eventos externos, es igual al tiempo de proceso: $\sigma = \Delta$. En consecuencia, el nuevo estado es: $(\text{“activo”}, \Delta, q^*)$.

Función de transición confluyente

La *función de transición confluyente* es la siguiente:

$$\delta_{con}\left(s, x^b\right) = \delta_{ext}\left(\delta_{int}(s), 0, x^b\right) \tag{4.9}$$

la cual indica que en caso de que llegue una bolsa de eventos externos en el preciso instante en que está planificada una transición interna, entonces en primer lugar se ejecutará el evento interno y a continuación se ejecutarán los eventos externos. Para evaluar la función δ_{con} se procede de la forma siguiente:

1. En primer lugar, se calcula el estado resultante del evento interno, que viene dado por $\delta_{int}(s)$. Este estado es transitorio, ya que sobre él se produce en ese mismo instante la transición externa.
2. El estado resultante del evento externo se calcula pasando como parámetros a la función δ_{ext}:
   - El estado transitorio calculado anteriormente, $\delta_{int}(s)$.
   - El tiempo desde la anterior transición, que es igual a cero, ya que acaba de producirse la transición interna.
   - La bolsa de eventos externos, x^b.

Otra posibilidad sería que se ejecutara primero el evento externo y a continuación el interno. En este caso, la función de transición confluyente sería:

$$\delta_{con}\left(s, ta(s), x^b\right) = \delta_{int}\left(\delta_{ext}\left(s, ta(s), x^b\right)\right) \tag{4.10}$$

En general, no es necesario expresar δ_{con} en términos de las funciones de transición interna o externa. Puede definirse δ_{con} independientemente de aquellas dos funciones, de modo que exprese circunstancias especiales que ocurren cuando están planificadas para el mismo instante de tiempo una transición interna y una externa.

Función de salida

La *función de salida* es la siguiente:

$$\lambda(\text{“activo”}, \sigma, v \bullet q^*) = v \tag{4.11}$$

donde nuevamente se supone que la variable de estado q está compuesta por la concatenación del elemento v, que es aquel cuyo proceso acaba de finalizar, y de la cola q^*. Es decir, $q = v \bullet q^*$. El valor del evento de salida es v. Todo esto es equivalente a decir que el valor del evento de salida es igual al primer componente almacenado en q.

Función de avance del tiempo

La *función de avance del tiempo* es la siguiente:

$$ta\left(fase, \sigma, q\right) = \sigma \tag{4.12}$$

4.3. MODELOS COMPUESTOS

Los modelos acoplados DEVS paralelo se especifican de la misma forma que los modelos DEVS clásico, con la diferencia de que en el formalismo DEVS paralelo la función *select* se omite. Recuérdese que en el formalismo DEVS clásico la función *select* establece la prioridad de ejecución de los eventos cuando varios componentes tienen planificados eventos internos en un mismo instante de tiempo.

En DEVS clásico, cuando varios eventos están planificados para el mismo instante, se escoge uno de ellos para su ejecución. En contraste, en DEVS paralelo no se decide qué evento ejecutar en primer lugar, sino que todos los eventos internos planificados para un mismo instante son ejecutados simultáneamente.

Para poder entender correctamente el tratamiento de los eventos, recuérdese que la ejecución de un evento interno, suponiendo que el sistema se encuentra en el estado s_1, consta de los tres pasos descritos a continuación:

Paso 1 Se asigna el tiempo transcurrido desde la última transición: $e = ta(s_1)$.

Paso 2 Se genera un evento de salida, cuyo valor viene dado por $\lambda(s_1)$.

Paso 3 Se realiza la transición interna del estado. El estado del sistema pasa de ser s_1 a ser $s_2 = \delta_{int}(s_1)$.

Teniendo presente lo anterior, supongamos que los eventos internos de varios componentes están planificados para un mismo instante de tiempo. El procedimiento de ejecución en DEVS paralelo es el siguiente:

1. En cada componente que tiene planificado un evento interno, se realiza el Paso 1 y el Paso 2 del tratamiento del evento. En particular, en el Paso 2 se genera el correspondiente evento de salida. Por el momento no se ejecuta el Paso 3, que es la transición interna del estado, sino que se mantiene la planificación, para este mismo instante, de la transición interna.

2. Estos eventos de salida se propagan a través de las conexiones entre componentes, convirtiéndose en eventos de entrada.

3. En cada componente al que llega un evento de entrada, se planifica una transición externa del estado para ese mismo instante.

4. En los componentes que tienen una única transición planificada, ya sea interna o externa, no existe ambigüedad respecto al orden en el cual realizar las transiciones, ya que sólo hay una. En los componentes en los cuales hay planificada una transición interna del estado y una externa, se usa la *función de transición confluyente* del componente para decidir cuál de ellos se realiza antes.

Ejemplo 4.3.1. *Para ilustrar la diferencia entre el tratamiento de los eventos realizado en DEVS clásico y en DEVS paralelo, vamos a retomar el Ejemplo 3.4.1. En dicho ejemplo se analizaba, desde la perspectiva de DEVS clásico, el establecimiento de prioridades para la ejecución de los eventos cuando hay varios eventos internos planificados para un mismo instante. El sistema bajo estudio era la conexión en cadena de tres procesos mostrada en la Figura 4.4.*

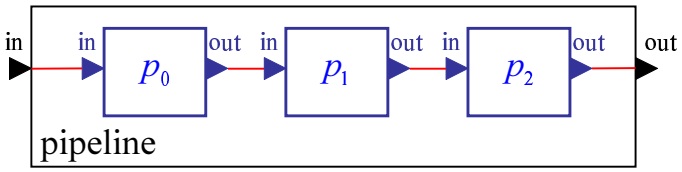

Figura 4.4: Pipeline compuesta por tres etapas, descrita en la Sección 3.4.1.

Supongamos que los componentes p_0 y p_1 tienen planificados sendos eventos internos para un mismo instante de tiempo. El evento interno de p_0 genera un evento de salida de p_0, que es un evento de entrada para p_1. En DEVS clásico, el diseñador del modelo debe decidir qué hacer respecto al orden en que se disparan estos dos eventos internos:

1. *Si se ejecuta primero el evento interno de p_1, el componente p_1 pasa de "activo" a "pasivo", con lo cual, cuando a continuación se ejecute el evento externo, p_1 está en disposición de aceptar la entidad proveniente de p_0.*

2. *Si se dispara primero el evento interno de p_0 y el evento externo de p_1, la entidad que llega a p_1 encuentra que éste está en la fase "activo", con lo cual el evento de entrada es ignorado. A continuación, se dispara el evento interno*

de p_1, pasando este componente de "activo" a "pasivo" y quedando, por tanto, en la fase "pasivo".

En DEVS clásico, aplicando la función select se determina el orden en que deben ejecutarse los eventos internos simultáneos. En el Ejemplo 3.4.1 la función select establecía que si hay dos eventos planificados para el mismo instante en los componentes p_0 y p_1, entonces el evento asociado al componente de mayor índice, que en este caso es p_1, se dispara en primer lugar. Así pues, se verificaría la primera de las dos opciones anteriores.

Obsérvese que en DEVS clásico, cuando varios eventos están planificados para el mismo instante, sólo uno de ellos es escogido para su ejecución. En contraste, en DEVS paralelo todos los eventos internos planificados para un mismo instante son ejecutados simultáneamente:

1. *Se generan los eventos de salida de los componentes p_0 y p_1.*
2. *El evento de salida de p_0 es un evento de entrada de p_1.*
3. *El componente p_0 tiene planificada una transición interna del estado, que puede realizarse sin ambigüedades. El componente p_1 tiene planificada una transición interna y una externa. Se usa la función de transición confluyente del componente p_1 para decidir cuál de las dos transiciones del estado se ejecuta en primer lugar. Si esta función realiza primero la transición interna y a continuación la externa, entonces en primer lugar la entidad cuyo proceso ha finalizado libera el recurso, el cual queda en fase "pasivo" y puede ser capturado por la nueva entidad.*

□

4.4. EJERCICIOS DE AUTOCOMPROBACIÓN

Ejercicio 4.1

Modifique el modelo DEVS paralelo del proceso con una cola FIFO, de modo que la cola tenga una capacidad máxima, N_{max}. Mientras la cola tiene su tamaño máximo, las entidades que llegan no se añaden al final de la cola, sino que abandonan inmediatamente el sistema.

Ejercicio 4.2

En la Sección 3.2.2 se describió el modelo DEVS clásico de un sistema acumulador SISO. Ahora se propone modificar este modelo, de manera que tenga dos puertos de entrada, llamados "Input" y "Read", a través de los cuales el sistema acepta los valores a almacenar y las solicitudes de generación de eventos, respectivamente. Las salidas se producen a través del puerto llamado "Output".

Describa el modelo empleando el formalismo DEVS paralelo. Debe tenerse en cuenta la posibilidad de que el sistema reciba eventos simultáneos en uno o en los dos puertos de entrada. Escoja el criterio que desee para resolver la colisión entre las transiciones internas y externas.

Ejercicio 4.3

En la Sección 3.2.3 se describió un sistema generador de eventos. Suponga que este sistema se modifica, de manera que tenga dos puertos de entrada: uno para iniciar la generación (puerto "Start") y otro para suspenderla (puerto "Stop"). Escriba el modelo DEVS paralelo de este sistema modificado. Tome las decisiones de diseño que considere convenientes, justificando los motivos por los cuáles las ha adoptado.

Ejercicio 4.4

En la Figura 4.5 se muestra una red de conmutación. Uno de los componentes de la red de conmutación, llamado s_0, es un conmutador. Escriba un modelo DEVS paralelo del conmutador, de modo que pueda aceptar eventos simultáneos en am-

bos puertos de entrada. Asimismo, describa un modelo DEVS paralelo del modelo compuesto de la red de conmutación.

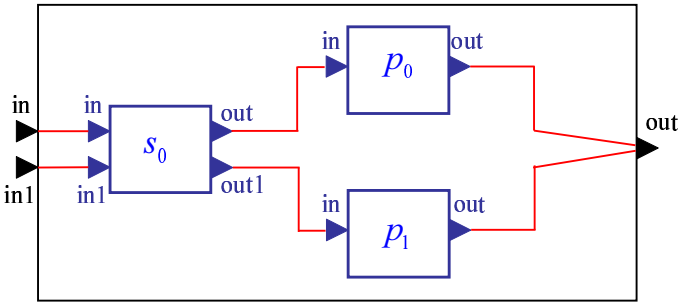

Figura 4.5: Red de conmutación.

4.5. SOLUCIONES A LOS EJERCICIOS

Solución al Ejercicio 4.1

El número de entidades que se encuentran en el sistema es igual al número de entidades que almacena la variable q. Llamaremos $length(q)$ a dicho número, que es la suma de la entidad que está en proceso más las entidades que esperan en la cola.

El valor $length(q)$ no puede superar el valor N_{max}. Supongamos que en un determinado instante de tiempo se produce un evento en el puerto "*In*", en el cual se produce la llegada de las entidades $x_1, \cdots, x_n$. Hay dos posibilidades:

1. Si $length(q)+n \leq N_{max}$, entonces todas las entidades que llegan son admitidas en el sistema y el número de entidades pasaría a ser $length(q) + n$.
2. Si $length(q)+n > N_{max}$, entonces de las n entidades que han llegado, $N_{max} - length(q)$ son admitidas en el sistema y las restantes $length(q) + n - N_{max}$ abandonan inmediatamente el sistema.

En determinadas aplicaciones es necesario distinguir entre las entidades que abandonan el sistema nada más llegar y las que abandonan el sistema tras ser procesadas normalmente. Por ello, definimos dos puertos de salida (véase la Figura 4.6):

– "*Out*": entidades que abandonan el sistema tras ser procesadas.

– "*Balk*": entidades que no pueden ser aceptadas en el sistema por haberse superado la capacidad máxima del mismo.

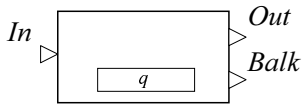

Figura 4.6: Diagrama del modelo.

Los **puertos de entrada y salida, y los posibles valores de los eventos** en estos puertos son los siguientes:

$$\begin{array}{rcl} X_M & = & \{\ (\text{“}In\text{”}, \mathbb{R})\} \\ Y_M & = & \{\ (\text{“}Out\text{”}, \mathbb{R}), (\text{“}Balk\text{”}, \mathbb{R})\ \} \end{array} \tag{4.13}$$

El modelo tiene cinco variables de estado. Tres de ellas son las ya descritas en la versión anterior del modelo: $fase$, σ, y la variable q, que almacena las entidades que se encuentran en el sistema. En este caso, la variable de estado $fase$ puede tomar tres valores: "*pasivo*", "*activo*" y "*balking*".

Las otras dos variables de estado son las siguientes:

- La variable de estado τ_p almacena el tiempo que debe transcurrir hasta que termine de ser procesada la entidad que se encuentra en proceso.
- La variable de estado $q_{balking}$ almacena las entidades que deben abandonar el sistema por haberse superado su capacidad máxima.

La utilidad de definir la $fase$ "*balking*", y las variables de estado τ_p y $q_{balking}$ se explicará al describir la función de transición externa. Las cinco variables de estado son $(fase, \sigma, q, \tau_p, q_{balking})$. El **conjunto de estados secuenciales** es:

$$\begin{aligned} S &= \{\text{“}pasivo\text{”}, \text{“}activo\text{”}, \text{“}balking\text{”}\} \times \mathbb{R}_0^+ \times \{\text{secuencia finita de valores } \mathbb{R}\} \\ &\quad \times \mathbb{R}_0^+ \times \{\text{secuencia finita de valores } \mathbb{R}\} \end{aligned}$$

Cuando se produce un evento de entrada, en el cual llegan n entidades al sistema $(x_1, \cdots, x_n)$, puede darse una de las cuatro situaciones siguientes:

1. El sistema está en la $fase$ "pasivo", con lo cual no hay ninguna entidad en el sistema ($q = \emptyset$), y además el número de entidades que llega (n) es menor o igual que la capacidad máxima (N_{max}).

   En este caso, $fase$ pasa a valer "*activo*", las n entidades se almacenan en la variable q, y se programa una transición interna, que sucederá transcurridas Δ unidades de tiempo. Para ello, el valor de σ pasa a ser Δ. Este es el tiempo que se tarda en procesar una de las entidades recién llegadas. Ninguna de las entidades que han llegado debe ser expulsada inmediatamente, con lo cual $q_{balking} = \emptyset$. El nuevo estado es:

   $$(\text{“}activo\text{”}, \Delta, \{x_1, \ldots, x_n\}, \Delta, \emptyset)$$

2. El sistema está en la $fase$ "pasivo" y además el número de entidades que llega (n) es mayor que N_{max}.

En este caso, las n entidades que han llegado se dividen en dos grupos. El primer grupo, compuesto por N_{max} entidades, se almacena en la variable q. El segundo grupo, compuesto por $n - N_{max}$ entidades, deben abandonar inmediatamente el sistema, con lo cual se almacena en la variable $q_{balking}$. La forma de que estas entidades abandonen el sistema es forzar inmediatamente una transición interna. Para ello, se asigna a la variable de estado σ el valor cero.

Esta fase transitoria del sistema, intermedia entre la llegada de entidades y la salida de las que exceden de la capacidad máxima, está definida mediante el valor "*balking*" de la variable de estado $fase$.

Puesto que cuando se produce el evento de entrada no hay ninguna entidad en proceso, el tiempo que tardará en procesarse la entidad es Δ. En consecuencia, el estado resultante de la transición externa es:

$$(\text{“}balking\text{”}, 0, \{x_1, \ldots, x_{N_{max}}\}, \Delta, \{x_{N_{max}+1}, \ldots, x_n\})$$

3. El sistema está en la $fase$ "activo" (equivalentemente, $q \neq \emptyset$) y además el número de entidades que llega es menor o igual que $N_{max} - length(q)$. Es decir, el número de entidades que se encuentran en el sistema ($length(q)$), más el número de entidades que llegan (n), no superan la capacidad máxima del sistema (N_{max}).

   En este caso, la fase del sistema no cambia: sigue siendo "*activo*". Las entidades recién llegadas se añaden a q, que pasa a valer: $q \bullet \{x_1, \cdots, x_n\}$. Ninguna de las entidades recién llegadas debe ser expulsada del sistema: $q_{balking} = \emptyset$. El tiempo restante hasta que finalice el proceso de la entidad que se encuentra actualmente en proceso es: $\sigma - e$.

   El estado resultante de la transición externa es:

$$(\text{“}activo\text{”}, \sigma - e, q \bullet \{x_1, \ldots, x_n\}, \sigma - e, \emptyset)$$

4. El sistema está en la $fase$ "activo" y además el número de entidades que llega es mayor que $N_{max} - length(q)$.

   En este caso, parte de las entidades se añaden a q, de modo que el sistema alcance su capacidad máxima, y el resto de entidades debe abandonar el sistema.

Este caso es el que justifica la introducción de la variable de estado τ_p, en la cual se almacena el tiempo que queda para completar el proceso de la entidad que actualmente se encuentra en proceso: $\tau_p = \sigma - e$.

Puesto que algunas de las entidades recién llegadas deben abandonar el sistema, el modelo pasa a la $fase$ "*balking*" y σ pasa a valer cero. El estado resultante de la transición externa es:

$$(\text{“}balking\text{”}, 0, q \bullet \{x_1, \ldots, x_{N_{max}-length(q)}\}, \sigma - e, \{x_{N_{max}-length(q)+1}, \ldots, x_n\})$$

La definición formal de la función de transición externa, en la que se contemplan las cuatro posibles situaciones descritas anteriormente, es la siguiente:

$$\delta_{ext}\,(fase, \sigma, q, \tau_p, q_{balking}, e, ((\text{“}In\text{”}, x_1), (\text{“}In\text{”}, x_2), \ldots, (\text{“}In\text{”}, x_n))) =$$
$$\begin{cases} (\text{“}activo\text{”}, \Delta, \{x_1, \ldots, x_n\}, \Delta, \emptyset) \\ \qquad\qquad \text{si } fase = \text{“}pasivo\text{”} \text{ y } n \leq N_{max} \\ (\text{“}balking\text{”}, 0, \{x_1, \ldots, x_{N_{max}}\}, \Delta, \{x_{N_{max}+1}, \ldots, x_n\}) \\ \qquad\qquad \text{si } fase = \text{“}pasivo\text{”} \text{ y } n > N_{max} \\ (\text{“}activo\text{”}, \sigma - e, q \bullet \{x_1, \ldots, x_n\}, \sigma - e, \emptyset) \\ \qquad\qquad \text{si } fase = \text{“}activo\text{”} \text{ y } n \leq (N_{max} - length(q)) \\ (\text{“}balking\text{”}, 0, q \bullet \{x_1, \ldots, x_{N_{max}-length(q)}\}, \sigma - e, \{x_{N_{max}-length(q)+1}, \ldots, x_n\}) \\ \qquad\qquad \text{si } fase = \text{“}activo\text{”} \text{ y } n > (N_{max} - length(q)) \end{cases}$$

Cuando se produce una transición interna, se realiza una llamada a la **función de salida**. En este modelo, pueden darse dos situaciones, en función del valor de la variable de estado *fase*.

1. Si $fase =$ "*activo*", la transición interna implica que ha finalizado el proceso de una entidad y ésta debe abandonar el sistema por el puerto "*Out*". En este caso, llamando v al primer elemento de q, la función de salida devuelve el valor ("*Out*", v).

2. Si $fase =$ "*balking*", la transición interna corresponde a que parte de las entidades que acaban de llegar al sistema deben abandonarlo, puesto que se ha alcanzado la capacidad máxima del sistema. La variable de estado $q_{balking}$ almacena las entidades que deben abandonar el sistema. El puerto de salida es "*Balk*". En consecuencia, el valor que devuelve la función de salida es: ("*Out*", $q_{balking}$).

En consecuencia, la función de salida puede describirse de la forma siguiente:

$$\lambda(fase, \sigma, v \bullet q^*, \tau_p, q_{balking}) = \begin{cases} (\text{“}Out\text{”}, v) & \text{si } fase = \text{“}activo\text{”} \\ (\text{“}Balk\text{”}, q_{balking}) & \text{si } fase = \text{“}balking\text{”} \end{cases}$$

La **función de transición interna** determina el cambio en el estado del modelo que se produce en la transición interna. Al igual que en el caso de la función de salida, hay que distinguir dos situaciones: $fase =$ “*activo*” y $fase =$ “*balking*”. El nuevo estado, en cada uno de estos casos, es el siguiente:

1. Si $fase =$ “*activo*”, la transición interna implica que ha finalizado el proceso de una entidad y ésta debe abandonar el sistema. El nuevo estado del sistema depende de si el sistema queda vacío o si, por el contrario, debe comenzar el proceso de otra entidad.

   – Si $q^* = \emptyset$, el sistema queda vacío. El nuevo estado es:

   $$(\text{“}pasivo\text{”}, \infty, \emptyset, \infty, \emptyset)$$

   – Si $q^* \neq \emptyset$, quedan entidades en el sistema, con lo cual una de ellas debe comenzar a ser procesada. El nuevo estado es:

   $$(\text{“}activo\text{”}, \Delta, q^*, \Delta, \emptyset)$$

2. Si $fase =$ “*balking*”, la transición interna corresponde a que parte de las entidades que acaban de llegar al sistema deben abandonarlo. El nuevo estado es:

   $$(\text{“}activo\text{”}, \tau_p, q, \tau_p, \emptyset)$$

   Obsérvese que se asigna a σ el valor del tiempo de proceso restante de la entidad que está actualmente en proceso (τ_p).

Teniendo en cuenta las tres posibles situaciones anteriores, la función de transición interna puede definirse de la forma siguiente:

$$\delta_{int}\left(fase, \sigma, v \bullet q^*, \tau_p, q_{balking}\right) = \begin{cases} (\text{“}pasivo\text{”}, \infty, \emptyset, \infty, \emptyset) \\ \qquad\qquad \text{si } fase = \text{“}activo\text{”} \text{ y } q^* = \emptyset \\ (\text{“}activo\text{”}, \Delta, q^*, \Delta, \emptyset) \\ \qquad\qquad \text{si } fase = \text{“}activo\text{”} \text{ y } q^* \neq \emptyset \\ (\text{“}activo\text{”}, \tau_p, v \bullet q^*, \tau_p, \emptyset) \\ \qquad\qquad \text{si } fase = \text{“}balking\text{”} \end{cases}$$

La **función de avance en el tiempo** devuelve el valor σ:

$$ta\left(fase, \sigma, q, \tau_p, q_{balking}\right) = \sigma$$

Finalmente, la **función de transición confluyente** sería la siguiente.

$$\delta_{con}\left(s, x^b\right) = \delta_{ext}\left(\delta_{int}(s), 0, x^b\right) \tag{4.14}$$

Solución al Ejercicio 4.2

Tal como se indica en el enunciado del problema, el modelo tiene tres puertos: dos de entrada, llamados “*Input*” y “*Read*”, y uno de salida, llamado “*Output*”.

El modelo, descrito mediante el formalismo DEVS paralelo, debe considerar la posibilidad de que lleguen varios eventos simultáneamente a uno o a los dos puertos de entrada. El comportamiento del sistema es el siguiente:

– Supongamos que el sistema está en la fase “*pasivo*” y llegan simultáneamente n eventos al puerto “*Input*”:

$$(\ (\text{“}Input\text{”}, x_1), \ldots, (\text{“}Input\text{”}, x_n)\ )$$

Los valores de estos n eventos de entrada son almacenados en una variable de estado llamada *almacena*:

$$almacena = \{x_1, \ldots, x_n\}$$

eliminándose los valores anteriores que pudiera estar almacenando dicha variable de estado.

– Si el sistema está en la fase "*pasivo*" y llega uno o varios eventos al puerto "*Read*", entonces el sistema cambia a la fase "*responde*". Transcurrido un tiempo Δ, el sistema genera $length(almacena)$ eventos de salida en el puerto "*Output*", correspondientes a los valores almacenados en la variable *almacena*. Dichos eventos de salida se representan de la forma siguiente: ("*Output*", *almacena*).

– Si el sistema está en la fase "*pasivo*" y llegan n eventos al puerto "*Input*"

$$(\ (\text{“}Input\text{”}, x_1), \ldots, (\text{“}Input\text{”}, x_n)\ )$$

y simultáneamente uno o varios eventos al puerto "*Read*", entonces se guardan los valores de los n eventos del puerto "*Input*" en *almacena*

$$almacena = \{x_1, \ldots, x_n\}$$

eliminándose los valores anteriores que pudiera estar almacenando dicha variable de estado. Además el sistema cambia a la fase "*responde*". Transcurrido un tiempo Δ, el sistema genera n eventos de salida en el puerto "*Output*", correspondientes a los n valores que contiene la variable *almacena*.

– Si el sistema está en la fase "*responde*" y llega uno o varios eventos a cualquiera de los puertos de entrada, dichos eventos son ignorados por el sistema.

Los puertos de entrada y salida, y los posibles valores de los eventos en estos puertos son los siguientes:

$$\begin{array}{rcl} X_M & = & \{\ (\text{“}Input\text{”}, \mathbb{R}), (\text{“}Read\text{”}, \mathbb{R})\ \} \\ Y_M & = & \{\ (\text{“}Output\text{”}, \mathbb{R})\ \} \end{array} \tag{4.15}$$

El modelo tiene tres variables de estado: $fase$, σ, y *almacena*. La variable de estado $fase$ puede tomar dos valores: "*pasivo*" y "*responde*". El conjunto de estados secuenciales es:

$$S = \{\text{“}pasivo\text{”}, \text{“}responde\text{”}\} \times \mathbb{R}_0^+ \times \{\text{ secuencia finita de valores } \mathbb{R}\ \}$$

La función de transición externa es:

$$\delta_{ext}(fase, \sigma, almacena, e,$$
$$((\text{“}Input\text{”}, x_1), \ldots, (\text{“}Input\text{”}, x_n), (\text{“}Read\text{”}, r_1), \ldots, (\text{“}Read\text{”}, r_m))) =$$
$$\begin{cases} (\text{“}pasivo\text{”}, \sigma - e, \{x_1, \ldots, x_n\}) & \text{si } fase = \text{“}pasivo\text{” y } p = \text{“}Input\text{”} \\ (\text{“}responde\text{”}, \Delta, almacena) & \text{si } fase = \text{“}pasivo\text{” y } p = \text{“}Read\text{”} \\ (\text{“}responde\text{”}, \Delta, \{x_1, \ldots, x_n\}) & \text{si } fase = \text{“}pasivo\text{” y } p = \{\text{“}Input\text{”}, \text{“}Read\text{”}\} \\ (\text{“}responde\text{”}, \sigma - e, almacena) & \text{si } fase = \text{“}responde\text{”} \end{cases}$$

La función de salida, la función de transición interna y la función de avance en el tiempo son las siguientes:

$$\begin{aligned} \delta_{int}\left(fase, \sigma, almacena\right) &= (\text{“}pasivo\text{”}, \infty, almacena) \\ \lambda\left(fase, \sigma, almacena\right) &= (\text{“}Output\text{”}, almacena) \\ ta\left(fase, \sigma, almacena\right) &= \sigma \end{aligned}$$

Cuando coincide la llegada de eventos externos con el instante en que está planificada una transición interna, en este modelo es conveniente que se produzca en primer lugar la transición interna y a continuación se atiendan los eventos de entrada, produciendo la correspondiente transición externa. De esa forma, en primer lugar se atiende la solicitud de lectura del valor almacenado y a continuación se almacenan los eventos de entrada.

$$\begin{aligned} &\delta_{con}(fase, \sigma, almacena, e, \\ &\quad ((\text{“}Input\text{”}, x_1), \ldots, (\text{“}Input\text{”}, x_n), (\text{“}Read\text{”}, r_1), \ldots, (\text{“}Read\text{”}, r_m))) = \\ &\quad \delta_{ext}(\delta_{int}\left(fase, \sigma, almacena\right), 0, \\ &\qquad ((\text{“}Input\text{”}, x_1), \ldots, (\text{“}Input\text{”}, x_n), (\text{“}Read\text{”}, r_1), \ldots, (\text{“}Read\text{”}, r_m))) \end{aligned}$$

Solución al Ejercicio 4.3

Según se indica en el enunciado, el modelo tiene dos puertos de entrada (“Start” y “Stop”) y uno de salida (“Output”). Los valores posibles de los eventos en estos puertos son:

$$\begin{array}{rcl} X_M & = & \{\ (\text{“}Start\text{”}, \mathbb{R}), (\text{“}Stop\text{”}, \mathbb{R})\ \} \\ Y_M & = & \{\ (\text{“}Output\text{”}, \{1\})\ \} \end{array} \tag{4.16}$$

Se realizan las hipótesis siguientes acerca del funcionamiento del sistema:

1. La llegada simultánea de varios eventos a un mismo puerto tiene el mismo efecto que si llegara un único evento a ese puerto. Es decir, si llega uno o varios eventos al puerto “Start”, el sistema comienza a generar eventos. Si llega uno o varios eventos al puerto “Stop”, el sistema deja de generar eventos.

2. La acción de parada tiene prioridad sobre la acción de inicio. Es decir, si llegan simultáneamente uno o varios eventos a los dos puertos de entrada, entonces el sistema se comporta como si hubieran llegado eventos únicamente al puerto “Stop”.

3. Si en el instante en que está planificada una transición interna se recibe uno o varios eventos de entrada, entonces primero se genera el evento de salida y a continuación se realiza la acción (parada o inicio) asociada a los eventos externos.

El modelo tiene dos variables de estado: $fase$ y σ. La variable de estado $fase$ puede tomar dos valores: $\{\text{“}pasivo\text{”}, \text{“}activo\text{”}\}$. El conjunto de estados secuenciales es:

$$S = \{\text{“}pasivo\text{”}, \text{“}activo\text{”}\} \times \mathbb{R}_0^+$$

El comportamiento del sistema está descrito por las siguientes funciones de transición, de salida y de avance en el tiempo:

$$\begin{array}{rcl} \delta_{ext}\left(fase, \sigma, e, X_M^b\right) & = & \begin{cases} (\text{“}pasivo\text{”}, \infty) & \text{si } p = \text{“}Stop\text{”}, \text{ o bien } p = \{\text{“}Start\text{”}, \text{“}Stop\text{”}\} \\ (\text{“}activo\text{”}, 0) & \text{si } p = \text{“}Start\text{”} \end{cases} \\ \delta_{int}\left(fase, \sigma\right) & = & (\text{“}activo\text{”}, periodo) \\ \delta_{con}\left(fase, \sigma, e, X_M^b\right) & = & \begin{cases} (\text{“}pasivo\text{”}, \infty) & \text{si } p = \text{“}Stop\text{”}, \text{ o bien } p = \{\text{“}Start\text{”}, \text{“}Stop\text{”}\} \\ (\text{“}activo\text{”}, periodo) & \text{si } p = \text{“}Start\text{”} \end{cases} \\ \lambda\left(fase, \sigma\right) & = & 1 \\ ta\left(fase, \sigma\right) & = & \sigma \end{array}$$

Supongamos ahora que modificamos la tercera hipótesis que hemos realizado acerca del funcionamiento del sistema, de manera que, cuando se produce confluencia entre la transición interna y la externa, sólo se genera un evento de salida si la acción externa es inicio. Si la acción externa es parar, entonces no se genera el evento de salida.

En este caso, el modelo puede definirse empleando tres variables de estado:

- $fase$, puede tomar tres valores: $\{$"*pasivo*", "*activo*", "*transitorio*"$\}$.
- σ, puede tomar valores reales positivos, incluyendo el cero.
- *enable*, puede tomar los valores: $\{false, true\}$.

El conjunto de estados secuenciales es:

$$S = \{\text{"}pasivo\text{"}, \text{"}activo\text{"}, \text{"}transitorio\text{"}\} \times \mathbb{R}_0^+ \times \{false, true\}$$

El comportamiento del sistema está descrito por las siguientes funciones de transición, de salida y de avance en el tiempo:

$$\begin{aligned}
\delta_{ext}\left(S, e, X_M^b\right) &= \begin{cases} (\text{"}pasivo\text{"}, \infty, false) & \text{si } p = \text{"}Stop\text{" o } p = \{\text{"}Start\text{"}, \text{"}Stop\text{"}\} \\ (\text{"}transitorio\text{"}, 0, true) & \text{si } p = \text{"}Start\text{"} \end{cases} \\
\delta_{int}\left(S\right) &= \begin{cases} (\text{"}transitorio\text{"}, 0, true) & \text{si } fase = \text{"}activo\text{"} \\ (\text{"}activo\text{"}, periodo, false) & \text{si } fase = \text{"}transitorio\text{"} \end{cases} \\
\delta_{con}\left(S, e, X_M^b\right) &= \begin{cases} (\text{"}pasivo\text{"}, \infty, false) & \text{si } p = \text{"}Stop\text{", o } p = \{\text{"}Start\text{"}, \text{"}Stop\text{"}\} \\ (\text{"}transitorio\text{"}, 0, true) & \text{si } p = \text{"}Start\text{"} \end{cases} \\
\lambda\left(S\right) &= \begin{cases} 1 & \text{si } enable = true \\ \emptyset & \text{si } enable = false \end{cases} \\
ta\left(S\right) &= \sigma
\end{aligned}$$

El estado inicial de este modelo sería: $\{$"*pasivo*", $\infty, false\}$.

Solución al Ejercicio 4.4

En la Figura 4.7 se muestra la interfaz del modelo de un conmutador. Tiene dos puertos de entrada, $InPorts = \{in, in1\}$, y dos puertos de salida, $OutPorts =$

$\{out, out1\}$. Obsérvese que puede producirse la llegada simultánea de varios eventos a los puertos de entrada y la generación de varios eventos simultáneos en los puertos de salida.

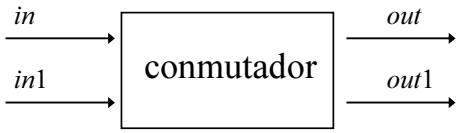

Figura 4.7: Modelo DEVS de un conmutador.

El comportamiento del conmutador está caracterizado por el valor de la variable de estado Sw, que puede ser $\{true, false\}$. Cada vez que se produce uno o varios eventos simultáneos de entrada, cambia el valor de la variable Sw, el cual determina el funcionamiento del conmutador:

- Mientras $Sw = true$, las entidades que llegan al puerto in son enviadas al puerto out. Similarmente, las entidades que llegan al puerto $in1$ son enviadas al puerto $out1$.
- Mientras $Sw = false$, las entidades que llegan al puerto in son enviadas al puerto $out1$ y las que llegan al puerto $in1$ son enviadas al puerto out.

En ambos casos, se produce un retardo desde la llegada del evento o eventos simultáneos de entrada, y la generación del evento o eventos de salida. Este tiempo de proceso es un parámetro del modelo, que llamaremos Δ. Durante este tiempo de proceso, el sistema no responde a los eventos de entrada.

El estado del sistema está caracterizado por las cinco variables de estado siguientes:

- La variable *fase*, que puede valer {"pasivo", "ocupado"}.
- La variable σ, que puede tomar valores pertenecientes al conjunto $\mathbb{R}_0^+$.
- Las variables $almacena$ y $almacena1$ guardan los valores de los eventos de entrada llegados al puerto in y $in1$, respectivamente. Cada una de estas variables almacena una secuencia finita de valores reales.
- La variable Sw, que puede tomar valores $\{true, false\}$.

Así pues, el estado del sistema está caracterizado por las siguientes cinco variables: $(fase, \sigma, almacena, almacena1, Sw)$. El conjunto de posibles estados viene definido por el producto cartesiano de los posibles valores de cada una de estas variables: $S = \{\text{“pasivo”}, \text{“activo”}\} \times \mathbb{R}_0^+ \times \{\text{Secuencia finita de valores } \mathbb{R}\} \times \{\text{Secuencia finita de valores } \mathbb{R}\} \times \{true, false\}$. La descripción del sistema es la siguiente:

$$\text{DEVS}_{\Delta} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{4.17}$$

donde los conjuntos de entrada y salida son los siguientes:

$$\begin{array}{rcll} X_M & = & \{\ (p,v) \mid p \in InPorts = \{in, in1\},\ v \in X_p\ \}, & \text{con } X_{in} = X_{in1} = \mathbb{R} \\ Y_M & = & \{\ (p,v) \mid p \in OutPorts = \{out, out1\},\ v \in Y_p\ \}, & \text{con } Y_{out} = Y_{out1} = \mathbb{R} \end{array} \tag{4.18}$$

Supongamos que simultáneamente se reciben eventos en los dos puertos de entrada. Llamemos X_{in}^b a la bolsa de eventos recibida en el puerto in y X_{in1}^b a la recibida en el puerto $in1$. Si en alguno de estos puertos no se recibieran eventos, la correspondiente bolsa sería igual al conjunto vacío. Las funciones de transición son:

$$\begin{array}{rcll} \delta_{ext}\left(S, e, X_M^b\right) & = & \left\{ \begin{array}{ll} \left(\text{“activo”}, \Delta, X_{in}^b, X_{in1}^b, !Sw\right) & \text{si } fase = \text{“pasivo”} \\ \left(fase, \sigma - e, almacena, almacena1, Sw\right) & \text{en caso contrario} \end{array} \right. \\ \delta_{int}\left(S\right) & = & \left(\text{“pasivo”}, \infty, almacena, almacena1, Sw\right) & \\ \delta_{con}\left(S, e, X_M^b\right) & = & \delta_{ext}\left(\delta_{int}\left(S\right), 0, X_M^b\right) & \end{array} \tag{4.19}$$

La función de salida es:

$$\lambda\left(S\right) = \left\{ \begin{array}{ll} \left(out, almacena\right), \left(out1, almacena1\right) & \text{si } Sw = true \\ \left(out, almacena1\right), \left(out1, almacena\right) & \text{si } Sw = false \end{array} \right. \tag{4.20}$$

Si la bolsa de eventos asociada a un puerto está vacía, no se produce ningún evento de salida por el correspondiente puerto. Finalmente, la función de avance del tiempo es:

$$ta\left(S\right) = \sigma \tag{4.21}$$

El modelo de la red de conmutación mostrada en la Figura 4.5 es el siguiente.

$$N = \langle\ X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC\ \rangle \tag{4.22}$$

donde:

$InPorts = \{\text{“in”}, \text{“in1”}\}$ $X_{in} = X_{in1} = \mathbb{R}$	Puerto de entrada del modelo compuesto. Posibles valores de los eventos en los puertos “in” e “in1” del modelo compuesto.
$OutPorts = \{\text{“out”}\}$ $Y_{out} = \mathbb{R}$	Puerto de salida del modelo compuesto. Posibles valores de los eventos en el puerto “out” del modelo compuesto.
$D = \{s_0, p_0, p_1\}$	Conjunto de nombres de los componentes.
$M_{s_0} = M_{\text{conmutador}}$ $M_{p_0} = M_{p_1} = M_{\text{recurso}}$	Tipo de los componentes.
$\text{EIC} = \{\ ((N, \text{“in”}), (s_0, \text{“in”}))\,,$ $\quad ((N, \text{“in1”}), (s_0, \text{“in1”}))\ \}$	Conexiones externas de entrada.
$\text{EOC} = \{\ ((p_0, \text{“out”}), (N, \text{“out”}))\,,$ $\quad ((p_1, \text{“out”}), (N, \text{“out”}))\ \}$	Conexiones externas de salida.
$\text{IC} = \{\ ((s_0, \text{“out”}), (p_0, \text{“in”}))\,,$ $\quad ((s_0, \text{“out1”}), (p_1, \text{“in”}))\ \}$	Conexiones internas.

Tema 5

MODELADO HÍBRIDO EN DEVS

OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema debería saber:

- Describir modelos híbridos empleando el formalismo DEV&DESS.
- Discutir cómo los formalismos básicos DESS, DTSS y DEVS clásico pueden ser interpretados como tipos especiales de DEV&DESS.
- Interpretar la conexión de componentes de diferentes formalismos.
- Discutir el algoritmo de la simulación de modelos híbridos y DESS.

5.1. INTRODUCCIÓN

Los modelos híbridos son aquellos que tienen una parte de tiempo continuo, y otra parte de eventos discretos y/o tiempo discreto. Una forma de describir este tipo de modelos es mediante la definición de un nuevo formalismo[1], que combine la especificación de modelos de eventos discretos (DEVS clásico) y la especificación de modelos mediante ecuaciones diferenciales (DESS). Este nuevo formalismo, que engloba los dos formalismos originales (DEVS y DESS), recibe el nombre de DEV&DESS (Zeigler et al. 2000). En este tema se describe el modelado empleando dicho formalismo y algunos conceptos de la simulación de este tipo de modelos.

5.2. FORMALISMO DEV&DESS

Como se ha indicado anteriormente, el formalismo DEV&DESS (*Discrete Event and Differential Equation System Specification*) permite describir modelos híbridos, combinando para ello los formalismos DEVS clásico y DESS. En la Figura 5.1 se ilustra el concepto de modelado desarrollado en DEV&DESS.

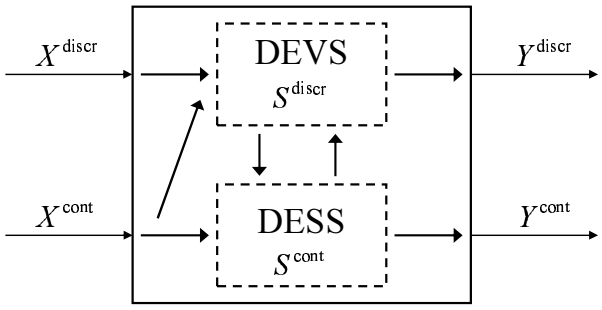

Figura 5.1: Modelo combinando los formalismos DEVS y DESS.

Los puertos de entrada X^{discr} aceptan eventos, mientras que los puertos de entrada X^{cont} aceptan segmentos continuos a tramos y constantes. Estos últimos influyen sobre la parte continua y discreta del modelo, mientras que los eventos de

[1]Abreviaturas usadas para designar los formalismos:

DESS Differential Equation System Specification

DEVS Discrete Event System Specification

DTSS Discrete Time System Specification

DEV&DESS Discrete Event and Differential Equation System Specification

entrada sólo afectan a la parte discreta del modelo. Cada parte del modelo produce su propia trayectoria de salida: Y^{discr} la parte discreta e Y^{cont} la parte continua. Asimismo, cada una de las partes puede influir sobre el estado de la otra parte. Se representa S^{discr} y S^{cont} el estado de la parte discreta y continua, respectivamente.

5.2.1. Detección de los eventos en el estado

Un concepto fundamental del modelado híbrido es cómo la parte discreta es afectada por la parte continua. Es decir, cómo la parte DESS del modelo puede disparar la ejecución de eventos. En la Figura 5.2 se ilustra la forma en que esto se produce. Se asocia a cada tipo de evento una función lógica, denominada *condición de evento*, que tiene en general la forma siguiente: *expresion* > 0. El evento se dispara en el instante preciso en que la condición de evento asociada pasa de valer *false* a valer *true*.

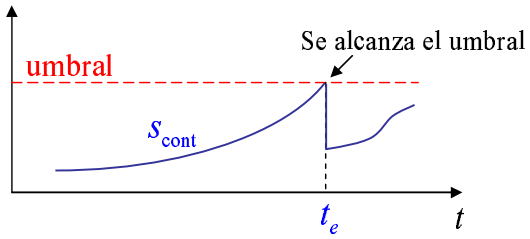

Figura 5.2: Evento en el estado.

En el ejemplo mostrado en la Figura 5.2, la condición de disparo del evento es que el valor de la variable de tiempo continuo s_{cont} alcance determinado valor umbral. La *condición de evento* es: $s_{\text{cont}} - \text{umbral} > 0$.

En el algoritmo de la simulación del modelo, cada condición de evento es sustituida por una *función detectora de cruce por cero*, de modo que el evento se dispara cuando la función pasa de tomar un valor distinto de cero a valer cero, o cuando cruza por cero. Por ejemplo, la condición de evento $s_{\text{cont}} - \text{umbral} > 0$, es sustituida por la función detectora de cruce por cero es: $z = s_{\text{cont}} - \text{umbral}$.

Un evento cuya condición de disparo viene expresada en términos de variables de tiempo continuo se denomina *evento en el estado*. Los eventos de la parte DEVS del modelo, cuya condición de disparo es que el reloj de la simulación alcance determinado valor, se denominan *eventos en el tiempo*.

5.2.2. Modelos atómicos

En esta sección se muestra la forma en que se describe un modelo atómico híbrido empleando el formalismo DEV&DESS. Por simplicidad, se define el formalismo sin distinguir entre los *eventos en el tiempo* y los *eventos en el estado*. Todos los eventos son tratados como si fueran eventos en el estado.

Por ejemplo, pueden planificarse las transiciones internas de la parte DEVS del modelo asociando al evento interno la condición de evento: $e - ta(s) > 0$, donde e es el tiempo transcurrido desde el último evento. La función de cruce asociada es: $z = e - ta(s)$.

Si bien esto simplifica las explicaciones, no es eficiente desde el punto de vista computacional, ya que el simulador debe ir calculando las funciones de cruce durante la integración de la parte continua del modelo. Podría extenderse el formalismo de manera sencilla con el fin de considerar los eventos en el tiempo, siguiendo para ello una estrategia para la planificación de eventos en el tiempo similar a la que se sigue en DEVS.

De acuerdo con el formalismo DEV&DESS, el modelo híbrido está formado por los componentes de la tupla siguiente:

$$\text{DEV\&DESS} = \left\langle X^{\text{discr}}, X^{\text{cont}}, Y^{\text{discr}}, Y^{\text{cont}}, S^{\text{discr}}, S^{\text{cont}}, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{\text{discr}}, f, \lambda^{\text{cont}} \right\rangle \tag{5.1}$$

donde:

$X^{\text{discr}}, Y^{\text{discr}}$	Conjunto de pares puerto-valor, de entrada y salida, del modelo DEVS.
$X^{\text{cont}} = \{(x_1^{\text{cont}}, x_2^{\text{cont}}, \dots) \mid x_1^{\text{cont}} \in X_1^{\text{cont}}, \dots\}$	Conjunto de variables de entrada del modelo continuo.
$Y^{\text{cont}} = \{(y_1^{\text{cont}}, y_2^{\text{cont}}, \dots) \mid y_1^{\text{cont}} \in Y_1^{\text{cont}}, \dots\}$	Conjunto de variables de salida del modelo continuo.
$S = S^{\text{discr}} \times S^{\text{cont}}$	Estado del sistema híbrido: producto cartesiano del estado discreto y del estado continuo.

$\delta_{ext} = Q \times X^{\text{cont}} \times X^{\text{discr}} \rightarrow S$	Función de transición externa. Los argumentos de la función son las entradas continuas, los eventos de entrada y el estado total del sistema híbrido. La función devuelve el nuevo estado del sistema híbrido.
$Q = \{(s^{\text{discr}}, s^{\text{cont}}, e) \mid s^{\text{discr}} \in S^{\text{discr}}, s^{\text{cont}} \in S^{\text{cont}}, e \in \mathbb{R}_0^+\}$	Estado total, donde e es el tiempo transcurrido desde el último evento.
$\delta_{int} : \quad Q \times X^{\text{cont}} \rightarrow S$	Función de transición interna.
$\lambda^{\text{discr}} : \quad Q \times X^{\text{cont}} \rightarrow Y^{\text{discr}}$	Función de salida de la parte discreta.
$\lambda^{\text{cont}} : \quad Q \times X^{\text{cont}} \rightarrow Y^{\text{cont}}$	Función de salida de la parte continua.
$f : \quad Q \times X^{\text{cont}} \rightarrow S^{\text{cont}}$	Función para el cálculo de la derivada de las variables de estado.
$C_{int} : \quad Q \times X^{\text{cont}} \rightarrow Boolean$	Condiciones de evento.

5.2.3. Simulación de modelos atómicos

Empleando la semántica del modelo DEV&DESS, puede describirse de manera informal cómo realizar la simulación del modelo híbrido:

1. **Intervalos (t_1, t_2) sin eventos.** Durante los intervalos de tiempo en que no se producen eventos, únicamente cambia el valor de las variables de estado de tiempo continuo, S^{cont}, el valor de las variables de salida de tiempo continuo, Y^{cont}, y el tiempo transcurrido desde el último evento, e. El comportamiento continuo del modelo está especificado por la función para el cálculo de las derivadas, f, y por la función para el cálculo de las salidas de tiempo continuo, λ^{cont}.

- El valor al final del intervalo (instante t_2) de las variables de estado continuas se calcula como la suma de su valor al comienzo del intervalo (instante t_1) más la integral de la función f a lo largo del intervalo.
- El valor que tiene el tiempo transcurrido, e, al final del intervalo se calcula incrementando en $t_2 - t_1$ el valor que tiene e al comienzo del intervalo.

2. **Evento en el estado en el instante t del intervalo (t_1, t_2).** Supongamos que en el intervalo (t_1, t) no se produce ningún evento, y que en el instante t, con $t_1 < t < t_2$, la condición de evento C_{int} pasa de valer *false* a valer *true*. Esto significa que en el instante t se verifican las condiciones de disparo de un evento en el estado. Las acciones asociadas son:

   a) Se calcula el valor que tienen las variables de estado de la parte continua del modelo en el instante t, justo antes de la ejecución del evento. Para ello, se suma el valor que tienen en t_1 al valor de la integral sobre (t_1, t) de la función derivada, f. Asimismo, se genera la salida continua hasta el instante t, evaluando para ello la función λ^{cont}.

   b) Se calcula el valor de e, incrementando en $t - t_1$ el valor que tuviera e al comienzo del intervalo.

   c) Se ejecuta la función de salida de la parte discreta, λ^{discr}, para generar el evento de salida en el instante t.

   d) Se ejecuta la función de transición interna, $\delta_{int}(s, e, x^{\text{cont}})$, para calcular el nuevo valor de las variables de estado continuas y discretas.

   e) Se asigna el valor cero al tiempo transcurrido: $e = 0$.

3. **Evento externo en un puerto de entrada, en el instante t del intervalo (t_1, t_2).** Supongamos que en el intervalo (t_1, t) no se produce ningún evento, y que en el instante t, con $t_1 < t < t_2$, se produce un evento externo en un puerto de entrada de la parte discreta del modelo. Las acciones asociadas son:

   a) Se calcula el valor que tienen las variables de estado de la parte continua del modelo en el instante t, justo antes de la ejecución del evento. Para ello, se suma el valor que tienen en t_1 al valor de la integral sobre (t_1, t) de la función derivada, f. Asimismo, se genera la salida de la parte continua del modelo hasta el instante t.

   b) Se calcula el valor de e, incrementando en $t - t_1$ el valor que tuviera e al comienzo del intervalo.

   c) Se ejecuta la función de transición externa, δ_{ext}, para calcular el nuevo estado en el instante t.

d) Se asigna el valor cero al tiempo transcurrido: $e = 0$.

5.2.4. Modelos compuestos

La especificación formal de un sistema DEV&DESS compuesto es análoga a la descrita en la Sección 3.4 para los modelos acoplados según el formalismo DEVS clásico. Por consiguiente, consiste en la tupla siguiente:

$$N = (X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC,\ select) \tag{5.2}$$

5.2.5. Proceso de llenado de barriles

En esta sección se muestra un ejemplo de modelado, empleando el formalismo DEV&DESS, de un proceso de llenado de barriles.

Descripción informal del funcionamiento del sistema

El llenado con líquido de un barril es un proceso continuo, sin embargo, el control del proceso puede modelarse de manera discreta. El control incluye posicionar el barril bajo el grifo, abrir la válvula en el momento en que debe iniciarse el vertido del líquido y, finalmente, cerrar la válvula cuando el líquido dentro del barril alcanza un determinado nivel. En la Figura 5.3 se muestra la interfaz y las variables de estado del modelo.

Figura 5.3: Modelo del proceso de llenado de barriles.

El modelo tiene un puerto de entrada continuo (*flujoIn*) y un puerto de entrada discreto (*on/off*). Tiene un puerto de salida discreto (*barril*) y un puerto de salida continuo (*cout*). Asimismo, tiene dos variables de estado:

– Una continua, llamada (*contenido*), que representa el volumen de líquido actual del barril.

– Una discreta, llamada (*valvula*), que representa el estado de la válvula y puede tomar los dos valores siguientes: {"abierta", "cerrada"}.

La variable de entrada *flujoIn* representa el flujo volumétrico de líquido que se emplea para llenar el barril. La relación que hay entre la variable de entrada *flujoIn* y la variable de estado *contenido* es la siguiente: la derivada de la variable de estado *contenido* es igual a la variable de entrada *flujoIn* cuando la válvula está abierta ($valvula$ = "abierta"), e igual a cero cuando la válvula está cerrada ($valvula$ = "cerrada"). Es decir:

$$\frac{dcontenido}{dt} = \begin{cases} flujoIn & \text{si } valvula = \text{"abierta"} \\ 0 & \text{si } valvula = \text{"cerrada"} \end{cases} \tag{5.3}$$

La entrada discreta *on/off*, que sólo puede tomar los valores {"on", "off"}, hace que el modelo conmute entre dos fases: válvula abierta ($valvula$ = "abierta") y válvula cerrada ($valvula$ = "cerrada").

El modelo tiene las dos salidas siguientes:

– La salida cout es una variable de tiempo continuo, cuyo valor coincide con el volumen actual de llenado del barril.

– El modelo genera un evento de salida de valor "barrilDe10litros", a través del puerto de salida *barril*, cada vez que se completa el llenado de un barril.

El modelo genera un evento en el puerto de salida *barril* cada vez que un barril alcanza su nivel máximo de llenado, es decir, cuando la variable de estado de tiempo continuo *contenido* alcanza el valor 10 litros. Se trata, por tanto, de un evento en el estado. Cuando un barril se llena, se supone que es reemplazado por otro vacío. Consecuentemente, la variable *contenido* es puesta a cero.

En la Figura 5.4 se muestra el comportamiento del sistema frente a una determinada trayectoria de entrada. Se observa que la entrada *on/off* condiciona el valor de la variable *valvula*, la cual, a su vez, determina el comportamiento de la variable *contenido*:

– Mientras la válvula está abierta, el flujo de entrada llena los barriles (la derivada de *contenido* es igual a *flujoIn*).

– Mientras está cerrada, el contenido del barril permanece constante (la derivada de *contenido* es cero).

Cuando el contenido del barril alcanza los 10 litros, se genera un evento de salida por el puerto *barril* y se pone a cero la variable *contenido*. En todo momento, la variable de salida *cout* es igual a la variable *contenido*.

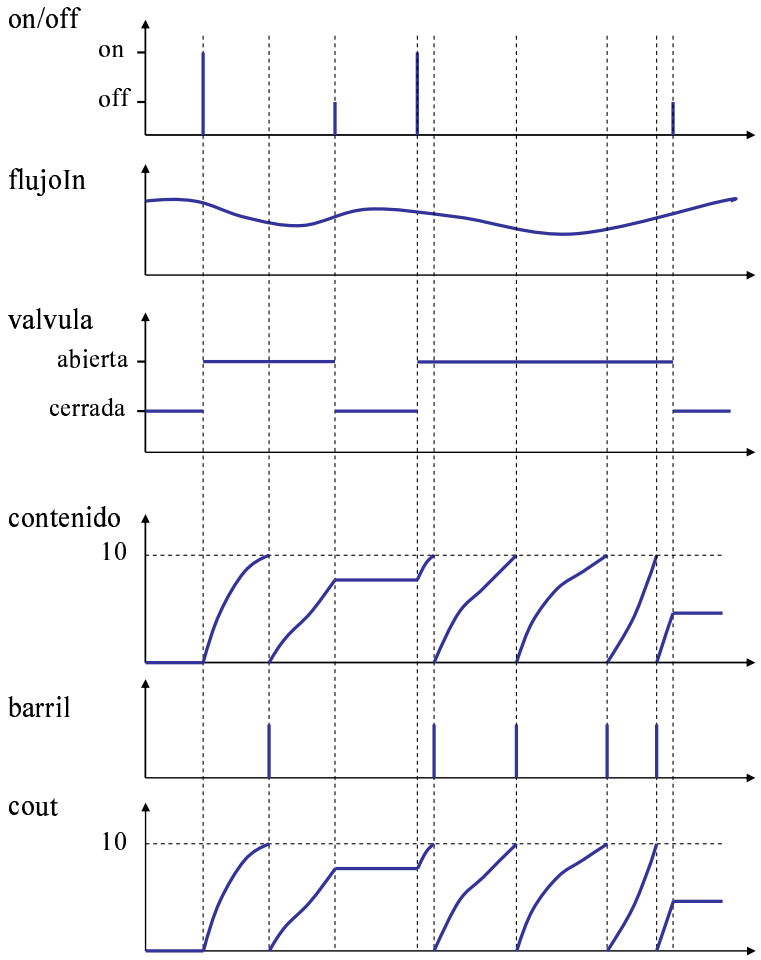

Figura 5.4: Trayectorias del modelo del proceso de llenado de barriles.

Descripción formal

La descripción formal del modelo es la siguiente:

$$\text{LlenadoBarriles} = \left\langle X^{\text{discr}}, X^{\text{cont}}, Y^{\text{discr}}, Y^{\text{cont}}, S^{\text{discr}}, S^{\text{cont}}, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{\text{discr}}, f, \lambda^{\text{cont}} \right\rangle \tag{5.4}$$

donde la interfaz del modelo está definida por los siguientes elementos de la tupla:

$$\begin{aligned} X^{\text{cont}} &= \{flujoIn \mid flujoIn \in \mathbb{R}\} && (5.5)\\ X^{\text{discr}} &= \{on/off \mid on/off \in \{\text{“on”}, \text{“off”}\}\} && (5.6)\\ Y^{\text{cont}} &= \{cout \mid cout \in \mathbb{R}\} && (5.7)\\ Y^{\text{discr}} &= \{barril \mid barril \in \{\text{“barrilDe10litros”}\}\} && (5.8) \end{aligned}$$

El estado del sistema está definido mediante los dos siguientes elementos de la tupla:

$$\begin{aligned} S^{\text{cont}} &= \{contenido \mid contenido \in \mathbb{R}\} && (5.9)\\ S^{\text{discr}} &= \{valvula \mid valvula \in \{\text{“abierta”}, \text{“cerrada”}\}\} && (5.10) \end{aligned}$$

La función de transición externa define cómo cambia el estado del modelo en función de las entradas al modelo, tanto las de tiempo discreto (los eventos de entrada) como las entradas de tiempo continuo. En este modelo, la función de transición externa determina el valor de la variable de estado valvula en función del valor del evento de entrada recibido a través del puerto *on/off*:

$$\begin{aligned} &\delta_{ext}((contenido, valvula), e, flujoIn, on/off) : \\ &\quad \text{if } on/off = \text{“on”} \text{ then } valvula := \text{“abierta”} \\ &\quad \text{if } on/off = \text{“off”} \text{ then } valvula := \text{“cerrada”} \end{aligned} \tag{5.11}$$

La condición que determina el disparo del evento en el estado es que el volumen de líquido en el barril alcance el valor 10 litros. La condición de evento es:

$$\begin{aligned} &C_{int}((contenido, valvula), e, flujoIn) : \\ &\quad contenido > 10 \end{aligned} \tag{5.12}$$

Cuando se dispara un evento en el estado, se evalúa la función de transición interna con el fin de calcular el nuevo estado. En este modelo, el evento pone a cero la variable contenido (el barril lleno es reemplazado por otro vacío) y no modifica el valor de la variable valvula.

$$\begin{array}{l}\delta_{int}((contenido, valvula), e, flujoIn) : \\ \quad contenido = 0\end{array} \tag{5.13}$$

Como parte de las acciones asociadas a una transición interna, se evalúa la función de salida discreta, λ^{discr}, con el fin de calcular el valor del evento de salida. En este modelo, cuando se produce el evento en el estado, se genera un evento en el puerto de salida *barril* cuyo valor es siempre el mismo: "barrilDe10litros". La función de salida de la parte discreta del modelo es:

$$\begin{array}{l}\lambda^{discr}((contenido, valvula), e, flujoIn) : \\ \quad \text{Genera evento de valor "barrilDe10litros" en el puerto barril}\end{array} \tag{5.14}$$

La función f, de cálculo de las derivadas de las variables de estado de tiempo continuo, es la siguiente:

$$\begin{array}{l}f((contenido, valvula), e, flujoIn) : \\ \quad \text{if } valvula = \text{"abierta" then } \frac{dcontenido}{dt} = flujoIn \\ \quad \text{if } valvula = \text{"cerrada" then } \frac{dcontenido}{dt} = 0\end{array} \tag{5.15}$$

Finalmente, la función de salida de la parte continua del modelo, λ^{cont}, permite calcular la variable de salida cout, cuyo valor es igual en todo momento al valor de la variable de estado contenido:

$$\begin{array}{l}\lambda^{cont}((contenido, valvula), e, flujoIn) : \\ \quad cout = contenido\end{array} \tag{5.16}$$

5.3. FORMALISMOS BÁSICOS Y DEV&DESS

En esta sección veremos cómo los formalismos básicos DESS, DTSS y DEVS clásico pueden ser interpretados como tipos especiales de DEV&DESS. Esta in-

terpretación nos permitirá, cuando en la próxima sección definamos los modelos DEV&DESS modulares y jerárquicos, realizar modelos acoplados multiformalismo.

5.3.1. Formalismo DESS

Es posible especificar un modelo DESS (*Differential Equation System Specification*) mediante el formalismo DEV&DESS. Para ello, basta con omitir toda la parte relativa a eventos del formalismo DEV&DESS.

Con el fin de justificar la afirmación anterior, en primer lugar analizaremos cuál es la definición formal de un modelo de acuerdo al formalismo DESS. Un modelo descrito mediante el formalismo DESS es la tupla siguiente:

$$\text{DESS} = (X_{dess}, Y_{dess}, Q_{dess}, f_{dess}, \lambda_{dess}) \tag{5.17}$$

donde:

X_{dess}	Conjunto de entradas.
Y_{dess}	Conjunto de salidas.
Q_{dess}	Conjunto de estados.
$f_{dess} : Q_{dess} \times X_{dess} \rightarrow Q_{dess}$	Funciones para el cálculo de las derivadas. $\frac{dq}{dt} = f_{dess}(q, x)$
$\lambda_{dess} : Q_{dess} \rightarrow Y_{dess}$	Función de salida tipo Moore. La salida es función del tiempo y del estado: $\lambda_{dess}(q)$.
$\lambda_{dess} : Q_{dess} \times X_{dess} \rightarrow Y_{dess}$	Función de salida tipo Mealy. La salida es función del tiempo, del estado y de la entrada: $\lambda_{dess}(q, x)$.

Las entradas, salidas y variables de estado son variables reales de tiempo continuo. Así pues, X_{dess}, Y_{dess} y Q_{dess} son los espacios vectoriales $\mathbb{R}^m$, $\mathbb{R}^p$ y $\mathbb{R}^n$ respectivamente. La función de salida puede ser o bien de tipo Moore, o bien de tipo Mealy.

El modelo DESS descrito anteriormente constituye un tipo especial de modelo DEV&DESS,

$$\text{DEV\&DESS} = \left\langle X^{\text{discr}}, X^{\text{cont}}, Y^{\text{discr}}, Y^{\text{cont}}, S^{\text{discr}}, S^{\text{cont}}, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{\text{discr}}, f, \lambda^{\text{cont}} \right\rangle \tag{5.18}$$

donde:

$$X^{\text{cont}} = X_{dess} \tag{5.19}$$
$$Y^{\text{cont}} = Y_{dess} \tag{5.20}$$
$$S^{\text{cont}} = Q_{dess} \tag{5.21}$$
$$f = f_{dess} \tag{5.22}$$
$$\lambda_{cont} = \lambda_{dess} \tag{5.23}$$

y toda la parte relativa a los eventos es omitida:

$$X^{\text{discr}} = \{\} \tag{5.24}$$
$$Y^{\text{discr}} = \{\} \tag{5.25}$$
$$S^{\text{discr}} = \{\} \tag{5.26}$$
$$\delta_{int}((s,e), x^{\text{cont}}) = s \tag{5.27}$$
$$\delta_{ext}((s,e), x^{\text{cont}}) = s \tag{5.28}$$
$$\lambda_{discr}((s,e), x^{\text{cont}}) = \emptyset \tag{5.29}$$
$$C_{int}((s,e), x^{\text{cont}}) = false \tag{5.30}$$

5.3.2. Formalismo DTSS

Puede obtenerse un modelo DEV&DESS que es equivalente a un modelo DTSS empleando la variable e, que representa el tiempo transcurrido desde el último evento, para planificar las transiciones en el estado a intervalos de tiempo constantes h. Cuando el tiempo transcurrido desde el último evento alcanza el valor h, entonces la *condición de evento* del modelo DEV&DESS, definida como $C_{int} : e \geq h$, pasa de valer *false* a valer *true*, y se dispara un evento interno. La función de transición interna del modelo DEV&DESS debe describirse de tal manera que el estado resultante sea el especificado en el modelo DTSS equivalente. Recuérdese que el valor de

e se pone a cero en el modelo DEV&DESS como parte de las acciones asociadas a una transición interna.

Las ideas anteriores pueden expresarse de manera formal. Para hacerlo, en primer lugar analizaremos cuál es la definición formal de un modelo DTSS. Un modelo DTSS está compuesto por la tupla siguiente:

$$\text{DTSS} = (X_{dtss}, Y_{dtss}, Q_{dtss}, \delta_{dtss}, \lambda_{dtss}, h_{dtss}) \tag{5.31}$$

donde:

X_{dtss}	Conjunto de entradas.
Y_{dtss}	Conjunto de salidas.
Q_{dtss}	Conjunto de estados.
$\delta : Q_{dtss} \times X_{dtss} \rightarrow Q_{dtss}$	Función de transición de estados.
$\lambda_{dtss} : Q_{dtss} \rightarrow Y_{dtss}$	Función de salida tipo Moore. La salida es función del tiempo y del estado: $\lambda_{dess}(q)$.
$\lambda_{dtss} : Q_{dtss} \times X_{dtss} \rightarrow Y_{dtss}$	Función de salida tipo Mealy. La salida es función del tiempo, del estado y de la entrada: $\lambda_{dess}(q, x)$.
h_{dtss}	Paso de avance en el tiempo.

El modelo DTSS descrito anteriormente constituye un tipo especial de modelo DEV&DESS,

$$\text{DEV\&DESS} = \left\langle X^{\text{discr}}, X^{\text{cont}}, Y^{\text{discr}}, Y^{\text{cont}}, S^{\text{discr}}, S^{\text{cont}}, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{\text{discr}}, f, \lambda^{\text{cont}} \right\rangle \tag{5.32}$$

donde:

$$\begin{array}{ll} X^{\text{discr}} = X_{dtss} & X^{\text{cont}} = \{\} \\ Y^{\text{discr}} = Y_{dtss} & Y^{\text{cont}} = \{\} \\ S^{\text{discr}} = Q_{dtss} \times \{h_{dtss}\} & S^{\text{cont}} = \{\} \end{array} \tag{5.33}$$

La condición de evento se define de la forma siguiente:

$$C_{int}((q_{dtss}, h), e, x^{cont}) : \qquad e \geq h \tag{5.34}$$

La función de transición interna del modelo DEV&DESS se construye, a partir de la función de transición interna del modelo DTSS, de la forma siguiente:

$$\delta_{int}((q_{dtss}, h), e, x^{cont}) = (\delta_{dtss}(q_{dtss}, x_{dtss}), h) \tag{5.35}$$

La función de salida discreta se construye de la forma siguiente:

$$\lambda^{discr}((q_{dtss}, h), e, x^{cont}) = \begin{cases} \lambda_{dtss}(q_{dtss}, x_{dtss}) & \text{para sistemas de tipo Mealy} \\ \lambda_{dtss}(q_{dtss}) & \text{para sistemas de tipo Moore} \end{cases} \tag{5.36}$$

La función para el cálculo de las derivadas, f, y la función de salida continua, λ^{cont}, son omitidas:

$$f \;:\; Q \times X^{cont} \to \{\} \tag{5.37}$$

$$\lambda^{cont} \;:\; Q \times X^{cont} \to \{\} \tag{5.38}$$

5.3.3. Formalismo DEVS clásico

De forma similar a como se ha hecho con el formalismo DTSS, a continuación describimos un procedimiento para describir un modelo DEVS clásico empleando el formalismo DEV&DESS. En particular, empleamos la variable e, que es el tiempo transcurrido desde el último evento y la condición de evento, que define la condición de disparo de los eventos en el estado, para planificar los eventos internos. La condición de evento pasa de valer *false* a valer *true* en el instante en que el reloj de la simulación alcanza el instante en que está planificado el siguiente evento interno del modelo DEVS.

El modelo DEVS

$$\text{DEVS} = \langle X_{devs}, S_{devs}, Y_{devs}, \delta_{int,devs}, \delta_{ext,devs}, \lambda_{devs}, ta_{devs} \rangle \tag{5.39}$$

puede expresarse empleando el modelo DEV&DESS siguiente

$$\text{DEV\&DESS} = \left\langle X^{\text{discr}}, X^{\text{cont}}, Y^{\text{discr}}, Y^{\text{cont}}, S^{\text{discr}}, S^{\text{cont}}, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{\text{discr}}, f, \lambda^{\text{cont}} \right\rangle \tag{5.40}$$

de la forma siguiente:

$$\begin{array}{ll} X^{\text{discr}} = X_{devs} & X^{\text{cont}} = \{\} \\ Y^{\text{discr}} = Y_{devs} & Y^{\text{cont}} = \{\} \\ S^{\text{discr}} = Q_{devs} & S^{\text{cont}} = \{\} \end{array} \tag{5.41}$$

La condición de evento, C_{int}, del modelo DEV&DESS se define de la forma siguiente:

$$C_{int}(s_{devs}, e): \qquad e \geq ta_{devs}(s_{devs}) \tag{5.42}$$

La función de transición interna del modelo DEV&DESS, $\delta_{int} : Q \times X^{cont} \rightarrow S$, se construye a partir de la función de transición interna del modelo DEVS:

$$\delta_{int}((s_{devs}, e)) = \delta_{int,devs}(s_{devs}) \tag{5.43}$$

La función para el cálculo de las derivadas, f, y la función de salida continua, λ^{cont}, son omitidas:

$$f \;:\; Q \times X^{cont} \rightarrow \{\} \tag{5.44}$$

$$\lambda^{cont} \;:\; Q \times X^{cont} \rightarrow \{\} \tag{5.45}$$

5.4. MODELADO MULTIFORMALISMO

Como hemos visto en la sección anterior, los modelos DESS son un caso especial de modelo DEV&DESS. También hemos visto cómo embeber DEVS o DTSS en DEV&DESS. Por tanto, podemos usar un formalismo básico (DESS, DEVS y DTSS) allí donde usemos el formalismo DEV&DESS y, sacando partido de esta capacidad, si somos capaces de construir modelos compuestos DEV&DESS, podemos construir modelos compuestos multiformalismo, cuyos componentes representen modelos de tiempo continuo, de tiempo discreto y de eventos discretos.

En esta sección se analizará cómo se interpreta la conexión entre componentes de diferente formalismo. Para ello, es fundamental la interpretación del acoplo entre las salidas de tiempo discreto (eventos) y las entradas de tiempo continuo (variables de tiempo continuo), y viceversa.

En la Figura 5.5 se muestran las trayectorias típicas de salida a las que da lugar cada uno de los tres formalismos básicos. Obsérvese que una trayectoria de salida constante a tramos (véase la Figura 5.5a) es un caso particular de salida de tiempo continuo (véase la Figura 5.5b) del formalismo DESS. Las trayectorias E/S de los formalismos DEVS y DTSS son eventos (véanse las Figuras 5.5c & 5.5d). En el caso del formalismo DTSS, el intervalo de tiempo entre eventos consecutivos es un parámetro h característico del modelo.

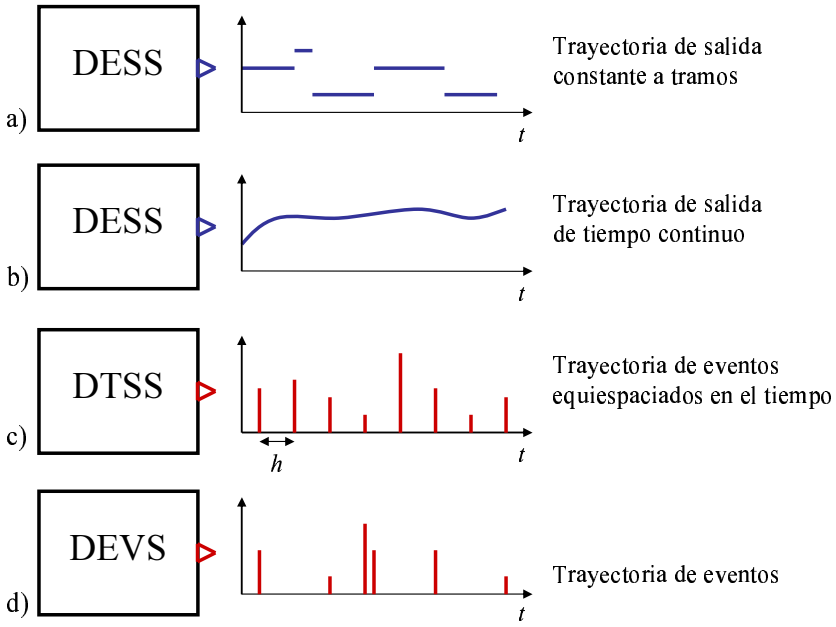

Figura 5.5: Trayectorias de salida de los formalismos básicos.

Puesto que las trayectorias de eventos pueden ser traducidas a trayectorias constantes a tramos y viceversa, tenemos una forma de interpretar la conexión entre salidas discretas y entradas continuas, y también la conexión entre salidas continuas constantes a tramos y entradas discretas. A continuación, vamos a analizar los diferentes tipos de conexión entre componentes de diferente formalismo.

– **DTSS → DEVS.** Al conectar un puerto de salida de un modelo DTSS con un puerto de entrada de un modelo DEVS, los eventos de salida del modelo DTSS, generados a intervalos regulares de tiempo, son interpretados como eventos de

entrada por el modelo DEVS. En la Figura 5.6 se muestra una trayectoria típica de salida del modelo DTSS, la cual puede ser interpretada tal cual por el modelo DEVS.

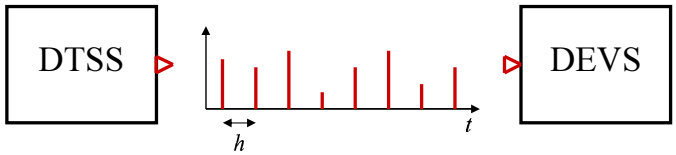

Figura 5.6: Conexión DTSS → DEVS.

– **DEVS → DTSS.** En general, los eventos de salida del modelo DEVS no se producen de manera periódica. La forma de traducir estos eventos de salida a una secuencia de entrada eventos equiespaciados en el tiempo es la siguiente: el evento de entrada se construye a partir del último evento de salida. En la Figura 5.7 se han dibujado de color rojo los eventos de salida del modelo DEVS y en color azul los eventos de entrada correspondientes del modelo DTSS.

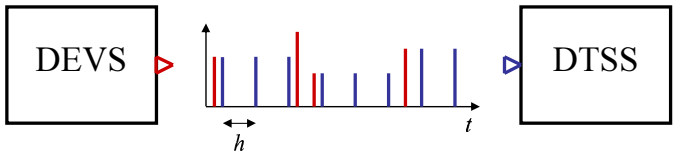

Figura 5.7: Conexión DEVS → DTSS.

– **DEVS → DESS.** Se considera que los eventos de salida del modelo DEVS definen los cambios en la trayectoria de entrada, constante a tramos, del modelo DESS. En la Figura 5.8 se muestra un ejemplo. En rojo están dibujados los eventos de salida y en azul la correspondiente trayectoria de tiempo continuo de entrada.

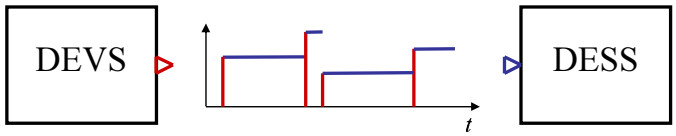

Figura 5.8: Conexión DEVS → DESS.

– **DESS → DEVS.** La salida del modelo DESS es una trayectoria de tiempo continuo constante a tramos. Los cambios en los valores constantes de la trayectoria se interpretan como eventos de entrada al modelo DEVS. En la Figura 5.9 se muestra la trayectoria constante a tramos de salida en color azul y la correspondiente trayectoria de eventos en rojo.

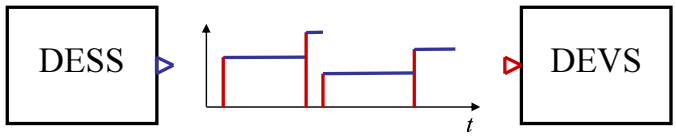

Figura 5.9: Conexión DESS → DEVS.

– **DTSS → DESS.** Los eventos equiespaciados en el tiempo que constituyen la trayectoria de salida del modelo DTSS son interpretados como una trayectoria constante a tramos por el modelo DESS. El valor del tramo constante de la trayectoria de entrada es igual al del último evento de salida. Se muestra un ejemplo en la Figura 5.10, donde los eventos son representados en rojo y la trayectoria de tiempo continuo en azul.

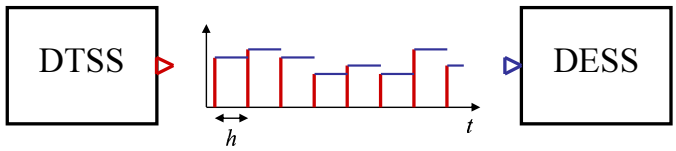

Figura 5.10: Conexión DTSS → DESS.

– **DESS → DTSS.** La trayectoria de salida de tiempo continuo del modelo DESS es muestreada, con periodo h, a fin de convertirla en una trayectoria de eventos equiespaciados en el tiempo. En la Figura 5.11 se muestra la trayectoria de salida de tiempo continuo, dibujada en azul, y los eventos, dibujados en rojo, que son obtenidos de muestrear la trayectoria continua con periodo h.

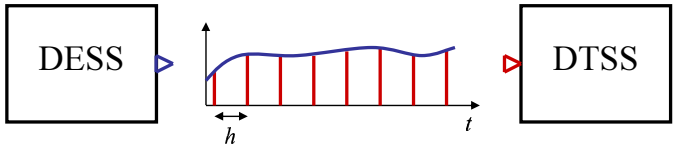

Figura 5.11: Conexión DESS → DTSS.

5.5. TRATAMIENTO DE LOS EVENTOS EN EL ESTADO

El estado de un sistema híbrido evoluciona mediante el cambio continuo de sus estados continuos o mediante cambios instantáneos en su estado total, continuo y discreto, llamados eventos.

Durante la ejecución de la simulación de un modelo híbrido se realiza, o bien una simulación enteramente de eventos discretos o bien una simulación enteramente continua, ya que la ejecución de una simulación simultáneamente continua y discreta no existe. Por ello, el simulador de modelos híbridos debe estar compuesto de un simulador de eventos discretos, un simulador de sistemas continuos, y algoritmos describiendo las actividades a realizar cuando se pasa de la simulación de eventos discretos a la de sistemas continuos y viceversa.

La principal extensión necesaria para simular modelos híbridos es el tratamiento de los eventos en el estado, que incluye:

1. La detección de los eventos en el estado. Se realiza vigilando, durante la simulación de la parte continua del modelo, las funciones de cruce por cero asociadas a los eventos en el estado. Cuando una función de cruce pasa a valer cero o corta el cero, se suspende la solución del problema continuo y se realiza el tratamiento del evento correspondiente.

   En la Figura 5.12 se muestra la evolución temporal de una función de cruce por cero. La parte continua del modelo se calcula en el instante t_i, incluyendo la evaluación de la función de cruce, cuyo valor positivo indica que no se satisfacen las condiciones para el disparo del evento. A continuación, se avanza un paso de tiempo, cuya longitud $t_{i+1}-t_i$ está determinada por el algoritmo de integración. Se evalúa la parte continua del modelo en el instante t_{i+1}, obteniéndose que la función de cruce tiene un valor negativo. Esto significa que en algún instante entre t_i y t_{i+1} se ha satisfecho la condición de disparo del evento, es decir, la función de cruce se ha hecho cero.

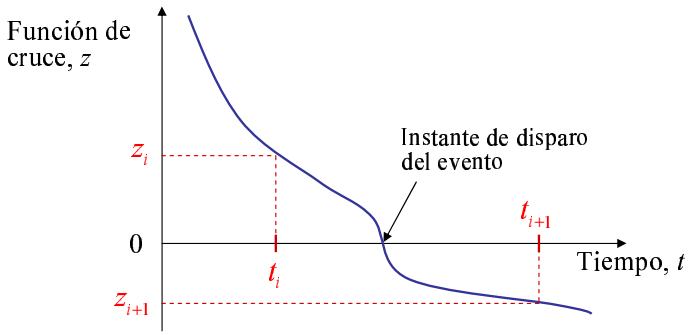

Figura 5.12: Detección de evento en el estado mediante función de cruce.

2. Se determina, con una precisión inferior a una determinada, el instante de disparo del evento, que es aquel en el cual la función de cruce pasa a valer cero o corta el cero. Para ello, pueden usarse varios métodos, entre los cuales se encuentra el de la bisección. Este método consiste en ir dividiendo el intervalo por la mitad y determinando en cuál de las dos mitades se produce el cruce por cero. La mitad en la que se produce el cruce se divide a su vez por la mitad y se determina en cuál de las dos mitades se produce el cruce, y así sucesivamente.

3. Se planifican los eventos para su ejecución.

4. La ejecución de los eventos es idéntica a la ejecución de los eventos internos en DEVS.

5.6. SIMULADOR PARA MODELOS DESS

El problema fundamental de la simulación de modelos de tiempo continuo es cómo calcular un comportamiento de tiempo continuo mediante pasos de tiempo discretos. La principal forma de realizarlo es mediante el empleo de los métodos de integración numérica. Los métodos de integración numérica pueden clasificarse en dos grupos: *causales* y *no causales*.

5.6.1. Métodos numéricos de integración causales

Los métodos causales usan el valor de los estados en el pasado y en el presente para calcular el valor del estado en el instante de tiempo futuro. Normalmente, los

métodos de integración causales calculan el valor del estado en el instante t_{i+1} como una combinación lineal de los valores del estado y de la derivada en m instantes de tiempo anteriores, t_{i-m}, t_{i-m+1}, ..., t_i. Los coeficientes de la combinación lineal son escogidos de tal manera que se minimice el error en la estimación. Un ejemplo de método causal es el método de integración de Euler explícito, descrito en la Sección 2.3.4.

La forma general de un método causal que usa m valores pasados de la derivada y del estado para calcular el estado en el instante de tiempo t_{i+1} es la siguiente:

$$\begin{aligned} q(t_{i+1}) = \text{MétodoCausal}(\quad & q(t_{i-m}), \ldots, q(t_i), \\ & r(t_{i-m}), \ldots, r(t_i), \\ & x(t_{i-m}), \ldots, x(t_i), \\ & \Delta t) \end{aligned} \tag{5.46}$$

donde q representa las variables de estado, r las derivadas de las variables de estado, x las entradas y Δt el tamaño del paso de integración ($t_{i+1} = t_i + \Delta t$).

Un problema general en los métodos de integración causales es el arranque. Consiste en la determinación de la derivada y el estado en los instantes de tiempo t_{-m}, t_{-m+1}, ..., t_0, que deben conocerse para poder calcular $q(t_1)$.

Una solución al *problema del arranque* es usar métodos causales de menor orden en el arranque. Es decir, en el primer paso de integración, sólo se dispone del valor inicial, con lo cual se usa un método de orden uno. En los siguientes pasos de integración, el orden del método va aumentándose hasta alcanzar el orden m.

Otra solución al problema del arranque es emplear un método de integración diferente durante la fase de arranque, por ejemplo, un método no causal como los descritos en la Sección 5.6.2. Un algoritmo para la simulación de modelos DESS atómicos, empleando un método de integración causal, es el siguiente:

```
Dess-causal-simulator
  variables:
    DESS = (X, Y, Q, f, lambda)   // Modelo asociado
    [q(t_{i-m}), ..., q(t_{i})]   // Vector de valores pasados del estado
    [r(t_{i-m}), ..., r(t_{i})]   // Vector de valores pasados de la derivada
    [x(t_{i-m}), ..., x(t_{i})]   // Vector de valores pasados de la entrada
    h                             // Paso de integración
  when recibe mensaje-i (i,t_{i}) en el instante t_{i} {
    inicializa [q(t_{i-m}), ..., q(t_{i})],
               [r(t_{i-m}), ..., r(t_{i})],
               [x(t_{i-m}), ..., x(t_{i})]
```

```
                mediante el procedimiento del método de integración
    }
  when recibe mensaje-* (*,t_{i}) en el instante t_{i} {
    y = lambda( q(t_{i}) )
    envia mensaje-y (y,t_{i}) a parent
    }
  when recibe mensaje-x (x, t_{i}) con valor en entrada x {
    x(t_{i}) = x
    q(t_{i+1}) = MétodoCausal ( q(t_{i-m}), ..., q(t_{i}),
                                r(t_{i-m}), ..., r(t_{i}),
                                x(t_{i-m}), ..., x(t_{i}),
                                h )
    }
end Dess-causal-simulator
```

Cuando el algoritmo recibe un mensaje de inicialización en el instante t_i, el algoritmo resuelve el problema del arranque, es decir, calcula el valor de los vectores $[q(t_{i-m}), \ldots, q(t_i)]$, $[r(t_{i-m}), \ldots, r(t_i)]$ y $[x(t_{i-m}), \ldots, x(t_i)]$. El método empleado para ello depende del algoritmo de integración. Obsérvese que, una vez calculados estos tres vectores, se dispone de los datos necesarios para calcular $q(t_{i+1})$ empleando el método de integración.

Los sucesivos instantes de tiempo en que se evalúa el modelo vienen determinados por el valor inicial del tiempo y por el tamaño del paso de integración. En cada instante de evaluación, t_i, el algoritmo recibe dos mensajes:

1. Un mensaje-*. En respuesta a este mensaje, el algoritmo calcula el valor de la salida en el instante t_i. Es decir: $y_i = \lambda(q(t_i))$.

2. Un mensaje-x, en el cual recibe el valor de la entrada en el instante t_i, es decir, $x(t_i)$. En respuesta a este mensaje, el algoritmo calcula el estado en el instante t_{i+1}, es decir, $q(t_{i+1})$.

5.6.2. Métodos numéricos de integración no causales

Para calcular el estado en el instante de tiempo futuro, t_{i+1}, los métodos no causales usan, además de valores de los estados y de las derivadas en el pasado y en el presente, también estimaciones del valor del estado, las derivadas y las entradas en instantes de tiempo posteriores al actual, t_i. Los métodos de Euler-Richardson, de RK-4 y RKF-4,5 descritos en la Sección 2.3.4 son ejemplos de métodos no causales.

Los métodos no causales tienen dos fases. En la fase *predictor*, se realiza una estimación de los valores futuros. En la fase *corrector*, finalmente se determina el valor del estado en el instante de tiempo futuro.

Al igual que en los métodos causales, para aplicar un método no causal deben almacenarse los m valores pasados del estado $(q(t_{i-m}), \ldots, q(t_i))$, de las derivadas $(r(t_{i-m}), \ldots, r(t_i))$ y de las entradas $(x(t_{i-m}), \ldots, x(t_i))$. Además, debe almacenarse el conjunto de valores predichos calculados en la fase predictor, $q'(1)$, $\ldots$, $q'(n)$.

En la fase predictor se calcula el valor del estado en n instantes futuros:

$$\begin{aligned} q'(k+1) = \text{k-ésimoPredictor}(\quad & q(t_{i-m}), \ldots, q(t_i), \quad q'(1), \ldots, q'(k), \\ & r(t_{i-m}), \ldots, r(t_i), \quad r'(1), \ldots, r'(k), \\ & x(t_{i-m}), \ldots, x(t_i), \quad x'(1), \ldots, x'(k), \\ & \Delta t) \end{aligned} \tag{5.47}$$

para $k : 0, \ldots, n-1$. Los valores calculados en la fase predictor se usan en la fase corrector para calcular el valor del estado en el instante t_{i+1}.

$$\begin{aligned} q(t_{i+1}) = \text{Corrector}(\quad & q(t_{i-m}), \ldots, q(t_i), \quad q'(1), \ldots, q'(n), \\ & r(t_{i-m}), \ldots, r(t_i), \quad r'(1), \ldots, r'(n), \\ & x(t_{i-m}), \ldots, x(t_i), \quad x'(1), \ldots, x'(n), \\ & \Delta t) \end{aligned} \tag{5.48}$$

A continuación, se muestra un algoritmo para la simulación de modelos DESS atómicos, empleando un método de integración no causal.

```
Dess-no-causal-simulator
  variables:
    DESS = (X, Y, Q, f, lambda)   // Modelo asociado
    [q(t_{i-m}), ..., q(t_{i})]   // Vector de valores pasados del estado
    [r(t_{i-m}), ..., r(t_{i})]   // Vector de valores pasados de la derivada
    [x(t_{i-m}), ..., x(t_{i})]   // Vector de valores pasados de la entrada
    [q'(1), ..., q'(n)]           // Vector de valores predichos del estado
    [r'(1), ..., r'(n)]           // Vector de valores predichos de la derivada
    [x'(1), ..., x'(n)]           // Vector de valores predichos de la entrada
    h                             // Paso de integración
    k                             // Indicador de la fase del integrador
  when recibe mensaje-i (i,t_{i}) en el instante t_{i} {
    inicializa [q(t_{i-m}), ..., q(t_{i})],
               [r(t_{i-m}), ..., r(t_{i})],
               [x(t_{i-m}), ..., x(t_{i})]
```

```
                mediante el procedimiento del método de integración
    k = 0
    }
  when recibe mensaje-* (*,t_{i}) en el instante t_{i} {
    if k = 0 then y = lambda( q(t_{i}) )
             else y = lambda( q'(k) )
    envia mensaje-y (y,t_{i}) a parent
    }
  when recibe mensaje-x (x, t_{i}) con valor en entrada x {
    if k = 0 then x(t_{i}) = x
             else x'(k)    = x
    k = (k + 1) mod (n+1)     // k: 0,1,...,n,0,1,...,n,0,1, etc.
    if k > 0
    then
      q'(k) =k-ésimoPredictor( q(t_{i-m}), ..., q(t_i), q'(1), ..., q'(k-1),
                               r(t_{i-m}), ..., r(t_i), r'(1), ..., r'(k-1),
                               x(t_{i-m}), ..., x(t_i), x'(1), ..., x'(k-1),
                               h )
    else
      q(t_{i+1}) = Corrector(  q(t_{i-m}), ..., q(t_i), q'(1), ..., q'(n),
                               r(t_{i-m}), ..., r(t_i), r'(1), ..., r'(n),
                               x(t_{i-m}), ..., x(t_i), x'(1), ..., x'(n),
                               h )
    }
end Dess-no-causal-simulator
```

El algoritmo recibe, en cada instante de evaluación, $n+1$ mensajes-* y $n+1$ mensajes-x. En el algoritmo se emplea el contador k para distinguir entre las fases del predictor y el corrector. En función del valor del contador k:

- La salida se calcula empleando el estado actual o el estado estimado. Cuando $k = 0$, se calcula la salida correspondiente al estado actual: $y_i = \lambda(q(t_i))$. Para valores $k \neq 0$, se calcula la salida correspondiente a la estimación k-ésima $q'(k)$ del estado futuro: $y = \lambda(q'(k))$.

- Una entrada recibida en un mensaje-x se interpreta como la k-ésima estimación o como el valor actual de la entrada. Si se recibe un mensaje-x y $k = 0$, entonces el valor de x corresponde con el valor actual de la entrada, $x(t_i)$. En caso contrario, la entrada representa una estimación y su valor se almacena en $x'(k)$.

- Se decide si debe realizarse la k-ésima predicción o la corrección final.

5.7. EJERCICIOS DE AUTOCOMPROBACIÓN

Ejercicio 5.1

Modifique el modelo del proceso de llenado de barriles descrito en el Tema 5 del texto base de teoría, de modo que el modelo detecte cuándo el valor del flujo de líquido *flujoIn* es inferior a un determinado valor umbral y detenga el proceso de llenado, generando un evento de salida que alerte de dicha situación. Puede realizar todas las hipótesis de modelado adicionales que desee.

Ejercicio 5.2

Indiqué como podría realizarse la conexión entre dos modelos DTSS con diferente tamaño del paso, h. Considere el uso de una función promediadora en el caso de la conexión entre una salida rápida (h pequeño) y una entrada lenta (h grande).

Ejercicio 5.3

Escriba el modelo del sistema descrito a continuación, aplicando el formalismo DEV&DESS. El sistema consta de un depósito para el almacenamiento de líquido, una bomba y dos válvulas (véase la Figura 5.13). Además tiene un sensor/actuador, que manipula la válvula 2, y un sensor de nivel.

La función de la bomba es introducir líquido en el depósito. El voltaje de entrada a la bomba es una variable de tiempo continuo, v_{bomba}. El flujo de líquido que sale de la bomba y entra en el depósito (F_{bomba}) es proporcional al voltaje aplicado a la bomba (v_{bomba}). La constante de proporcionalidad (K_{bomba}) es un parámetro.

$$F_{bomba} = K_{bomba} \cdot v_{bomba} \tag{5.49}$$

El sistema tiene dos válvulas. Cada una de las válvulas puede encontrarse en uno de dos modos de funcionamiento: {abierta, cerrada}.

– La apertura y cierre de la válvula 1 es accionada a través de una de las entradas del sistema: la entrada de eventos discretos *On/Off*. Los eventos que llegan a esa entrada sólo pueden tomar dos valores: $\{0, 1\}$. Cuando se recibe un evento de valor 0, la válvula se cierra: $F_{out1} = 0$. Por el contrario, cuando se recibe un

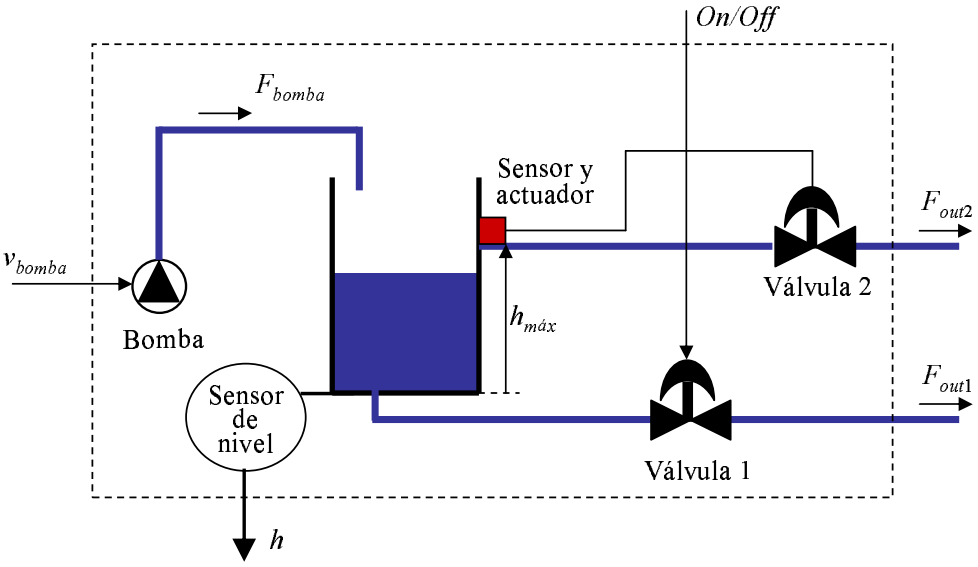

Figura 5.13: Representación esquemática del sistema.

evento de valor 1, la válvula se abre y el flujo a su través es: $F_{out1} = K_1 \cdot h$, donde K_1 es un parámetro y h es la altura del líquido contenido en el depósito.

– La apertura de la válvula 2 es accionada mediante un actuador, que está conectado a un sensor que detecta la presencia de líquido. Tanto el sensor como el actuador son parte del sistema. Esta válvula hace las funciones de válvula de seguridad, abriéndose cuando el nivel de líquido en el depósito supera un cierto valor umbral (h_{max}).

  Así pues, cuando el nivel de líquido h es mayor que un cierto valor umbral h_{max}, la válvula 2 se abre y el flujo a su través es: $F_{out2} = K_2 \cdot (h - h_{max})$, donde K_2 es un parámetro. Mientras el nivel de líquido sea menor o igual que h_{max}, la válvula 2 está cerrada: $F_{out2} = 0$.

La variación en el nivel del líquido contenido en el depósito, h, depende de los flujos de entrada y salida. La constante de proporcionalidad C es un parámetro del modelo.

$$C \cdot \frac{dh}{dt} = F_{bomba} - F_{out1} - F_{out2} \tag{5.50}$$

En la base del depósito está conectado un sensor de nivel, que proporciona el valor actual del nivel del líquido contenido en el depósito (h).

El modelo tiene tres variables de salida de tiempo continuo: la altura del líquido (h), el flujo a través de la válvula 1 (F_{out1}) y el flujo a través de la válvula 2 (F_{out2}).

Ejercicio 5.4

Escriba el modelo del controlador descrito a continuación, aplicando el formalismo DEV&DESS. En la Figura 5.14 se muestra un esquema de la interfaz del controlador. Tiene dos entradas, una de tiempo continuo (h) y otra de eventos discretos (h^{ref}), y dos salidas, una de tiempo continuo (v_{bomba}) y otra de eventos discretos (*On/Off*).

Figura 5.14: Representación esquemática de la interfaz del controlador.

La finalidad del controlador es conseguir, una vez conectado al sistema descrito en el ejercicio anterior, que la altura de líquido en el depósito (h) sea igual a su valor de consigna (h^{ref}). Para ello, el controlador manipula el valor del voltaje aplicado a la (v_{bomba}) y la apertura/cierre de la válvula 1 (*On/Off*).

Los eventos recibidos en la entrada h^{ref} pueden tomar valor real positivo y representan el valor que se desea que tenga la altura de líquido (valor de referencia o de consigna).

La generación de eventos en la salida *On/Off* se realiza de la forma siguiente:

- Cuando el valor actual de la altura (h) se hace mayor que el valor de referencia (h^{ref}), entonces se genera un evento de valor 1 en la salida *On/Off*. Con ello se abre la válvula 1, lo cual hace que comience a vaciarse el depósito.
- Cuando el valor actual de la altura se hace menor que el valor de consigna, se genera un evento de valor cero en la salida *On/Off*, cuyo efecto es cerrar la válvula 1.

La salida de tiempo continuo v_{bomba} se calcula de la forma siguiente:

$$v_{bomba} = \begin{cases} 0 & \text{si } h \geq h^{ref} \\ K_C \cdot \left(h^{ref} - h\right) & \text{en caso contrario} \end{cases} \tag{5.51}$$

Es decir, mientras el nivel de líquido es menor que el nivel de referencia ($h < h^{ref}$), el voltaje aplicado a la bomba será proporcional a la diferencia $h^{ref} - h$. Con ello se consigue introducir líquido en el depósito y reducir la diferencia entre el nivel actual y el deseado. Por el contrario, mientras el nivel actual sea mayor o igual al deseado ($h \geq h^{ref}$), se aplica un voltaje nulo a la bomba, con lo cual no se introduce líquido en el depósito.

Ejercicio 5.5

Describa el modelo compuesto mostrado en la Figura 5.15, formado por la conexión del sistema del Ejercicio 5.3 y el controlador del Ejercicio 5.4, aplicando el formalismo DEV&DESS.

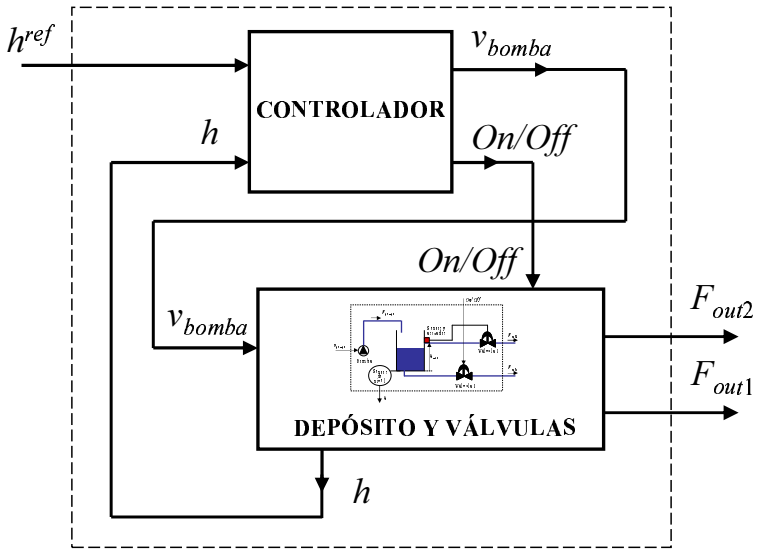

Figura 5.15: Representación esquemática del sistema controlado.

5.8. SOLUCIONES A LOS EJERCICIOS

Solución al Ejercicio 5.1

Existen varias formas de modificar el modelo del proceso de llenado de barriles. Una posibilidad es definir un nuevo puerto de salida discreto, que llamaremos *valorFlujo* (véase la Figura 5.16). Los eventos generados a través de este puerto pueden tomar dos valores: {“flujoBajo”, “flujoNormal”}.

- Cuando *flujoIn* es mayor o igual que el valor umbral y pasa a ser menor que el valor umbral, entonces se genera un evento de valor “flujoBajo”.
- Cuando *flujoIn* es menor o igual que el valor umbral y pasa a ser mayor, entonces se genera un evento de valor “flujoNormal”.

Figura 5.16: Modelo del proceso de llenado de barriles.

La interfaz del modelo está definida por los siguientes elementos de la tupla:

$$
\begin{aligned}
X^{\text{cont}} &= \{flujoIn \mid flujoIn \in \mathbb{R}\} \\
X^{\text{discr}} &= \{on/off \mid on/off \in \{\text{“on”}, \text{“off”}\}\} \\
Y^{\text{cont}} &= \{cout \mid cout \in \mathbb{R}\} \\
Y^{\text{discr}} &= \{barril \mid barril \in \{\text{“barrilDe10litros”}\}, \\
&\quad\ valorFlujo \mid valorFlujo \in \{\text{“flujoBajo”, “flujoNormal”}\}\}
\end{aligned}
$$

El estado del sistema es descrito por tres variables de estado: *contenido*, *valvula* y *flujo*. La variable de estado *flujo* puede tomar dos valores: {“bajo”,“normal”}. Describe el modo en el cual se encuentra el sistema: flujo de entrada (*flujoIn*) por debajo del valor umbral o flujo normal.

$$\begin{aligned} S^{\text{cont}} &= \{contenido \mid contenido \in \mathbb{R}\} \\ S^{\text{discr}} &= \{valvula \mid valvula \in \{\text{“abierta”}, \text{“cerrada”}\}, \\ &\quad\ flujo \mid flujo \in \{\text{“bajo”}, \text{“normal”}\}\} \end{aligned}$$

El modelo tiene las tres condiciones de evento siguientes:

– El volumen de líquido en el barril alcanza el valor 10 litros.

$$\begin{aligned} &C_{int,1}((contenido, valvula, flujo), e, flujoIn) : \\ &\quad contenido > 10 \end{aligned}$$

– El flujo de líquido pasa a ser menor que el valor umbral.

$$\begin{aligned} &C_{int,2}((contenido, valvula, flujo), e, flujoIn) : \\ &\quad flujo = \text{“normal”}\ y\ flujoIn < umbral \end{aligned}$$

– El flujo de líquido pasa a ser mayor o igual que el valor umbral.

$$\begin{aligned} &C_{int,3}((contenido, valvula, flujo), e, flujoIn) : \\ &\quad flujo = \text{“bajo”}\ y\ flujoIn \geq umbral \end{aligned}$$

Cuando se dispara una o varias de las condiciones de evento, se evalúa la función de transición interna con el fin de calcular el nuevo estado. Cada condición de evento tiene asociado el siguiente cambio en las variables de estado:

$$\begin{aligned} &\delta_{int}((contenido, valvula, flujo), e, flujoIn) : \\ &\quad contenido = 0 \qquad \text{si } C_{int,1} \\ &\quad flujo = \text{“bajo”} \qquad \text{si } C_{int,2} \\ &\quad flujo = \text{“normal”} \quad \text{si } C_{int,3} \end{aligned}$$

Como parte de las acciones asociadas a una transición interna, se evalúa la función de salida discreta, λ^{discr}, con el fin de calcular el valor del evento de salida. El valor del evento y del puerto de salida depende de la condición de evento o las condiciones de evento que se hayan activado. La función de salida de la parte discreta del modelo es la siguiente:

$\lambda^{discr}((contenido, valvula, flujo), e, flujoIn)$:

Si $C_{int,1}$, genera evento de valor "barrilDe10litros" en el puerto $barril$
Si $C_{int,2}$, genera evento de valor "flujoBajo" en el puerto $valorFlujo$
Si $C_{int,3}$, genera evento de valor "flujoNormal" en el puerto $valorFlujo$

La función f, de cálculo de las derivadas de las variables de estado de tiempo continuo, es la siguiente:

$$f((contenido, valvula, flujo), e, flujoIn) :$$
$$\frac{dcontenido}{dt} = \begin{cases} 0 & \text{Si } valvula = \text{"cerrada" o } flujo = \text{"bajo"} \\ flujoIn & \text{En caso contrario} \end{cases}$$

La función de salida de la parte continua, λ^{cont}, es la misma que la del modelo explicado en el texto base de teoría: permite calcular la variable de salida *cout*, cuyo valor es igual en todo momento al valor de la variable de estado *contenido*.

Solución al Ejercicio 5.2

La trayectoria de entrada o salida de un modelo DTSS es un conjunto de eventos equiespaciados en el tiempo. Sea h el intervalo de tiempo entre dos eventos consecutivos.

Cuando se conecta un modelo DTSS con salida rápida (h_1 pequeño) a otro modelo DTSS con entrada lenta (h_2 grande), puede construirse el valor del evento de entrada promediando los eventos de salida que se han producido durante el intervalo h_2.

En el caso contrario, salida lenta (h_1 grande) y entrada rápida (h_2 pequeña), la conexión sería similar al caso DEVS $\rightarrow$ DTSS, donde el evento de entrada se construye a partir del último evento de salida.

Solución al Ejercicio 5.3

La interfaz del modelo está compuesta por dos entradas, una de tiempo continuo y otra de eventos discretos, y por tres salidas de tiempo continuo. Está definida por los siguientes elementos de la tupla:

$$
\begin{aligned}
X^{\text{cont}} &= \{vBomba \mid vBomba \in \mathbb{R}\} \\
X^{\text{discr}} &= \{on/off \mid on/off \in \{0,1\}\} \\
Y^{\text{cont}} &= \{Fout1, Fout2, h \mid Fout1, Fout2, h \in \mathbb{R}\}
\end{aligned}
$$

El estado del sistema está definido por tres variables de estado: la altura del líquido dentro del depósito (*altura*), que es una variable de tiempo continuo, y el modo de cada una de las dos válvulas (*valvula*1, *valvula*2). Estas dos variables de tiempo discreto pueden tomar dos posible valores: 0 (cerrada), 1 (abierta). Así pues, los elementos de la tupla que representan el estado son los siguientes:

$$
\begin{aligned}
S^{\text{cont}} &= \{altura \mid altura \in \mathbb{R}\} \\
S^{\text{discr}} &= \{valvula1, valvula2 \mid valvula1, valvula2 \in \{0,1\}\}
\end{aligned}
$$

La función de transición externa define cómo cambia el estado del modelo en función de los eventos de entrada. Los eventos de entrada en el puerto *on/off* determinan el modo de la válvula 1, es decir, el valor de la variable de estado *valvula1*.

$$
\begin{aligned}
&\delta_{ext}((altura, valvula1, valvula2), e, vBomba, on/off) : \\
&\quad valvula1 = on/off
\end{aligned}
$$

El modelo tiene dos condiciones de evento, las cuales determinan el modo de la válvula 2. Las condiciones de evento son las siguientes:

– El volumen de líquido pasa a ser mayor que h_{max}.

$$
\begin{aligned}
&C_{int,1}((altura, valvula1, valvula2), e, vBomba) : \\
&\quad valvula2 = 0 \text{ y } altura > h_{max}
\end{aligned}
$$

– El volumen de líquido se hace menor que h_{max}.

$$
\begin{aligned}
&C_{int,2}((altura, valvula1, valvula2), e, vBomba) : \\
&\quad valvula2 = 1 \text{ y } altura < h_{max}
\end{aligned}
$$

Cuando se dispara una de las condiciones de evento, se evalúa la función de transición interna con el fin de calcular el nuevo estado. Cada condición de evento tiene asociado el siguiente cambio en la variable de estado *valvula2*:

$$\begin{aligned}&\delta_{int}((altura, valvula1, valvula2), e, vBomba) :\\ &\quad valvula2 \;\; = 1 \quad \text{si } C_{int,1}\\ &\quad valvula2 \;\; = 0 \quad \text{si } C_{int,2}\end{aligned}$$

El modelo no tiene puertos de eventos discretos, con lo cual no es preciso definir la función de salida discreta, λ^{discr}.

La función f, de cálculo de las derivadas de las variables de estado de tiempo continuo, es la siguiente:

$$\begin{aligned}&f((altura, valvula1, valvula2), e, vBomba) :\\ &\quad C \cdot \tfrac{d\ altura}{dt} = K_{bomba} \cdot vBomba - valvula1 \cdot K_1 \cdot h - valvula2 \cdot K_2 \cdot (altura - h_{max})\end{aligned}$$

La función de salida de la parte continua, λ^{cont}, calcula las variables de salida de tiempo continuo.

$$\begin{aligned}&\lambda^{cont}((altura, valvula1, valvula2), e, vBomba) :\\ &\quad Fout1 \;\; = valvula1 \cdot K_1 \cdot h\\ &\quad Fout2 \;\; = valvula2 \cdot K_2 \cdot (h - h_{max})\\ &\quad h \qquad\;\; = altura\end{aligned} \tag{5.52}$$

Solución al Ejercicio 5.4

La interfaz del modelo está compuesta por dos entradas, una de tiempo continuo (h) y una de eventos discretos ($href$), y por dos salidas, una de tiempo continuo ($vBomba$) y una de eventos discretos (*On/Off*). Así pues, la interfaz está definida por los siguientes elementos de la tupla:

$$\begin{aligned}X^{\text{cont}} &= \{h \mid h \in \mathbb{R}\}\\ X^{\text{discr}} &= \{href \mid href \in \mathbb{R}\}\\ Y^{\text{cont}} &= \{vBomba \mid vBomba \in \mathbb{R}\}\\ Y^{\text{discr}} &= \{on/off \mid on/off \in \{0, 1\}\}\end{aligned}$$

El estado del sistema está definido por dos variables de estado:

– *h_mayorQue_href*, con dos posibles valores: {0,1}. Si la altura de líquido es mayor que el valor de referencia, entonces *h_mayorQue_href* vale 1. En caso contrario, vale 0.

– *alturaRef*, que almacena el valor de referencia para la altura de líquido. Este valor, de tipo real, coincide con el valor del último evento recibido en el puerto *href*.

Así pues, los elementos de la tupla que representan el estado son los siguientes:

$$\begin{aligned} S^{\text{cont}} &= \emptyset \\ S^{\text{discr}} &= \{h\_mayorQue\_href \mid h\_mayorQue\_href \in \{0,1\}, \\ &\quad alturaRef \mid alturaRef \in \mathbb{R}\} \end{aligned}$$

La función de transición externa define cómo cambia el estado cuando llegan eventos al puerto de entrada *href*. El valor del evento recibido se asigna a la variable de estado *alturaRef*.

$$\begin{aligned} &\delta_{ext}((h\_mayorQue\_href, alturaRef), e, href, h) : \\ &\quad alturaRef = href \end{aligned}$$

El modelo tiene dos condiciones de evento, las cuales determinan cuándo el nivel de líquido pasa a ser mayor que el valor de referencia, o bien cuando pasa a ser igual o menor.

– El volumen de líquido pasa a ser mayor que el valor de referencia.

$$\begin{aligned} &C_{int,1}((h\_mayorQue\_href, alturaRef), e, h) : \\ &\quad h\_mayorQue\_href = 0 \text{ y } h > alturaRef \end{aligned}$$

– El volumen de líquido se hace menor o igual que el valor de referencia.

$$\begin{aligned} &C_{int,2}((h\_mayorQue\_href, alturaRef), e, h) : \\ &\quad h\_mayorQue\_href = 1 \text{ y } h \leq alturaRef \end{aligned}$$

Cuando se dispara una de las condiciones de evento, se evalúa la función de transición interna con el fin de calcular el nuevo estado. Cada condición de evento tiene asociado el siguiente cambio en la variable de estado *h_mayorQue_href*:

$$\begin{aligned}\delta_{int}((h\_mayorQue\_href, alturaRef), e, h) :\\ h\_mayorQue\_href &= 1 \quad \text{si } C_{int,1}\\ h\_mayorQue\_href &= 0 \quad \text{si } C_{int,2}\end{aligned}$$

La función de salida discreta, λ^{discr}, determina qué eventos de salida se generan cuando se producen transiciones internas.

$$\begin{aligned}\lambda^{discr}((h\_mayorQue\_href, alturaRef), e, h) :\\ \text{Evento de valor 1 en el puerto } On/Off &\quad \text{si } C_{int,1}\\ \text{Evento de valor 0 en el puerto } On/Off &\quad \text{si } C_{int,2}\end{aligned}$$

Dado que el modelo no tiene variables de estado de tiempo continuo, no se define la función f.

La función de salida de la parte continua, λ^{cont}, calcula la variable de salida de tiempo continuo.

$$\begin{aligned}\lambda^{cont}((h\_mayorQue\_href, alturaRef), e, h) :\\ vBomba &= (1 - h\_mayorQue\_href) \cdot K_C \cdot (alturaRef - h)\end{aligned}$$

Solución al Ejercicio 5.5

El modelo compuesto mostrado en la Figura 5.15 puede representarse mediante la tupla siguiente:

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

donde:

$InPorts = \{\text{“}href\text{”}\}$	Puerto de entrada del modelo compuesto.
$X_{href} = \mathbb{R}$	Posibles valores de los eventos en el puerto “*href*” del modelo compuesto.
$OutPorts = \{\text{“}Fout1\text{”}, \text{“}Fout2\text{”}\}$	Puerto de salida del modelo compuesto.
$Y_{Fout1} = Y_{Fout2} = \mathbb{R}$	Posibles valores de los eventos en el puerto “*Fout1*” y “*Fout2*” del modelo compuesto.

$D = \{C, DV\}$	Conjunto de nombres de los componentes: controlador (C), depósito y válvulas (DV).
$M_C = M_{\text{controlador}}$ $M_{DV} = M_{\text{deposito y valvulas}}$	Tipo de los componentes.
$\text{EIC} = \{(N, \text{“href”}), (C, \text{“href”})\}$	Conexiones externas de entrada.
$\text{EOC} =$ $\{\ ((DV, \text{“Fout1”}), (N, \text{“Fout1”})),$ $((DV, \text{“Fout2”}), (N, \text{“Fout2”}))\ \}$	Conexiones externas de salida.
$\text{IC} =$ $\{\ ((DV, \text{“h”}), (C, \text{“h”})),$ $((C, \text{“vBomba”}), (DV, \text{“vBomba”})),$ $((C, \text{“On/Off”}), (DV, \text{“On/Off”}))\ \}$	Conexiones internas.

La función *select* establece la prioridad en caso de que los dos componentes tengan en el mismo instante una transición interna.

Tema 6

DEVSJAVA

OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema debería saber:

– Emplear las clases de DEVSJAVA para describir modelos atómicos y modulares sencillos, aplicando el formalismo DEVS clásico y paralelo.

6.1. INTRODUCCIÓN

En este tema se presenta un entorno en Java, denominado *DEVSJAVA* (Zeigler & Sarjoughian 2005), para el modelado y simulación de modelos DEVS. DEVSJAVA, que ha sido desarrollado en la Universidad de Arizona, es una propiedad intelectual de los *Regents of the State of Arizona*.

Puesto que DEVS es un formalismo público, existen otras implementaciones de estos conceptos, y cualquiera es libre de desarrollar su propio software basándose en la literatura de dominio público sobre el tema. Por ejemplo, dos entornos de simulación gratuitos basados en DEVS son JDEVS[1] y CD++[2], que han sido desarrollados en la Universidad de Córcega y Carleton, respectivamente. Estos entornos no emplean DEVSJAVA, sino sus propias implementaciones de DEVS.

6.2. PAQUETES EN DEVSJAVA 3.0

DEVSJAVA 3.0 es un entorno de simulación programado en lenguaje Java, que está especialmente diseñado para la descripción de modelos siguiendo el formalismo DEVS.

DEVSJAVA 3.0 se distribuye en un único fichero de tipo jar llamado *coreDEVS.jar*, en el cual no se proporciona el código fuente de las clases. Se recomienda usar JDK 1.4 con DEVSJAVA 3.0. Contiene los paquetes mostrados en la Figura 6.1:

`GenCol`	Contiene los servicios básicos para manipular conjuntos de entidades: contenedor, bolsa, conjunto, relación, función, listas ordenadas, etc.
`genDevs`	Contiene a su vez los paquetes `modeling`, `simulation` y `plots`, en los cuales pueden encontrarse clases para la descripción del modelo, la simulación y la representación de los resultados de la simulación. El paquete `modeling` contiene clases para la descripción de varios tipos de modelos, incluyendo DEVS atómico, acoplado y celular. Usando estos modelos predefinidos, pueden construirse modelos complejos de manera jerárquica y modular.

[1]Sitio web de JDEVS: http://spe.univ-corse.fr/filippiweb/appli/index.html
[2]Sitio web de CD++: http://www.sce.carleton.ca/faculty/wainer/wbgraf/

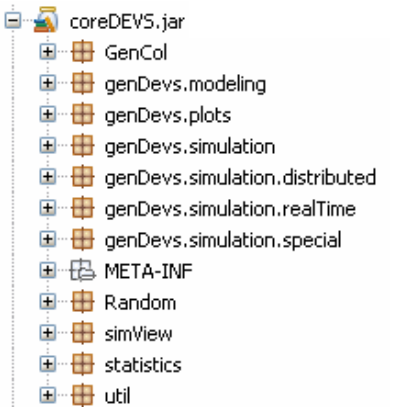

Figura 6.1: Paquetes de DEVSJAVA 3.0.

`simView`	Contiene una herramienta para la visualización de la estructura modular de los modelos y para el control de la ejecución de la simulación. Dentro de este paquete se encuentra la clase *SimView*, que contiene un método *main* que hay que ejecutar para arrancar la herramienta de visualización.
`statistics`	Contiene clases útiles para realizar análisis estadísticos.
`util`	Contiene clases de uso general.

Además del fichero *coreDEVS.jar*, la distribución de DEVSJAVA contiene un segundo fichero, llamado *IllustrationCode.zip*, que contiene una serie de ejemplos ilustrativos de los conceptos DEVS. Cada paquete está dedicado a una categoría diferente de modelos DEVS:

`SimpArc`	Contiene ejemplos básicos de modelos DEVS.
`oneDCellSpace`, `twoDCellSpace`	Contienen modelos de espacios celulares unidimensionales y bidimensionales respectivamente.
`Continuity`	Contiene modelos de sistemas continuos.
`variableStructure`	Contiene ejemplos de modelos con estructura variable.
`pulseModels`, `pulseExpFrames`	Contienen modelos y marcos experimentales respectivamente de sistemas de pulsos.

`Random`	Contiene ejemplos de modelos en los que se usan números aleatorios.
`Quantization`	Contiene modelos de sistemas cuantizados.

Se recomienda al alumno que en este punto cargue los ficheros *coreDEVS.jar* y *IllustrationCode.zip* en el entorno de desarrollo Java que habitualmente use y que inspeccione el contenido de ambos ficheros.

6.3. MODELOS SISO EN DEVS CLÁSICO

Por motivos didácticos, en esta primera sección dedicada al uso de DEVSJAVA se comienza explicando la descripción de los modelos DEVS clásico más sencillos, que son aquellos que tienen un único puerto de entrada y un único puerto de salida (modelos SISO), y en los cuales los eventos de entrada y salida toman valores reales. Estos modelos fueron explicados en la Sección 3.2.

El código Java de todos los modelos explicados en esta sección se encuentra en el paquete *SimpArc*, que está contenido en el fichero *IllustrationCode.zip*. Como se irá viendo, todos estos modelos DEVS SISO heredan de la clase *siso*.

6.3.1. Sistema pasivo

En la Sección 3.2.1 se describió el modelo de un sistema pasivo. Esto es, de un sistema que no responde con salidas a ninguna trayectoria de entrada y que se encuentra siempre en su único estado, que denominamos *passive* (pasivo). A continuación, se muestra el código de la clase *passive*, que describe el modelo DEVS pasivo.

```
package SimpArc;
public class passive extends siso {

  public passive()            {  super("passive"); }
  public passive(String name) {  super(name);      }

  public void initialize(){
     phase = "passive";
     sigma = INFINITY;
     super.initialize();
  }
```

```
  public void   Deltext(double e,double input) { passivate(); }

  public void   deltint()   { passivate(); }

  public double sisoOut()   { return 0; }

  public void   showState() { super.showState(); }
}
```

Puede observarse que el comportamiento del modelo se describe mediante la definición de los métodos siguientes:

`deltint`	Define la función de transición interna.
`Deltext`	Define la función de transición externa.
`sisoOut`	Define la función de salida, que genera un evento de salida de valor real justo antes de que se produzca la transición interna del estado.

En los métodos que definen las funciones de transición interna y externa, se hace una llamada al método *passivate*, que ya se encuentra predefinido y es empleado frecuentemente en la descripción de los modelos.

`passivate` Asigna al String *phase* el valor "passive" y al double *sigma* el valor INFINITY.

```
public void passivate() {
    passivateIn("passive");
}

public void passivateIn(String phase) {
    holdIn(phase, INFINITY);
}

public void holdIn(String phase, double sigma) {
   this.phase = phase;
   this.sigma = sigma;;
}
```

Obsérvese que hay dos variables (ya declaradas en las clases base, tales como *siso*, de DEVSJAVA) que se emplean como variables de estado en casi todos los modelos DEVS:

`sigma`	De tipo *double*, representa el tiempo que permanecerá el sistema en el estado actual en ausencia de eventos externos.
`phase`	De tipo *String*, almacena las diferentes fases del sistema.

6.3.2. Sistema acumulador

En la Sección 3.2.2 se describió el modelo de un sistema acumulador. El modelo tiene tres variables de estado:

1. *phase*, con valores {“passive”,“respond”}.
2. *sigma*, que puede tomar valores reales positivos.
3. *store*, que puede tomar valores reales diferentes de cero.

La fase “respond” es necesaria para indicar que una salida se encuentra en camino. El parámetro del modelo se llama *response_time* y corresponde con el tiempo de respuesta (Δ) descrito en la Sección 3.2.2.

A continuación, se muestra la parte más relevante del código Java de la clase *storage*. Obsérvese que las variables de estado *phase* y *sigma* ya han sido declaradas en la clase *siso*, con lo cual al definir la clase *storage* sólo es preciso declarar la variable de estado *store* y el parámetro *response_time*.

```
package SimpArc;
public class storage extends siso {

   protected double store;
   protected double response_time;

   public storage(String name,double Response_time){
      super(name);
      response_time = Response_time;
   }

   public void initialize(){
      phase = "passive";
      sigma = INFINITY;
      store = 0;
      response_time = 10;
      super.initialize();
   }
```

```
   public void  Deltext(double e,double input){
      Continue(e);
      if (input != 0)
         store = input;
      else
         holdIn("respond", response_time);
   }

   public void  deltint( ){
      passivate();
   }

   public double sisoOut(){
     if (phaseIs("respond")) return store; else return 0;
   }

   public void showState(){
     super.showState();
     System.out.println("store: " + store);
   }
}
```

En la programación de esta clase se han invocado dos métodos ya predefinidos en la clase *siso*:

`continue` Reduce *sigma* en *e*, donde *e* es el tiempo que ha transcurrido desde el anterior evento.

```
public void Continue(double e) {
   if (sigma < INFINITY) sigma = sigma - e;
}
```

`phaseIs` Devuelve *true* o *false* dependiendo de que el valor actual de *phase* coincida o no con el valor que se pasa como argumento al método.

```
public boolean phaseIs(String phase){
   return this.phase.equals(phase);
}
```

6.3.3. Generador de eventos

En la Sección 3.2.3 se describió el modelo de un generador de eventos. A continuación, se muestra su programación en DEVSJAVA.

```
package SimpArc;
public class generator extends siso {

   protected double period;

   public generator(String name,double Period){
      super(name);
      period = Period;
   }

   public void initialize(){
      phase = "active";
      sigma = period;
      super.initialize();
   }

   public void  deltint( ){
      holdIn("active",period);
      showState();
   }

   public double sisoOut(){
      return 1;
   }

   public void showState(){
      super.showState();
      System.out.println("period: " + period);
   }
}
```

Es posible visualizar la ejecución de este modelo empleando *simView*. Para arrancar *simView* debe ejecutarse (el método main de) la clase *SimView*, que se encuentra dentro del paquete *simView* de *coreDEVS.jar*.

En la GUI de *simView* hay un botón, *configure*, que debemos pulsar para introducir el path y el nombre de los paquetes con las clases de los modelos. En la Figura 6.2 se muestra el menú de configuración, en el que se indica el path del paquete (en el caso mostrado es "."), pero depende de dónde se hayan grabado los paquetes que contiene el fichero *IllustrationCode.zip*) y su nombre (*SimpArc*).

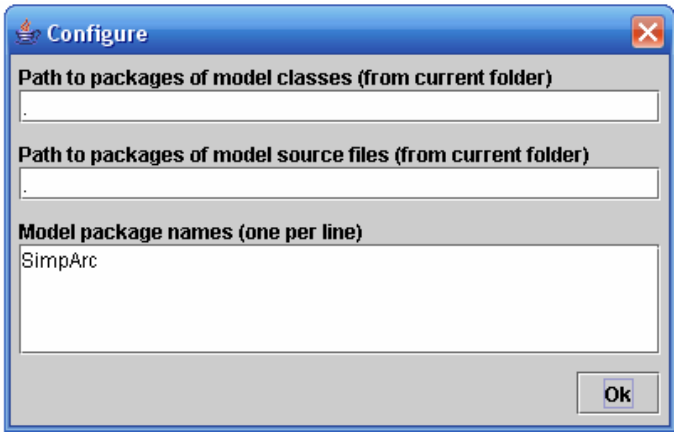

Figura 6.2: Menú de configuración de SimView.

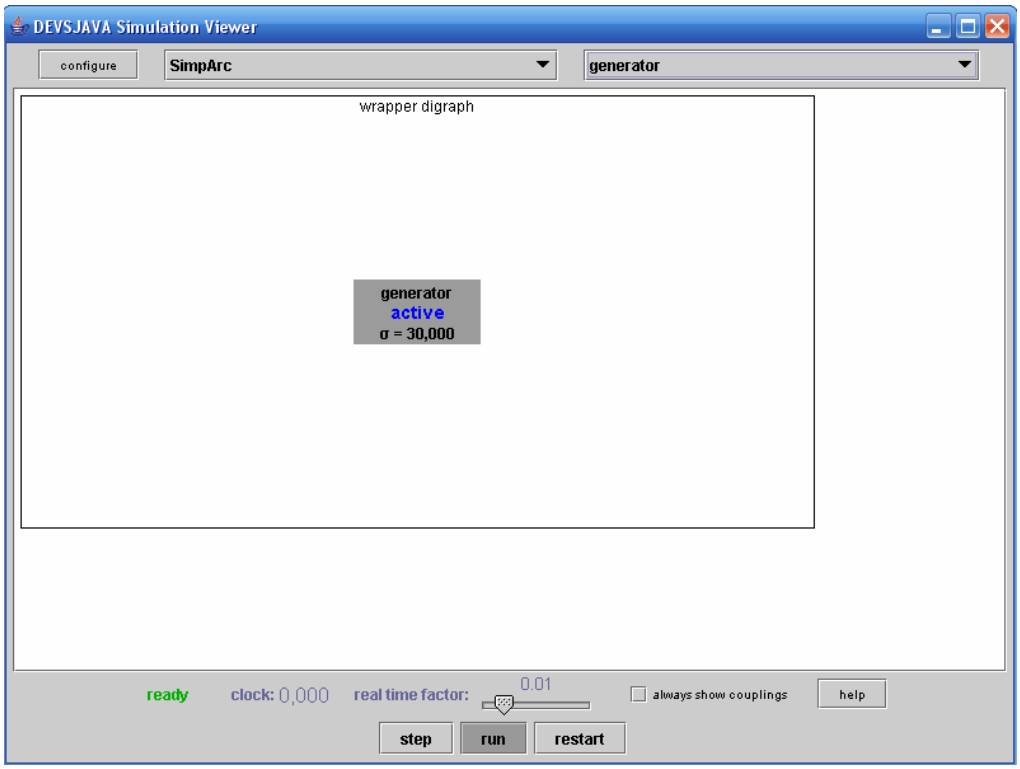

Figura 6.3: GUI de SimView con el modelo *SimpArc.generator* cargado.

A continuación, pulsando el botón "Select a package" de la GUI de *SimView*, puede escogerse entre los paquetes que se han especificado en la ventana anterior de configuración. Si se selecciona el paquete *SimpArc* y, dentro de éste, el modelo *generator*, entonces el modelo se carga en *SimView* (véase la Figura 6.3), pudiendo ser ejecutado.

6.3.4. Contador binario

En la Sección 3.2.4 se describió el modelo de un contador binario. El sistema genera un evento de valor uno por cada dos eventos de entrada de valor uno que recibe. En código en DEVSJAVA que describe ese modelo es el siguiente.

```
package SimpArc;
public class binaryCounter extends siso {
   double count;

   public binaryCounter(String name) { super(name); }

   public void initialize(){
      count  = 0;
      super.initialize();
   }

   public void  Deltext(double e, double input){
      Continue(e);
      count = count + (int)input;
      if (count >= 2){
         count = 0;
         holdIn("active",10);
      }
   }

   public void  deltint( ) { passivate(); }

   public double sisoOut() {
      if (phaseIs("active")) return 1; else return 0;
   }

   public void showState() {
      super.showState();
      System.out.println("count: " + count);
   }
}
```

6.4. CLASES Y MÉTODOS EN DEVSJAVA

En esta sección se describen algunas de las clases básicas de DEVSJAVA, que es preciso conocer para poder describir los modelos.

6.4.1. Clases contenedor

Las clases contenedor, que se encuentran en el paquete *GenCol*, se emplean para alojar instancias de objetos. En la Figura 6.4 se muestran algunas de estas clases, sus campos, algunos de sus métodos y su jerarquía. Se recomienda al alumno que inspeccione por sí mismo el contenido del paquete *GenCol*.

`entity`	Es la clase base de todas las clases de objetos que pueden alojarse en contenedores.
`Pair`	Almacena una pareja de entidades, llamadas *key* y *value*.
`Relation`	Es un conjunto de parejas *key-value*, que típicamente es usado a modo de diccionario.
`Bag`	Bolsa de entidades.
`Function`	Es una relación en la cual no puede haber dos parejas *key-value* con el mismo valor de *key*.
`Queue`	Mantiene los objetos ordenados de forma FIFO.
`intEnt`, `doubleEnt`	Entidades que contienen un número entero y real respectivamente.

6.4.2. Clases DEVS

Las clases para el modelado de sistemas están en el paquete *genDevs.Modeling*, y emplean las clases contenedor definidas en el paquete *GenCol*. En la Figura 6.5 se muestra una versión simplificada del árbol de herencia.

Las clases *atomic* y *digraph* son subclases de *devs*, que a su vez es subclase de *GenCol.entity*. Estas dos clases se emplean para el modelado de DEVS atómicos y compuestos, respectivamente.

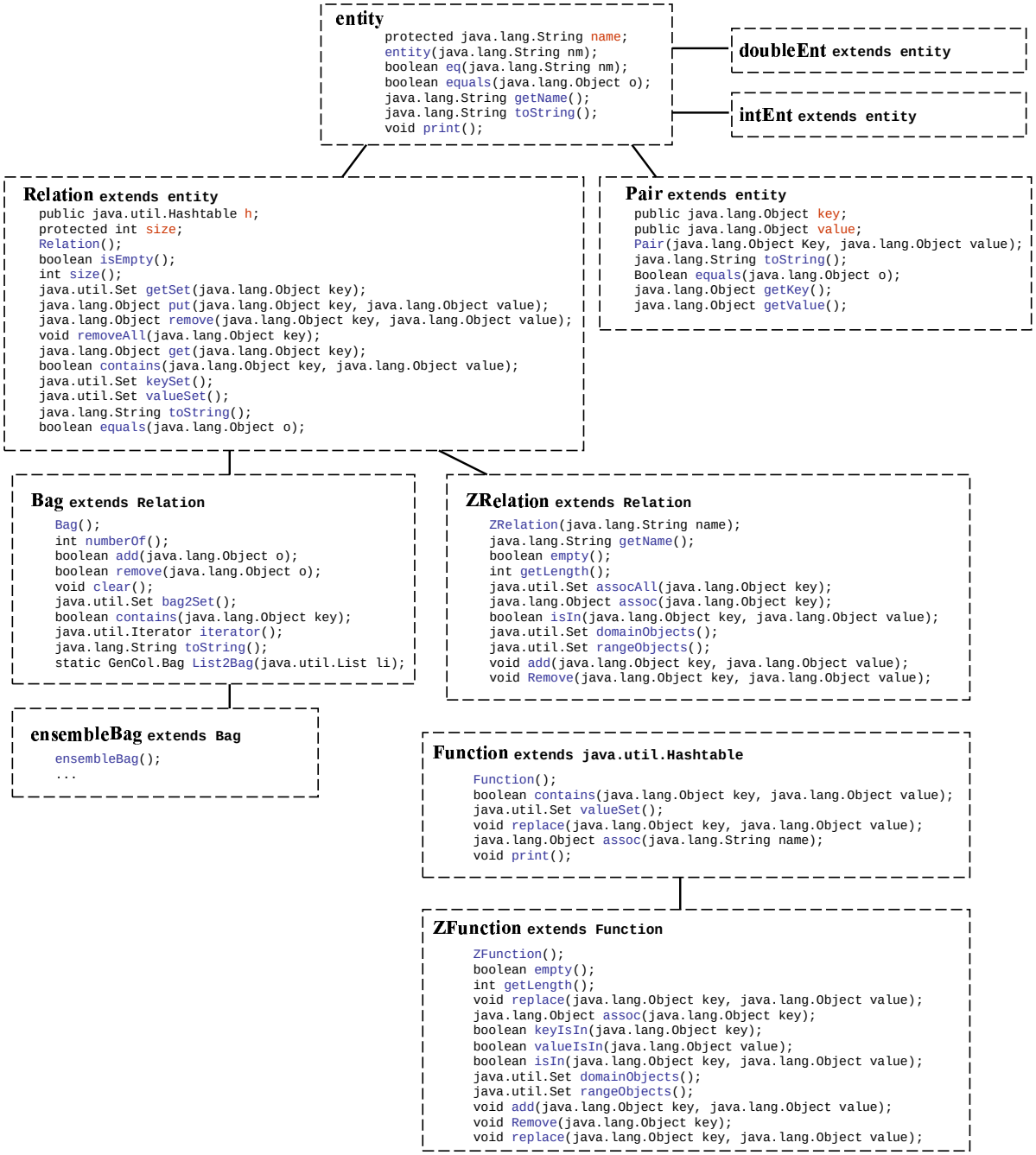

Figura 6.4: Algunas clases contenedor del paquete *GenCol*.

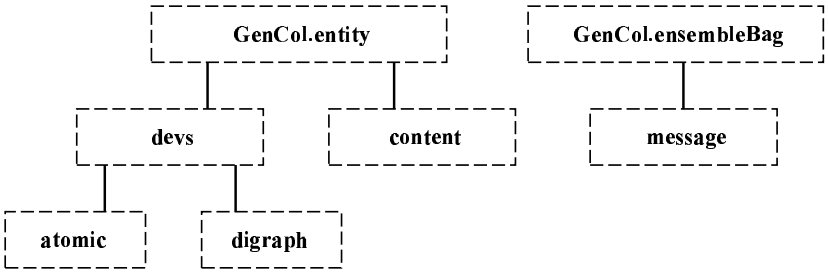

Figura 6.5: Algunas clases para la descripción de los modelos DEVS.

La clase *message* es una subclase de *GenCol.ensembleBag* y representa los mensajes que son transmitidos entre los componentes de un modelo compuesto. El mensaje aloja instancias de la clase *content*.

A continuación se describen algunas de las características de estas clases que son más relevantes a la hora de usarlas para la construcción de modelos.

Clase devs

La clase *devs* es la superclase de las dos clases para la descripción de los modelos, que son *atomic* y *digraph*. A continuación, se muestra parte del código de la clase *devs*.

```
public abstract class genDevs.modeling.devs extends    GenCol.entity
                                          implements genDevs.modeling.IODevs {

   protected double tL, tN;
   public final double INFINITY = Double.POSITIVE_INFINITE;

   public devs(String name) {
      super(name);
      ...
   }

   public void initialize() {
      tL = 0;
      tN = tL + ta();
      output = null;
   }

   public void inject(String p, entity val, double e) {
      mensaje in = new message();
      content co = makeContent(p, val);
      in.add(co);
   }

   public content makeContent(String p, entity value) {
      return new content(p, value);
   }

   public boolean messageOnPort(message x, String p, int i) {
      if (inports.is_in_name(p))
         System.out.println("Warning: model: " + name + " inport: " + p +
                            "has not been declared");
```

```
         return x.on_port(p, i);
   }
}
```

Clase message

La clase *message* es una subclase de *GenCol.ensembleBag* y almacena instancias de la clase *content*, la cual a su vez almacena el puerto, *p*, y el valor, *val* (una instancia de la clase *entity*).

```
public class genDevs.modeling.message extends    GenCol.ensembleBag
                                      implements genDevs.modeling.MessageInterface,
                                                 GenCol.EntityInterface {

   public message() {
      super();
   }

   public content read(int i) {
      // Devuelve el contenido i-ésimo almacenado en el mensaje
   }

   public boolean on_port(String portName, int i) {
      content con = read(i);
      return portName.equals(con.p);
   }

   public entity getValOnPort(String portName, int i) {
      if ( on_port(portName,i) ) return read(i).val;
      return null;
   }
}
```

Clase atomic

La clase *atomic* describe modelos DEVS atómicos. Contiene elementos que se corresponden con las diferentes partes del formalismo. Por ejemplo, tiene métodos para describir la función de transición interna, la función de transición externa, la función de salida y la función de avance en el tiempo. Estos métodos se aplican a las variables de la instancia, que caracterizan el estado del modelo.

```
public class genDevs.modeling.atomic extends    genDevs.modeling.devs
```

```
                                    implements genDevs.modeling.AtomicInterface,
                                               genDevs.modeling.variableStructureInterface {

   public atomic(String name) {
      super(name);
      phases = new set();
      lastOutput = new message();
      addInport("in");
      addOutport("out");
      phases.add("passive");
      passivate();
   }

   public void passivate() {
      phase = "passive";
      sigma = INFINITY;
   }

   public holdIn(String p, double s) {
      phase = p;
      sigma = s;
   }

   public void passivateIn(String phase) {
      holdIn(phase, INFINITY);
   }

   public boolean phaseIs(String Phase) {
      if ( !(phases.is_in_name(Phase)) )
         System.out.println("Warning: model: " + getName() + " phase: " +
                            Phase + " has not been declared" );
      return phase.equals(Phase);
   }

   public void deltint() { }

   public void deltext(double e, message x) { }

   public void deltcon(double e, message x) {
      // Transición externa seguida por transición interna
      // deltext(e,x);
      // deltint();

      // Transición interna seguida por transición externa
      deltint();
      deltext(0,x);
   }
```

```
}
```

Clase digraph

La clase *digraph* permite definir modelos DEVS compuestos. Además de los componentes, permite especificar las conexiones entre los componentes (acoplo interno), y las conexiones de los componentes con los puertos de entrada y de salida del modelo compuesto.

```
public class genDevs.modeling.digraph extends    genDevs.modeling.devs
                                      implements genDevs.modeling.Coupled {

   protected genDevs.modeling.couprel Coupling;

   public digraph(String nm) {
      super(nm);
      Coupling = new couprel();
      addInport("in");
      addOutport("out");
   }

   public void add(genDevs.modeling.IODevs iod) { ... }

   public void addCoupling(genDevs.modeling.IODevs src,  String p1,
                           genDevs.modeling.IODevs dest, String p2) {
      port port1 = new port(p1);
      port port2 = new port(p2);
      Coupling.add(src, p1, dest, p2);
   }
}
```

6.4.3. Clases DEVS SISO

DEVSJAVA permite que los modelos atómicos tengan eventos simultáneos, de valor arbitrario, en sus puertos de entrada. Esta capacidad puede ser limitada, de tal modo que sólo se permita la llegada de un evento en cada instante de tiempo (como en DEVS clásico). El único motivo de hacer esto es facilitar el aprendizaje de DEVSJAVA a aquellas personas con escasa experiencia en programación con Java.

A continuación, se describen las clases *classic* y *siso*, contenidas en el paquete *SimpArc*. La clase *siso* es una subclase de *classic*, que a su vez es una subclase de *atomic*.

Clase classic

En la clase *classic* se define una nueva función de transición externa, *Deltext*, que restringe el método *deltext* heredado de *atomic* a un único *content*. Cuando se usa la clase *classic*, debe definirse la función de transición externa definiendo el método *Deltext*.

Se emplea como superclase *ViewableAtomic* en lugar de *atomic* ya que *ViewableAtomic* permite la visualización empleando *simView*.

```
package SimpArc;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;

public class classic extends ViewableAtomic {

   public classic(String name){
      super(name);
   }

   public content get_content(message x){
      entity ent = x.read(0);
      return (content)ent;
   }

   public void  Deltext(double e, content con) {
   }  //virtual for single input at a time

   public void  deltext(double e,message x) {
      Deltext(e, get_content(x));
   }
}
```

Clase siso

La clase *siso* restringe *Deltext* de modo que espere un único evento de entrada en cada instante. La función de transición externa debe ser escrita por el usuario de la clase empleando el método `Deltext(double e, double input)`.

```
package SimpArc;

import genDevs.modeling.*;
import GenCol.*;

public class siso extends classic {

   public void AddTestPortValue(double input){
      addTestInput("in",new doubleEnt((double)input));
   }

   public siso(String name){
      super(name);
      AddTestPortValue(0);
   }

   public void  Deltext(double e, double input) {
   }  // Función de transición externa de la clase siso

   public void  Deltext(double e, content con) {
   // Función de transición externa que espera la clase classic
      doubleEnt fe = (doubleEnt)con.getValue();
      Deltext(e,fe.getv());
   }

   public double sisoOut(){
   // Un evento de salida en cada instante. Evento de valor real
      return 0;
   }

   public message  out() {
      message  m = new message();
      content con = makeContent("out",new doubleEnt(sisoOut()));
      m.add(con);
      return m;
   }
}
```

Obsérvese que en los métodos *Deltext* y *out* se emplea la clase *doubleEnt* para recibir y transmitir valores reales por los puertos. Análogamente, se usaría la clase *intEnt* para recibir o transmitir valores enteros. Ambas clases se encuentran en el paquete *GenCol*.

A continuación, se muestra un ejemplo de envío de mensajes de valor real a través de un puerto. En particular, a través del puerto *"out1"*.

```
public message out() {
   double store = 5.5;
   message m = new message();
   content con = makeContent("out1", new doubleEnt(store));
   m.add(con);
   return m;

}
```

Análogamente, el siguiente es un ejemplo de recepción de eventos de valor real en el puerto *"in1"*.

```
public void deltext(double e, message x) {
   for (int i=0; i < x.getLength(); i++)
      if (messageOnPort(x,"in1",i)) {
         entity val = x.getValOnPort("in1",i);
         doubleEnt d = (doubleEnt)val;
         double store = d.getv();
      }
}
```

6.5. MODELOS DEVS ATÓMICOS

En esta sección se describe la programación de varios modelos DEVS empleando DEVSJAVA.

6.5.1. Modelo de un recurso

En la Sección 3.2.5 se describió el modelo de un proceso con un recurso y sin cola. La clase *proc* está contenida en el paquete *SimpArc*. A continuación, se muestra parte del código de la clase.

Obsérvese que hay dos variables de estado especiales que son heredadas de *atomic*: *phase* y *sigma*. La variable *phase* permite controlar la fase en que se encuentra el modelo. En este caso, puede estar en dos fases: *"busy"* (activo) y *"passive"* (pasivo).

```
package SimpArc;

import genDevs.modeling.*;
import GenCol.*;
import simView.*;

public class proc extends ViewableAtomic {

   protected entity job;
   protected double processing_time;

   public proc(String name, double Processing_time) {
      super(name);
      addInport("in");
      addOutport("out");
      phases.add("busy");
      processing_time = Processing_time;
   }

   public void initialize(){
      phase = "passive";
      sigma = INFINITY;
      job = new entity("job");
      super.initialize();
   }

   public void  deltext(double e, message x) {
      Continue(e);
      if (phaseIs("passive"))
         for (int i=0; i< x.getLength();i++)
            if (messageOnPort(x,"in",i)) {
               job = x.getValOnPort("in",i);
               holdIn("busy",processing_time);
            }
   }

   public void  deltint() {
      passivate();
      job = new entity("none");
   }

   public void deltcon(double e, message x) {
```

```
      deltint();
      deltext(0,x);
   }

   public message out() {
      message m = new message();
      if (phaseIs("busy")) m.add(makeContent("out",job));
      return m;
   }
}
```

Examinemos los campos y los métodos constructor y de inicialización de la clase. Las declaraciones

```
   protected entity job;
   protected double processing_time;
```

definen la instancia que almacena la entidad que está siendo procesada y el parámetro tiempo de proceso, respectivamente. El constructor

```
   public proc(String name, double Processing_time) {
      super(name);
      addInport("in");
      addOutport("out");
      phases.add("busy");
      processing_time = Processing_time;
   }
```

declara los puertos: el puerto de entrada *"in"* y el puerto de salida *"out"*. También, asigna valor al nombre de la instancia de la clase *proc* y al parámetro *processing_time*. Finalmente, indica que aparte de la *phase* "passive", que es definida en la clase *atomic*, la clase *proc* tiene una segunda fase: *"busy"*.

El método de inicialización asigna valor inicial a todas las variables de estado. Siempre es necesario inicializar las variables *sigma* y *phase*, que son heredadas. La referencia *job* se asigna a un objeto de la clase *entity*, al que se asigna el nombre *"job"*.

```
   public void initialize(){
      phase = "passive";
      sigma = INFINITY;
      job = new entity("job");
      super.initialize();
   }
```

Con `super.initialize()` se ejecuta el método *initialize* de la superclase, en el cual se asigna valor a tL y tN:

```
tL = 0;
tN = tL + ta();
```

6.5.2. Modelo de un proceso con una cola

En la Sección 4.2.1 se explicó el modelo DEVS paralelo de un proceso con una cola FIFO. La clase *procQ*, que está contenida en el paquete *SimpArc*, es la descripción de este modelo en DEVSJAVA. A continuación, se muestra parte del código de esta clase.

```
package SimpArc;

import genDevs.modeling.*;
import GenCol.*;

public class procQ extends proc {

   protected Queue q;

   public procQ(String name, double Processing_time) {
      super(name, Processing_time);
      q = new Queue();
   }

   public void initialize() {
      q = new Queue();
      super.initialize();
   }

   public void deltext(double e, message x) {
      Continue(e);
      if (phaseIs("passive")){
         for (int i=0; i < x.size(); i++)
            if (messageOnPort(x, "in", i)) q.add(x.getValOnPort("in", i));
         holdIn("busy", processing_time);
         job = (entity)q.first();  // Se procesa la primera entidad de la cola
      } else if (phaseIs("busy")) {
         for (int i=0; i< x.size();i++)
            if (messageOnPort(x, "in", i)) {
               entity jb = x.getValOnPort("in", i);
```

```
            q.add(jb);
         }
      }
   }

   public void  deltint() {
      q.remove();
      if(!q.isEmpty()){
         job = (entity)q.first();
         holdIn("busy", processing_time);
      } else passivate();
   }
}
```

El modelo anterior puede hacerse más realista si se considera que el tiempo de proceso en lugar de ser constante obedece una determinada distribución de probabilidad. Esto puede modelarse de manera sencilla modificando la función de transición externa. Por ejemplo, de la forma siguiente:

```
holdIn("busy", randExpon(100,2));
```

El paquete *statistics* de DEVSJAVA contiene algunas clases para la generación de observaciones aleatorias.

6.5.3. Modelo de un conmutador

En la Sección 3.3.1 se describió el modelo de un conmutador. A continuación se muestra parte del código de la correspondiente clase, llamada *Switch*, que se encuentra en el paquete *SimpArc*.

```
package SimpArc;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;

public class Switch extends ViewableAtomic {

   protected entity  job;
   protected double  processing_time;
   protected boolean sw;
   protected String  input;
```

```
public Switch(String name, double Processing_time) {
   super(name);
   addInport("in");
   addOutport("out");
   addInport("in1");
   addOutport("out1");
   processing_time = Processing_time;
   phases.add("busy");
}

public void initialize() {
   phase = "passive";
   sigma = INFINITY;
   job = new entity("job");
   sw = false;
   input = new String("in");
   super.initialize();
}

public void deltext(double e, message x) {
   Continue(e);
   if (phaseIs("passive")) {
      for (int i=0; i< x.getLength();i++)
         if (messageOnPort(x,"in",i)) {
            job = x.getValOnPort("in",i);
            input = "in";
            holdIn("busy",processing_time);
         }
      for (int i=0; i< x.getLength();i++)
         if (messageOnPort(x,"in1",i)) {
            job = x.getValOnPort("in1",i);
            input = "in1";
            holdIn("busy",processing_time);
         }
   }
   sw = !sw;
}

public void deltint() {
   passivate();
}

public message out() {
   message   m = new message();
   if (phaseIs("busy")) {
      content   con;
```

```
         if (!sw && input.equals("in"))       con = makeContent("out", job);
         else if (!sw && input.equals("in1")) con = makeContent("out1",job);
         else if (sw && input.equals("in"))   con = makeContent("out1",job);
         else                                 con = makeContent("out", job);
         m.add(con);
      }
      return m;
   }
}
```

6.5.4. Modelo de un generador

A continuación se muestra el modelo DEVSJAVA de un generador de eventos con dos puertos de entrada, *start* y *stop*, que controlan el comienzo y la finalización de la generación de eventos. Mientras el generador se encuentra en la fase *"active"*, genera eventos de salida cada *interArrivalTime* unidades de tiempo. La clase *genr* se encuentra en el paquete *SimpArc*.

```
package SimpArc;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;

public class genr extends ViewableAtomic {

   protected double interArrivalTime;
   protected int    count;

   public genr(String name, double interArrivalTime) {
      super(name);
      addInport("in");
      addOutport("out");
      addInport("stop");
      addInport("start");
      this.interArrivalTime = interArrivalTime ;
   }

   public void initialize() {
      holdIn("active", interArrivalTime);
      count = 0;
      super.initialize();
   }
```

```
   public void deltext(double e, message x) {
      Continue(e);
      if (phaseIs("passive") && somethingOnPort(x,"start"))
         holdIn("active",interArrivalTime);
      if (phaseIs("active")  && somethingOnPort(x,"stop"))
         phase = "finishing";
   }

   public void deltint() {
      count = count + 1;
      if (phaseIs("active"))
         holdIn("active",interArrivalTime);
      else
         passivate();
   }

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

Los métodos *somethingOnPort* y *outputNameOnPort* están definidos en una de las superclases de *ViewableAtomic*, llamada *friendlyAtomic*, que está en el paquete *genDevs.modeling*. Las declaraciones de los métodos son las siguientes:

```
   public boolean somethingOnPort(message x, String port);

   public message outputNameOnPort(String nm, String port);
```

6.5.5. Modelo de un generador aleatorio

Puede modificarse el modelo descrito en la Sección 6.5.4 de modo que el intervalo de tiempo entre sucesivas generaciones de eventos esté distribuido aleatoriamente. La clase *genrRand* está en el paquete *SimpArc*.

```
package SimpArc;

import simView.*;
import genDevs.modeling.*;
import GenCol.*;
import statistics.*;

public class genrRand extends ViewableAtomic {
```

```
protected double interArrivalTime;
protected int    count;
protected rand   r;
protected long   seed;

public genrRand(String name, double InterArrivalTime, long seed) {
   super(name);
   this.seed = seed;
   addInport("stop");
   addInport("start");
   addOutport("out");
   interArrivalTime = InterArrivalTime ;
   r = new rand(seed);
}

public void initialize(){
   r = new rand(seed);
   holdIn("active",r.expon(interArrivalTime));
   count = 0;
   super.initialize();
}

public void deltext(double e, message x) {
   Continue(e);
   if (phaseIs("passive") && somethingOnPort(x,"start")) {
      holdIn("active",r.uniform(interArrivalTime));
      // holdIn("active",r.expon(interArrivalTime));
   }
   if (phaseIs("active") && somethingOnPort(x,"stop"))
      phase = "finishing";
}

public void deltint() {
   if(phaseIs("active")) {
      count = count + 1;
      holdIn("active",r.expon(interArrivalTime));
      // holdIn("active",r.uniform(interArrivalTime));
      // holdIn("active",r.normal(interArrivalTime,1.0));
   } else
      passivate();
}

public message out() {
   message  m = new message();
   content con = makeContent("out",new job("job"+name+count,r.expon(1000)));
   m.add(con);
```

```
        return m;
    }
}
```

6.5.6. Modelo de un transductor

El transductor está diseñado para medir las dos magnitudes siguientes, indicativas del funcionamiento de un proceso:

- *Throughput*. Es el número medio de entidades que son procesadas por unidad de tiempo. Se calcula como el número de entidades procesadas durante el tiempo que dura la simulación, dividido por el tiempo que ha durado la simulación.
- *Tiempo de ciclo* (*turnaround time*). Es el promedio del tiempo que transcurre entre que una entidad llega al proceso y lo abandona. En un proceso sin cola y con un único recurso, el ciclo de vida es igual al tiempo medio de proceso del recurso.

El transductor realiza los cálculos necesarios para estimar estas dos medidas del comportamiento. Para ello, mantiene una lista, *arrived*, en la que almacena el *job-id* (etiqueta identificativa de la entidad) y el instante de llegada de cada una de la entidades que ha llegado al puerto de entrada *"ariv"* del transductor. Cuando la entidad llega al transductor a través de su puerto de entrada *"solved"*, entonces se añade la entidad a la lista *solved* y se calcula su ciclo de vida, es decir, el tiempo transcurrido entre su llegada y su marcha. Las listas *arrived* y *solved* son instancias de la clase *Function*.

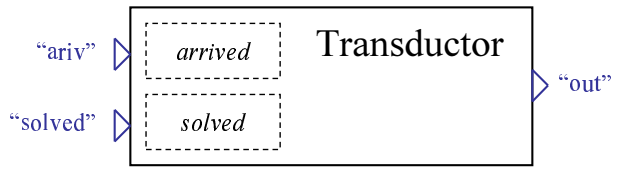

Figura 6.6: Modelo del transductor.

El transductor mantiene su propio reloj (variable *clock*) para poder obtener el instante de llegada y de marcha de las entidades. El formalismo DEVS no proporciona a los componentes acceso al reloj de la simulación. Por ello, si los modelos

necesitan conocer el valor del tiempo, deben gestionar su propio reloj. Esto puede hacerse de manera sencilla a partir de la información del tiempo transcurrido, que está disponible en las variables σ y e.

Obsérvese que el comportamiento del transductor está dictado esencialmente por su función de transición externa. En el modelo del transductor únicamente se usa una transición interna para producir una salida al final del intervalo de observación durante el cual se están calculando el *throughput* y el *tiempo de ciclo*.

A continuación, se muestra parte del código de la clase *transd*, que está situada en el paquete *SimpArc*. Obsérvese que cualquier modelo atómico puede escribir en la consola o en un fichero, por ejemplo, con la finalidad de registrar los instantes en que se producen los eventos. Véase el método *show_state* de la clase *transd*.

```
package SimpArc;
import simView.*;
import genDevs.modeling.*;
import GenCol.*;

public class transd extends ViewableAtomic {

   protected Function arrived, solved;
   protected double   clock, total_ta, observation_time;

   public transd(String name, double Observation_time) {
      super(name);
      addOutport("out");
      addInport("ariv");
      addInport("solved");
      arrived          = new Function();
      solved           = new Function();
      observation_time = Observation_time;
   }

   public void initialize() {
      phase    = "active";
      sigma    = observation_time;
      clock    = 0;
      total_ta = 0;
      arrived  = new Function();
      solved   = new Function();
   }

   public void  deltext(double e, message x) {
      clock = clock + e;
      Continue(e);
```

```
   entity val;
   for (int i=0; i < x.size(); i++) {
      if (messageOnPort(x,"ariv",i)) {
         val = x.getValOnPort("ariv",i);
         arrived.put(val.getName(), new doubleEnt(clock));
      }
      if (messageOnPort(x,"solved",i)) {
         val = x.getValOnPort("solved",i);
         if (arrived.containsKey(val.getName())) {
            entity     ent              = (entity)arrived.assoc(val.getName());
            doubleEnt  num              = (doubleEnt)ent;
            double     arrival_time     = num.getv();
            double     turn_around_time = clock - arrival_time;
            total_ta = total_ta + turn_around_time;
            solved.put(val, new doubleEnt(clock));
         }
      }
   } // final del bucle for
   show_state();
}

public void deltint() {
   clock = clock + sigma;
   passivate();
   show_state();
}

public message out() {
   message  m   = new message();
   content  con = makeContent("out", new entity("TA: "+compute_TA()));
   m.add(con);
   return m;
}

public double compute_TA() {
   double avg_ta_time = 0;
   if (!solved.isEmpty())
      avg_ta_time = ((double)total_ta)/solved.size();
   return avg_ta_time;
}

public double compute_Thru() {
   double thruput = 0;
   if(clock > 0)
      thruput = solved.size()/(double)clock;
   return thruput;
}
```

```
   public void show_state() {
      System.out.println("state of  " + name + ": ");
      System.out.println("phase, sigma : " + phase  + " " + sigma + " ");
      if (arrived != null && solved != null) {
         System.out.println(" jobs arrived :");
         // arrived.print_all();
         System.out.println("total :"  +  arrived.size());
         System.out.println("jobs solved :");
         // solved.print_all();
         System.out.println("total :" + solved.size());
         System.out.println("AVG TA = " + compute_TA());
         System.out.println("THRUPUT = " + compute_Thru());
      }
   }
}
```

6.6. MODELOS DEVS COMPUESTOS Y JERÁRQUICOS

En esta sección se muestran varios ejemplos que ilustran la descripción de modelos compuestos usando DEVSJAVA.

6.6.1. Modelo de una secuencia de etapas

En la Sección 3.4.1 se describió el modelo compuesto por tres etapas en serie, que es mostrado en la Figura 6.7.

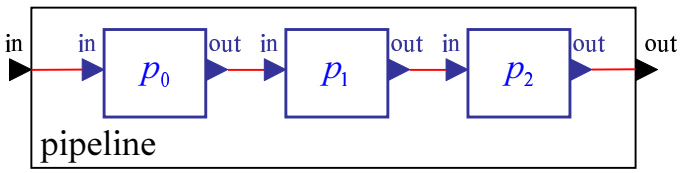

Figura 6.7: Pipeline compuesta por tres etapas.

A continuación, se muestra parte de la descripción de este modelo compuesto usando DEVSJAVA. La clase *pipeSimple* se encuentra en el paquete *SimpArc*.

```
package SimpArc;
import java.awt.*;
import simView.*;
```

```
import GenCol.*;

public class pipeSimple extends ViewableDigraph {

   public pipeSimple(String name, double proc_time) {
      super(name);
      make(proc_time);
      addInport("in");
      addOutport("out");
   }

   private void make (double proc_time) {
      ViewableAtomic  p0 = new proc("proc0", proc_time/3);
      ViewableAtomic  p1 = new proc("proc1", proc_time/3);
      ViewableAtomic  p2 = new proc("proc2", proc_time/3);
      add(p0);
      add(p1);
      add(p2);
      addCoupling(this, "in",  p0,   "in");
      addCoupling(p0,   "out", p1,   "in");
      addCoupling(p1,   "out", p2,   "in");
      addCoupling(p2,   "out", this, "out");
      initialize();
   }
}
```

6.6.2. Marco experimental

En la Sección 1.8.3 se introdujo el concepto de marco experimental. Se explicó que el marco experimental puede interpretarse como un sistema que interactúa con el sistema de interés para obtener observaciones, bajo determinadas condiciones experimentales, estando típicamente compuesto por los tres elementos mostrados en la Figura 1.7):

Generador	Genera las secuencias de entrada al sistema.
Receptor	Monitoriza el experimento, para comprobar que se satisfacen las condiciones experimentales requeridas.

Transductor Observa y analiza las secuencias de salida del sistema, calculando medidas del comportamiento del mismo, tales como el throughput y el tiempo de ciclo, que eran de interés en el modelo de la Sección 6.5.6, o la utilización de los recursos, la ocurrencia de ciertos eventos como fallos o bloqueos, el valor máximo o mínimo de ciertas magnitudes, etc.

Pueden conectarse entre sí instancias de las clases *genr* y *transd*, descritas en las Secciones 6.5.4 y 6.5.6, con el fin de modelar un marco experimental (*experimental frame*), tal como se muestra en la Figura 6.8.

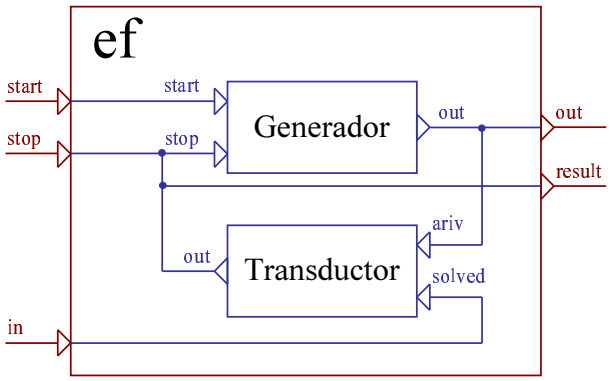

Figura 6.8: Marco experimental y sus componentes.

El marco experimental tiene los tres puertos de entrada siguientes:

`in` Recibe las entidades ya procesadas, que son enviadas al puerto *solved* del transductor.

`start`, `stop` Controlan el comienzo y finalización de la generación de entidades en el generador.

y los dos puertos de salida siguientes:

`out` Transmite los identificadores de las entidades creadas por el generador.

`result` Transmite las medidas del comportamiento calculadas por el transductor.

El marco experimental contiene dos conexiones internas:

– El puerto de salida del generador envía los identificadores de las entidades al puerto *ariv* del transductor.

– El puerto *out* del transductor envía las medidas calculadas del comportamiento al puerto *stop* del generador.

Obsérvese que cuando el generador genera una entidad, en el mismo instante de tiempo ésta llega al puerto *ariv* del transductor y a los puertos de entrada de los componentes externos conectados al puerto *out* del marco experimental. Igualmente, pueden conectarse varios puertos de salida a un mismo puerto de entrada. Esto no es un problema en DEVS paralelo, ya que las bolsas pueden alojar varios eventos que llegan simultáneamente a un puerto de entrada.

A continuación, se muestra parte del código de la clase *ef*, que está situada en el paquete *SimpArc*.

```
package SimpArc;
import simView.*;
import java.awt.*;
import GenCol.*;

public class ef extends ViewableDigraph {

   public ef(String nm, double int_arr_t, double observe_t) {
      super(nm);
      efConstruct(int_arr_t, observe_t);
   }

   public void efConstruct(double int_arr_t, double observe_t) {
      addInport("in");
      addOutport("out");
      addOutport("result");
      ViewableAtomic g = new genr("g", int_arr_t);
      ViewableAtomic t = new transd("t", observe_t);
      add(g);
      add(t);
      addTestInput("start", new entity());
      addTestInput("stop",  new entity());
      addTestInput("in",    new entity("jobg0"));
      addTestInput("in",    new entity("job0"));
      addTestInput("in",    new entity("job1"));
      initialize();
      addCoupling( g,    "out",   t,    "ariv"   );
      addCoupling( this, "in",    t,    "solved" );
      addCoupling( t,    "out",   g,    "stop"   );
```

```
      addCoupling( this, "start", g,    "start"  );
      addCoupling( this, "stop",  g,    "stop"   );
      addCoupling( g,    "out",   this, "out"    );
      addCoupling( t,    "out",   this, "result" );
   }
}
```

6.6.3. Modelo de una red de conmutación

En la Sección 3.4.2 se describió el modelo de la red de conmutación mostrada en la Figura 6.9. La clase *netSwitch*, que se encuentra en el paquete *SimpArc*, representa dicha red de conmutación y su marco experimental. Puede emplearse *simView* para visualizar el diagrama de la clase (véase la Figura 6.10) y para ejecutar la simulación.

```
package SimpArc;
import java.awt.*;
import simView.*;
import GenCol.*;
public class netSwitch extends ViewableDigraph {

   public netSwitch() {
      super("netSwitch");
      addInport("in");
      addOutport("out");
      ViewableDigraph ef = new ef("ef", 20, 500);
      ViewableAtomic s0  = new Switch("switch0", 2.5);
      ViewableAtomic p0  = new proc("proc0", 5);
      ViewableAtomic p1  = new proc("proc1", 10);
      add(ef);
      add(s0);
      add(p0);
      add(p1);
      initialize();
      showState();
      addCoupling(this, "in",   s0,   "in");
      addCoupling(ef,   "out",  s0,   "in");
      addCoupling(p0,   "out",  this, "out");
      addCoupling(p1,   "out",  this, "out");
      addCoupling(s0,   "out",  p0,   "in");
      addCoupling(s0,   "out1", p1,   "in");
      addCoupling(p0,   "out",  ef,   "in");
      addCoupling(p1,   "out",  ef,   "in");
   }
}
```

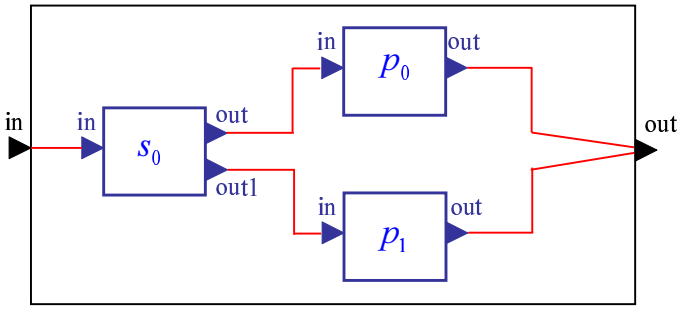

Figura 6.9: Red de conmutación.

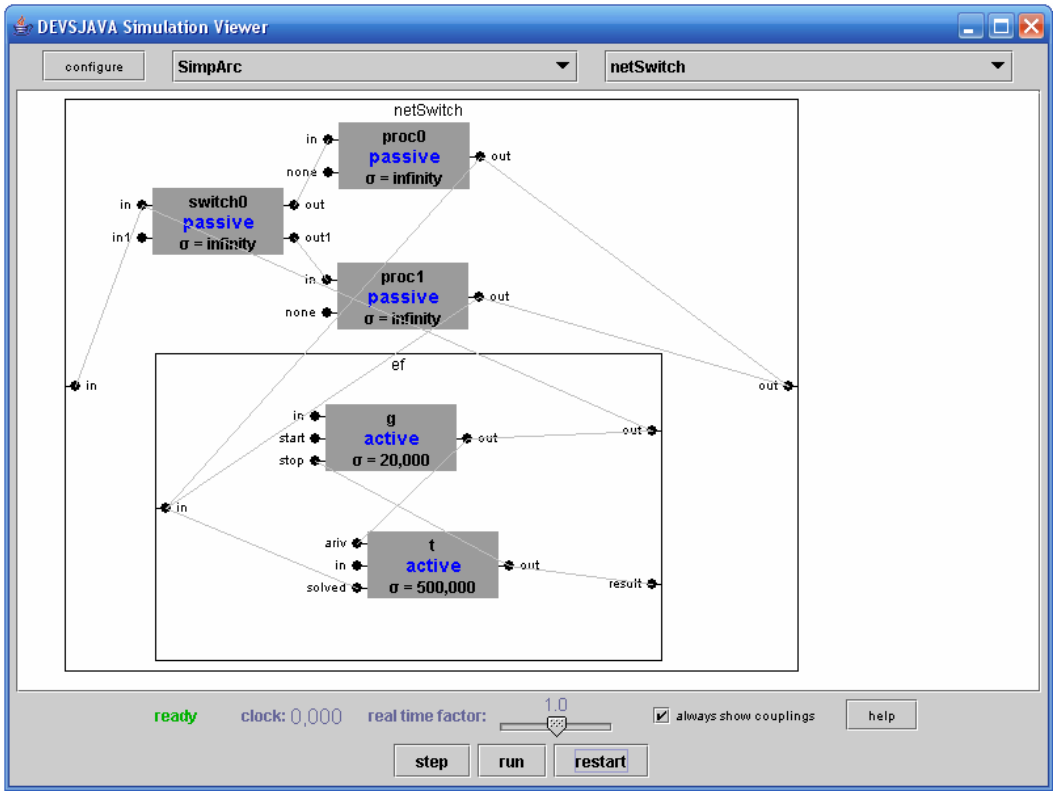

Figura 6.10: Diagrama de la clase *netSwitch*.

6.7. SIMVIEW

SimView (*DEVSJAVA Simulation Viewer*) es una aplicación, escrita por J. Mather (Mather 2003), que extiende DEVSJAVA con objeto de permitir visualizar la estructura y el comportamiento de los modelos DEVS. Se distribuye en el paquete *simView* del fichero *coreDEVS.jar*. La clase *SimView* contiene el método *main* que debe ejecutarse para arrancar la aplicación.

A continuación se describen algunas de las capacidades para la visualización y simulación del modelo que ofrece *SimView*.

Representación modular y jerárquica

Muestra los componentes del modelo, de manera jerárquica, sus puertos y la conexión entre los puertos. Obsérvese la Figura 6.10, en la cual está representada la estructura modular de cada uno de los niveles jerárquicos del modelo. El recuadro externo representa la clase *netSwitch*, dentro del cual se sitúan tres componentes atómicos (*switch0*, *proc0* y *proc1*), que son representados mediante rectángulos opacos, y un modelo compuesto (*ef*), que es representado mediante un recuadro, dentro del cual se muestran sus dos componentes atómicos (g y t), representados mediante rectángulos opacos.

Los componentes compuestos pueden representarse o bien mostrando su estructura interna (*blackbox=false*) o bien como cajas negras (*blackbox=true*). El método

```
void setBlackBox( boolean blackBox )
```

de la clase *simView.ViewableDigraph* permite especificar si debe o no mostrarse la estructura interna del componente. Por ejemplo, en el modelo de la red de conmutación puede hacerse que el marco experimental sea representado como una caja negra mediante: `ef.setBlackBox(true)`.

SimView permite cambiar la posición de un componente pinchando sobre él y arrastrando.

Control de la simulación

Permite, mediante los botones *step* y *run*, ejecutar la simulación paso a paso o en modo continuo.

Mediante el deslizador *real time factor*, permite ejecutar la simulación fijando la relación entre el reloj de la simulación y el tiempo físico. En particular, permite realizar simulaciones en tiempo real. En este caso, el reloj de la simulación es sincronizado con el tiempo físico, de modo que los eventos son planificados en base al tiempo físico.

Animación

Proporciona una animación visual de los mensajes que se transmiten a través de las conexiones entre los componentes, en cada paso de la simulación. En la Figura 6.11 se muestra el mensaje *jobg0*, generado en el puerto *out* del generador *g*, viajando a través de las conexiones a los componentes *t* y *switch0*.

Información en tiempo de simulación

Proporciona información durante la simulación del estado actual de cada componente. Para cada componente atómico se muestra el valor actual de *phase* y *sigma*, y situando el cursor sobre el componente se muestra información adicional. Por defecto, el valor de *tL* y *tN*. En el código Java del componente puede especificarse qué otra información debe mostrarse. Para ello, debe usarse el método *getTooltipText*, de la clase *simView.ViewableAtomic*.

Por ejemplo, en la Figura 6.12 se muestra el cuadro informativo (*tool-tip window*) que aparece al situar el cursor sobre el componente *proc0*. Obsérvese que se muestra el String devuelto por el método *getTooltipText* de la clase *proc*:

```
public String getTooltipText() {
   return super.getTooltipText() + "\n" + "job: " + job.getName();
}
```

Experimentación con el modelo

Durante la simulación, permite inyectar eventos externos en los puertos de entrada. Haciendo clic sobre cualquiera de los puertos de entrada de un componente, aparece una lista con los eventos que pueden ser inyectados. En dicha lista, para cada evento se especifica: el puerto de entrada, el valor y el tiempo *e* que debe esperarse, empezando a contar desde el instante actual, para inyectar el evento.

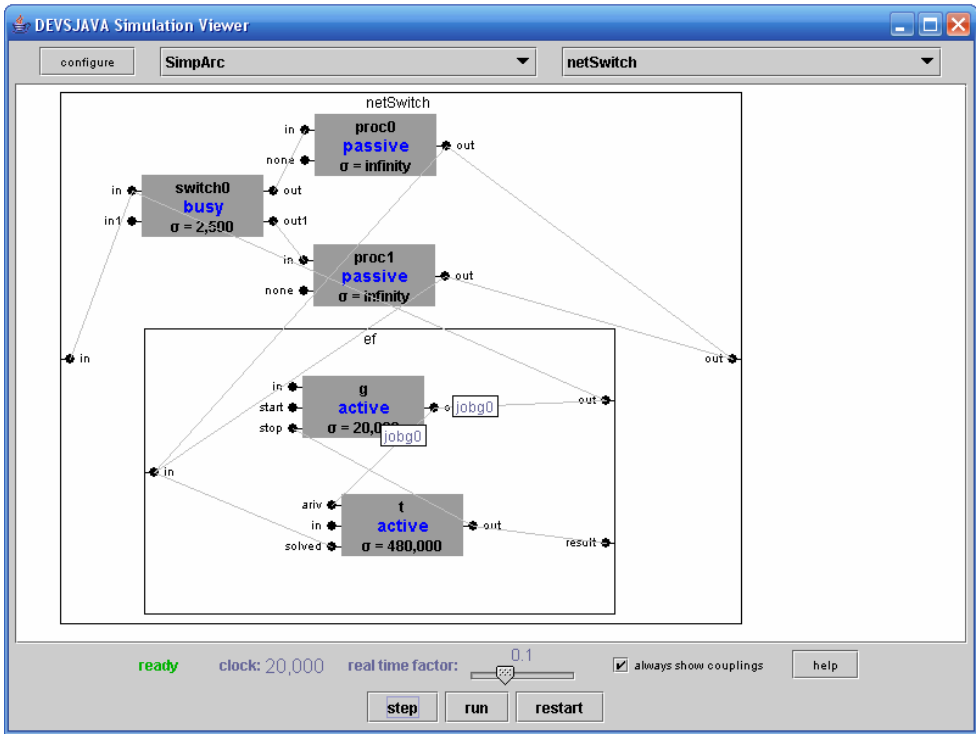

Figura 6.11: Visualización de los mensajes transmitidos por las conexiones.

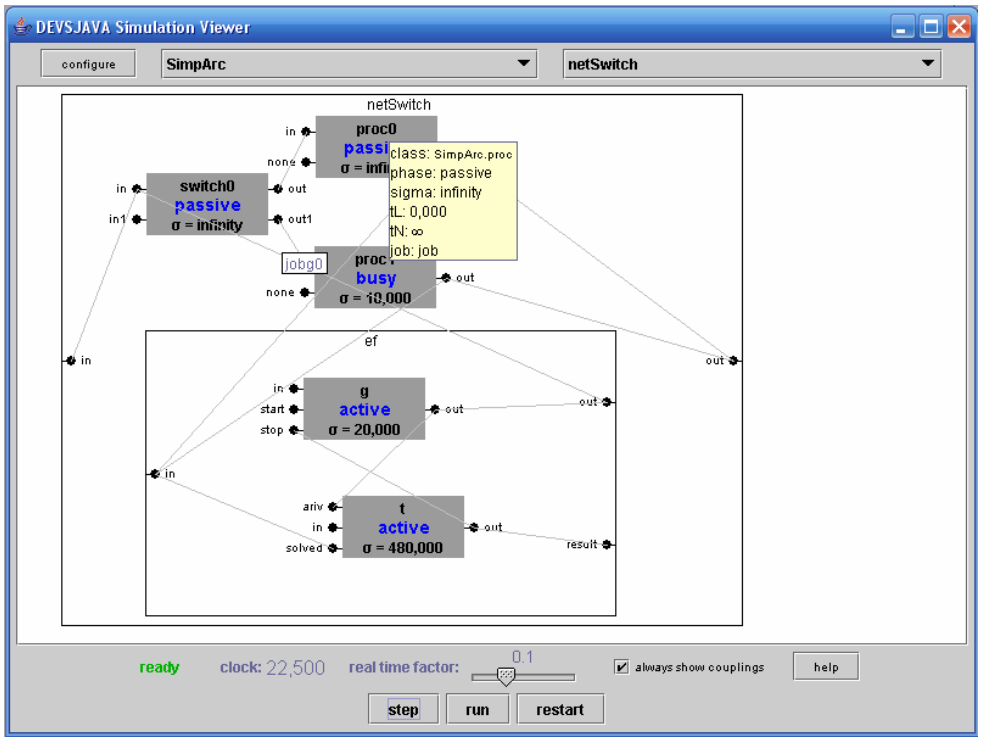

Figura 6.12: Visualización del estado de los componentes.

SimView muestra un mensaje de error si el usuario trata de inyectar un evento con un valor de *e* mayor que el tiempo que queda hasta el siguiente evento interno del componente.

Por ejemplo, haciendo clic sobre cualquiera de los dos puertos de entrada del componente *proc0* se abre la ventana mostrada en la Figura 6.13, en la que se muestran los valores *puerto-valor-e*. Para inyectar el evento *in-job2-5.0*, se selecciona éste en la lista y se pulsa el botón *inject*. Dentro de 5 unidades de tiempo, se producirá un evento de valor *"job2"* en el puerto *in* del componente *proc0*.

Los eventos que aparecen en la lista deben definirse en el código del componente, empleando para ello el método *addTestInput*. Por ejemplo, se ha realizado la definición siguiente en la clase *proc*.

```
public proc(String  name,double  Processing_time){
   super(name);
   addInport("in");
   addOutport("out");
   addInport("none");  // allows testing for null input
                       // which should cause only "continue"
   processing_time = Processing_time;
   addTestInput("in",   new entity("job1")    );
   addTestInput("in",   new entity("job2"),  5);
   addTestInput("none", new entity("job"),  10);
   addTestInput("in",   new entity("job"),  20);
}
```

Visualización de los resultados

Pueden visualizarse los resultados de la simulación, por ejemplo, representando gráficamente la evolución de algunas variables del modelo, empleando la clase *CellGridPlot*, que se encuentra en el paquete *genDevs.plots*. En la Figura 6.14 se muestra un ejemplo de uso de esta clase.

Las instancias de la clase *CellGridPlot* se emplean de la misma forma que cualquier otro componente DEVS. Tal como se muestra en la Figura 6.15, la clase tiene una serie de puertos, cada uno de los cuales espera recibir un determinado tipo de entidad y realiza un determinado tipo de representación gráfica de dichas entidades.

Los puertos de entrada *timePlot* y *pulsePlot* aceptan entidades del tipo *doubleEnt*.

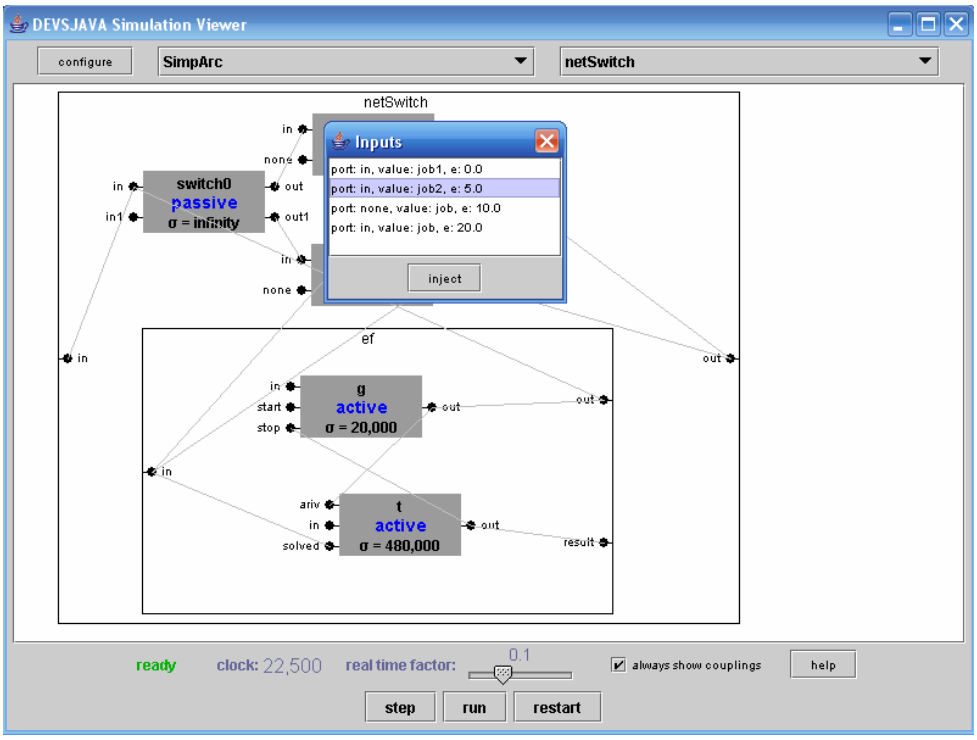

Figura 6.13: Lista de eventos que pueden ser inyectados en los puertos de entrada del componente *proc0*.

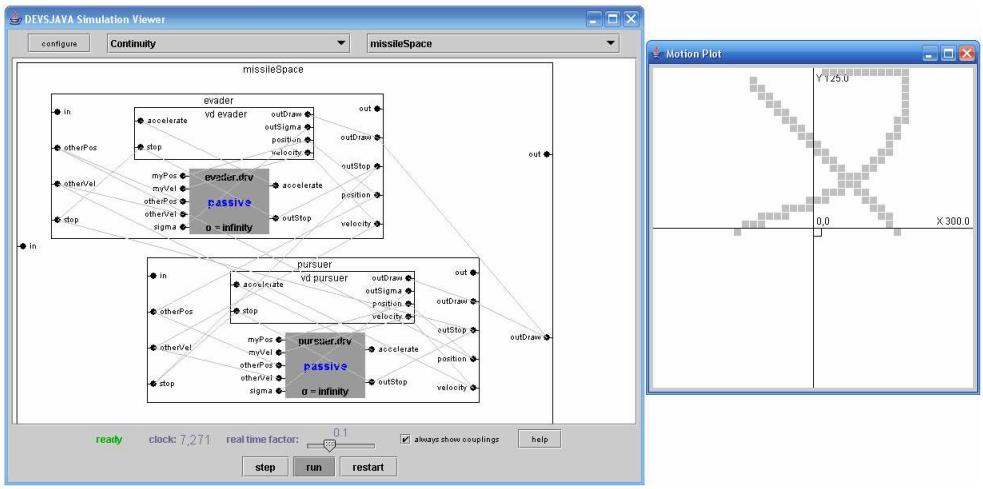

Figura 6.14: En la clase *Continuity.missileSpace* puede encontrarse un ejemplo de uso de *genDevs.plots.CellGridPlot*.

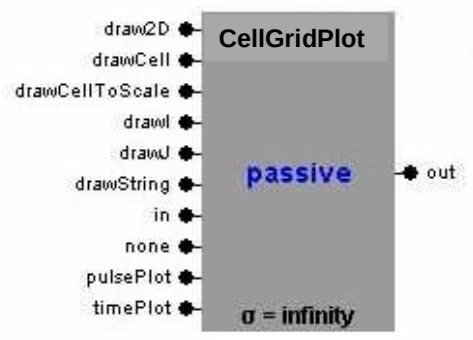

Figura 6.15: Puertos de la clase CellGridPlot.

`timePlot`	Muestra trayectorias frente al tiempo.
`pulsePlot`	Dibuja líneas verticales desde el eje x cuya altura representa los datos representados.

Por defecto, las instancias de la clase *CellGridPlot* están ocultas. Si se desea que una instancia sea visible, debe invocarse su método `setHidden(false)`.

La representación gráfica se realiza durante la simulación. Puede controlarse el aspecto gráfico de los valores pasados. Por ejemplo, pueden ir borrándose (dibujándolos de color blanco) o difuminándose (usando colores suaves). El argumento *delay* del constructor determina el tamaño del intervalo de tiempo que no es difuminado.

En la clase *boxCar*, que está contenida en el paquete *pulseModels*, puede encontrarse un ejemplo de uso de la clase *CellGridPlot*:

6.8. EJERCICIOS DE AUTOCOMPROBACIÓN

Los ejercicios propuestos a continuación están basados en el contenido del artículo

DEVS Component-Based M&S Framework: an Introduction

Bernard P. Zeigler, Hessan S. Sarjoughian

que puede encontrar en el fichero *lectura6a.pdf*. Por favor, lea en primer lugar este artículo. A continuación, resuelva los ejercicios.

Ejercicio 6.1

Suponga que el comportamiento de una neurona que puede emitir un único impulso es el siguiente. Inicialmente la neurona se encuentra en el estado *receptivo*. Cuando llega un evento externo, la neurona pasa al estado *disparo*, en el cual permanece durante un cierto tiempo, que es un parámetro del modelo denominado *tiempo de disparo*. Trascurrido ese tiempo, la neurona emite un impulso y pasa al estado *refractario*, en el cual permanece indefinidamente, y en el cual ignora cualquier evento de entrada. En la Figura 6.16 se muestra esquemáticamente el diagrama de estados de la neurona. Escriba la descripción del modelo de la neurona empleando el formalismo DEVS clásico.

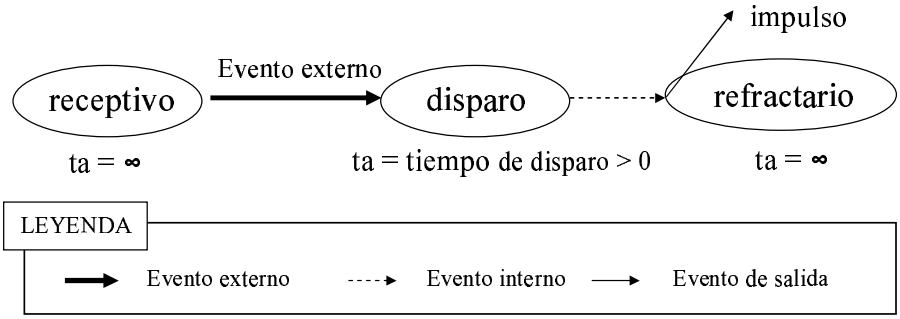

Figura 6.16: Neurona de un único disparo.

Ejercicio 6.2

En la Figura 6.17 se muestra una red compuesta por cuatro neuronas de un único disparo, como las descritas en el Ejercicio 6.1, y un generador. Describa este sistema compuesto empleando el formalismo DEVS.

Explique cómo pueden aplicarse redes neuronales en la búsqueda del camino más corto entre dos puntos (vea la explicación dada en el Apartado 2.4 de *lectura6a.pdf*).

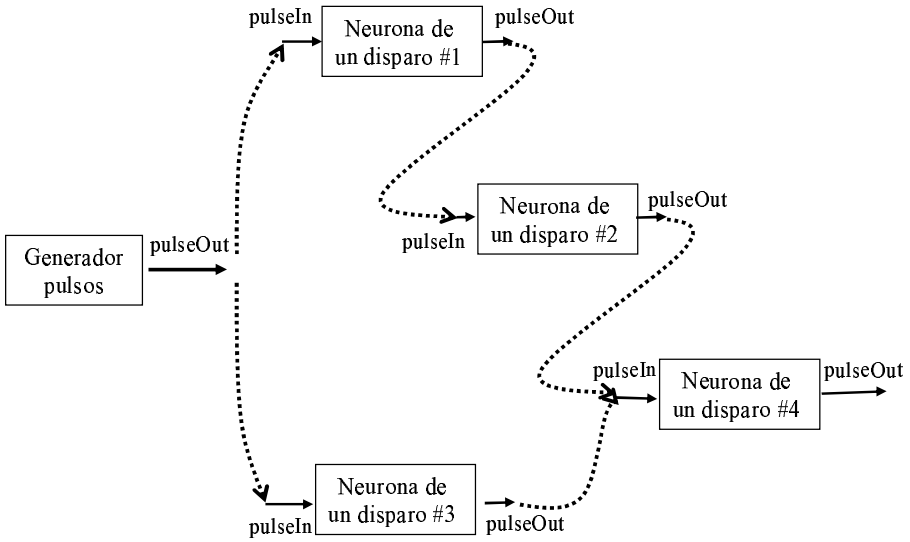

Figura 6.17: Red neuronal.

Ejercicio 6.3

Inspeccione los modelos *fireOnceNeuron* y *fireOnceNeuronNet* contenidos en el paquete *pulseModels*, que está en el fichero *IllustrationCode.zip*. Discuta las similitudes y diferencias entre el comportamiento descrito en estos modelos y los descritos para la neurona y la red en los Ejercicios 6.1 y 6.2.

Ejercicio 6.4

Modifique el modelo DEVS de la neurona que ha realizado en el Ejercicio 6.1, de modo que si llega un evento externo mientras la neurona está en el estado *disparo*,

entonces se cancela el impulso de salida que estaba planificado y la neurona pasa inmediatamente al estado *refractario*.

Modifique la clase *fireOnceNeuron* con el fin de describir este comportamiento.

Ejercicio 6.5

Modifique el modelo DEVS de la neurona que ha realizado en el Ejercicio 6.1 de modo que si llega un evento externo mientras la neurona está en el estado *disparo*, entonces se cancela el impulso de salida que estaba planificado y se planifica un nuevo impulso de salida para dentro de un tiempo determinado por el parámetro *tiempo de disparo*. Una vez generado el impulso, la neurona pasa al estado refractario.

Modifique la clase *fireOnceNeuron* con el fin de describir este comportamiento.

6.9. SOLUCIONES A LOS EJERCICIOS

Solución al Ejercicio 6.1

El modelo de la neurona tiene un puerto de entrada y un puerto de salida, ambos de eventos discretos. Llamemos *Input* y *Output* a estos dos puertos. Los eventos en estos puertos tienen valor {pulse}.

$$\begin{aligned} X &= (\text{“}Input\text{”}, \{\text{pulse}\}) \\ Y &= (\text{“}Output\text{”}, \{\text{pulse}\}) \end{aligned}$$

El modelo tiene dos variables de estado:

- La variable $fase$ describe el modo de funcionamiento en que se encuentra la neurona: receptivo, disparo o refractario. Así pues, tiene tres posibles valores: {“*receptivo*”, “*disparo*”, “*refractario*” }.
- La variable σ almacena el tiempo que queda hasta que se produzca, en ausencia de eventos externos, la siguiente transición interna.

Dado que la variable $fase$ puede tomar valores reales positivos, incluyendo el cero, el conjunto de estados secuenciales es:

$$S = \{\text{“}receptivo\text{”}, \text{“}disparo\text{”}, \text{“}refractario\text{”}\} \times \mathbb{R}_0^+$$

La función de transición externa define el nuevo estado de la neurona cuando llega un evento al puerto de entrada. Si la neurona está en el estado “*receptivo*” y se recibe un evento, entonces pasa al estado “*disparo*”. Mientras a neurona está en el estado “*disparo*” o “*refractario*”, ignora los eventos de entrada.

$$\delta_{ext}((fase, \sigma), e, x) = \begin{cases} (\text{“}disparo\text{”}, tiempoDeDisparo) & \text{si } fase = \text{“}receptivo\text{”} \\ (\text{“}disparo\text{”}, \sigma - e) & \text{si } fase = \text{“}disparo\text{”} \\ (\text{“}refractario\text{”}, \infty) & \text{si } fase = \text{“}refractario\text{”} \end{cases}$$

La función de salida, la función de transición interna y la función de avance en el tiempo son las siguientes:

$$\begin{aligned} \lambda\,(fase, \sigma) &= (\text{“}Output\text{”}, \{\text{pulse}\}) \\ \delta_{int}\,(fase, \sigma) &= (\text{“}refractario\text{”}, \infty) \\ ta\,(fase, \sigma) &= \sigma \end{aligned}$$

Si bien no se pide en el enunciado, podría realizarse la especificación DEVS paralelo de la neurona. Además de los elementos anteriores, la especificación debería contener:

– La función de transición confluyente, que especifica el cambio en el estado que se produce cuando llega un evento externo en el mismo instante en que está planificada una transición interna. En este caso:

$$\delta_{con}\,((fase, \sigma), e, x) = \delta_{int}\,(fase, \sigma)$$

– Contemplar la posibilidad de que lleguen simultáneamente varios eventos al puerto de entrada. En este modelo el comportamiento no depende de que se reciba un único evento o varios simultáneamente.

Solución al Ejercicio 6.2

El modelo compuesto engloba los cinco componentes: el generador y las cuatro neuronas. La interfaz del modelo compuesto tiene únicamente un puerto de salida, al cual está conectado el puerto de salida de la neurona #4. Llamaremos “pulseOut” al puerto de salida del modelo compuesto. La descripción formal DEVS paralelo del modelo compuesto es la siguiente.

$$N = \langle\ X,\ Y,\ D,\ \{M_d \mid d \in D\},\ EIC,\ EOC,\ IC\ \rangle \tag{6.1}$$

donde:

$X = \emptyset$	El modelo compuesto no tiene ningún puerto de entrada.
$Y =$ (“pulseOut”, {pulse})	Puerto de salida del modelo compuesto y valor de los eventos.

$D = \{G, N_1, N_2, N_3, N_4\}$	Conjunto de nombres de los componentes.
$M_G, M_{N_1}, M_{N_2}, M_{N_3}, M_{N_4}$	Cada componente es un modelo DEVS.
$\text{EIC} = \emptyset$	No hay conexiones externas de entrada.
$\text{EOC} = \{(N_4, \text{“pulseOut”}), (N, \text{“pulseOut”})\}$	Conexión del puerto de salida de N_4 al puerto de salida del modelo compuesto.

$$\text{IC} = \left\{ \begin{array}{l} ((G, \text{“pulseOut”}), (N_1, \text{“pulseIn”})), \\ ((G, \text{“pulseOut”}), (N_3, \text{“pulseIn”})), \\ ((N_1, \text{“pulseOut”}), (N_2, \text{“pulseIn”})), \\ ((N_2, \text{“pulseOut”}), (N_4, \text{“pulseIn”})), \\ ((N_3, \text{“pulseOut”}), (N_4, \text{“pulseIn”})) \end{array} \right\} \quad \text{Conexiones internas.}$$

Los modelos de redes de neuronas pueden emplearse para encontrar el camino más corto a través de una red. Para ello, hay que transformar las distancias entre nodos en valores equivalente del tiempo. Este valor se asigna como *tiempo de disparo* de la neurona que representa el nodo de la red.

Por ejemplo, en la red mostrada en la Figura 6.17, el impulso emitido por el generador recorre dos caminos concurrentemente hasta que llega a la neurona final (la neurona #4). Dependiendo de la suma de retardos a lo largo de cada camino, el impulso proveniente de cada camino alcanzará en un instante u otro la neurona final. El instante en el cual la neurona final emite su impulso de salida permite calcular el camino más corto a través de la red.

Con el fin de poder reconstruir el camino, puede modificarse el modelo de la neurona de manera que desde cada neurona “receptora” sea posible identificar qué neurona “emisora” ha disparado el impulso que ha llegado en primer lugar a dicha neurona “receptora”.

Solución al Ejercicio 6.3

El modelo tiene dos variables de estado, *phase* y *sigma*, que ya están definidas en la superclase y por tanto no se definen en **fireOnceNeuron**. Asimismo, la función de avance en el tiempo está definida por defecto de manera que devuelve el valor *sigma*.

El comportamiento del modelo de la neurona está definido mediante los métodos mostrados a continuación: inicialización, funciones de transición interna, externa y confluyente, y de salida.

```
public void initialize() {
   super.initialize();
   passivateIn("receptive");
}

public void deltext(double e,message x) {
   Continue(e);
   if ( phaseIs("receptive") && somethingOnPort(x,"in") )
      holdIn("fire",fireDelay);
}

public void deltint() {
   if (phaseIs("fire"))
      passivateIn("refract");
}

public void deltcon(double e,message x) {
   deltint();
}

public message out() {
   if (phaseIs("fire"))
      return outputNameOnPort("pulse","out");
   else
      return new message();
}
```

A continuación, se describen las acciones ejecutadas en cada uno de los métodos. El método `initialize()` asigna valor inicial a las variables de estado:

```
public void initialize() {
   super.initialize();
   passivateIn("receptive");
}
```

El método `passivateIn("receptive")` asigna el valor "*receptive*" a la variable de estado *phase* y asigna a la variable de estado *sigma* el valor *INFINITY*. La consecuencia de ello es que en ausencia de eventos externos el modelo permanece indefinidamente en el modo "*receptive*".

El siguiente método describe la función de transición externa:

```
public void deltext(double e,message x) {
   Continue(e);
   if ( phaseIs("receptive") && somethingOnPort(x,"in") )
      holdIn("fire",fireDelay);
}
```

Las acciones realizadas en la función son:

1. Se ejecuta el método `Continue(e)`, que está definido de la manera siguiente:

   ```
   public void Continue(double e) {
      if (sigma < INFINITY) sigma = sigma - e;
   }
   ```

   Si *sigma* vale *INFINITY*, entonces no se modifica su valor. En caso contrario, se reduce el valor de *sigma* en e. Es decir: $sigma = sigma - e$.

2. Si el valor de la variable de estado *phase* es "*receptive*", entonces se ejecuta `holdIn("fire",fireDelay)`, cuyo efecto es asignar a *phase* el valor "*fire*" y a *sigma* el valor *fireDelay* (parámetro *tiempo de disparo*).

Puede verse que estas acciones corresponden con la función de transición externa definida al responder al Ejercicio 6.1.

$$\delta_{ext}((fase, \sigma), e, x) = \begin{cases} (\text{“}disparo\text{”}, tiempoDeDisparo) & \text{si } fase = \text{“}receptivo\text{”} \\ (\text{“}disparo\text{”}, \sigma - e) & \text{si } fase = \text{“}disparo\text{”} \\ (\text{“}refractario\text{”}, \infty) & \text{si } fase = \text{“}refractario\text{”} \end{cases}$$

El siguiente método describe la función de transición interna:

```
public void deltint() {
   if (phaseIs("fire"))
      passivateIn("refract");
}
```

La acción realizada al ejecutar este método es la siguiente: si *phase* tiene el valor "*fire*", entonces la variable de estado *phase* se pone al valor "*refract*" y *sigma* al valor *INFINITY*. Esta definición concuerda con la dada al resolver el Ejercicio 6.1, donde se ha supuesto que la función sólo es invocada cuando el modelo se encuentra en la fase "*disparo*".

$$\delta_{int}\left(fase, \sigma\right) = (\text{“}refractario\text{”}, \infty)$$

La función de transición confluyente es igual a la función de transición interna (véase el método *deltcon*).

El comportamiento de la función de salida es el siguiente (véase el método *out*): si *phase* vale “*fire*”, entonces la función de salida envía un evento de valor “*pulse*” a través del puerto de salida “*out*”.

A continuación, puede verse cómo en la clase **fireOnceNeuronNet** se define el puerto de salida, los componentes de la red de neuronas, y las conexiones entre ellos y con el puerto de salida.

```
addOutport("out");

/* New a pulseGenr object */
ViewableAtomic pg = new pulseGenr("pg",100);
add(pg);
/* New Fireonce Neuron object 1 */
ViewableAtomic fon1 = new fireOnceNeuron("fireonce-neuron1",f1_firedelay);
add(fon1);
/* New Fireonce Neuron object 2 */
ViewableAtomic fon2 = new fireOnceNeuron("fireonce-neuron2",f2_firedelay);
add(fon2);
/* New Fireonce Neuron object 3 */
ViewableAtomic fon3 = new fireOnceNeuron("fireonce-neuron3",f3_firedelay);
add(fon3);
/* New Fireonce Neuron object 4 */
ViewableAtomic fon4 = new fireOnceNeuron("fireonce-neuron4",f4_firedelay);
add(fon4);

/*
** pg:   output ---> fon1:input
** pg:   output ---> fon3:input
** fon1: output ---> fon2:input
** fon2: output ---> fon4:input
** fon3: output ---> fon4:input
*/
addCoupling(pg,"out",fon1,"in");
addCoupling(pg,"out",fon3,"in");
addCoupling(fon1,"out",fon2,"in");
addCoupling(fon2,"out",fon4,"in");
addCoupling(fon3,"out",fon4,"in");
addCoupling(fon4,"out",this,"out");
```

Solución al Ejercicio 6.4

Sólo sería necesario modificar la función de transición externa, que sería la siguiente:

$$\delta_{ext}((fase, \sigma), e, x) = \begin{cases} (\text{“}disparo\text{”}, tiempoDeDisparo) & \text{si } fase = \text{“}receptivo\text{”} \\ (\text{“}refractario\text{”}, \infty) & \text{si } fase = \text{“}disparo\text{”} \\ (\text{“}refractario\text{”}, \infty) & \text{si } fase = \text{“}refractario\text{”} \end{cases}$$

Esta función de transición externa podría codificarse de la forma siguiente:

```
public void deltext(double e,message x) {
   if ( phaseIs("receptive") && somethingOnPort(x,"in") )
      holdIn("fire",fireDelay);
   else
      passivateIn("refract");
}
```

Solución al Ejercicio 6.5

Sería necesario modificar únicamente la función de transición externa, que sería la siguiente:

$$\delta_{ext}((fase, \sigma), e, x) = \begin{cases} (\text{“}disparo\text{”}, tiempoDeDisparo) & \text{si } fase = \text{“}receptivo\text{”} \\ (\text{“}disparo\text{”}, tiempoDeDisparo) & \text{si } fase = \text{“}disparo\text{”} \\ (\text{“}refractario\text{”}, \infty) & \text{si } fase = \text{“}refractario\text{”} \end{cases}$$

Esta función de transición externa podría codificarse de la forma siguiente:

```
public void deltext(double e,message x) {
   if ( ( phaseIs("receptive") || phaseIs("fire") ) && somethingOnPort(x,"in") )
      holdIn("fire",fireDelay);
   else
      passivateIn("refract");
}
```

Índice alfabético

Bibliografía

Cellier, F. C. (1991), *Continuous System Modeling*, Springer-Verlag.

Chow, A. (1996), 'Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator', *Transactions of the Society for Computer Simulation International* **13**(2), 55–68.

Gardner, M. (1970), 'The fantastic combinations of John Conway's new solitaire game of life', *Scientific American* **23**(4), 120–123.

Kelton, W. D., Sadowski, R. P. & Sadowski, D. A. (2002), *Simulation with Arena*, McGraw-Hill.

Klir, G. J. (1985), *Architecture of Systems Complexity*, Sauders, New York.

Ljung, L. & Torkel, G. (1994), *Modeling of Dynamic Systems*, Prentice-Hall.

Mather, J. (2003), *The DEVSJAVA Visualization Viewer: a Modular GUI that Visualizes the Structure and Behavior of Hierarchical DEVS Models*, Master Thesis, Dept. Electrical and Computer Engineering, The University of Arizona.

Pedgen, C. D., Shannon, R. E. & Sadowsky, R. P. (1995), *Introduction to Simulation Using SIMAN*, McGraw-Hill.

Urquia, A. (2003), *Simulación - Texto Base de Teoría*, Texto base de la asignatura "Simulación", de 3^{er} curso de la Ingeniería Técnica en Informática de Gestión de la UNED.

Wolfram, S. (1986), *Theory and Application of Cellular Automata*, World Scientific, Singapore.

Zeigler, B. P. (1976), *Theory of Modelling and Simulation*, Wiley Interscience.

Zeigler, B. P. (1990), *Object-Oriented Simulation with Hierarchical, Modular Models*, Academic Press.

Zeigler, B. P. (2003), 'DEVS today: recent advances in discrete event-based information technology', 11^{th} *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, MASCOTS 2003,* pp. págs. 148–161.

Zeigler, B. P., Praehofer, H. & Kim, T. G. (2000), *Theory of Modeling and Simulation*, Academic Press.

Zeigler, B. P. & Sarjoughian, H. S. (2005), *Introduction to DEVS modeling and simulation with JAVA: developing component-based simulation models*, Disponible en: http://www.acims.arizona.edu/SOFTWARE/software.shtml.