Biblioteca Digital
F C E N - U B A

BIBLIOTECA CENTRAL LUIS F LELOIR
BIBLIOTECA CENTRAL LELOIR
FACULTAD DE CIENCIAS EXACTAS Y NATURALES UBA

Tesis de Posgrado

Modelos celulares de eventos discretos con demoras explícitas

Wainer, Gabriel A.

1998

Tesis presentada para obtener el grado de Doctor en Ciencias de la Computación de la Universidad de Buenos Aires

Este documento forma parte de la colección de tesis doctorales y de maestría de la Biblioteca Central Dr. Luis Federico Leloir, disponible en digital.bl.fcen.uba.ar. Su utilización debe ser acompañada por la cita bibliográfica con reconocimiento de la fuente.

This document is part of the doctoral theses collection of the Central Library Dr. Luis Federico Leloir, available in digital.bl.fcen.uba.ar. It should be used accompanied by the corresponding citation acknowledging the source.

Cita tipo APA:
Wainer, Gabriel A.. (1998). Modelos celulares de eventos discretos con demoras explícitas. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.
http://digital.bl.fcen.uba.ar/Download/Tesis/Tesis_3092_Wainer.pdf

Cita tipo Chicago:
Wainer, Gabriel A.. "Modelos celulares de eventos discretos con demoras explícitas". Tesis de Doctor. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.
http://digital.bl.fcen.uba.ar/Download/Tesis/Tesis_3092_Wainer.pdf

EXACTAS UBA
Facultad de Ciencias Exactas y Naturales

UBA
Universidad de Buenos Aires

Dirección: Biblioteca Central Dr. Luis F. Leloir, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.
Intendente Güiraldes 2160 - C1428EGA - Tel. (++54 +11) 4789-9293

Contacto: digital@bl.fcen.uba.ar

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

TESIS DE DOCTORADO

Modelos celulares de eventos discretos con demoras explícitas.

Autor: Gabriel A. Wainer

DIRECTOR: Prof. Norbert Giambiasi

Septiembre de 1998.

Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires
Argentina

http://www.dc.uba.ar

№3092

A mi amada familia.
La que se fue, la que está, la que será.

DISCRETE EVENTS CELLULAR MODELS WITH EXPLICIT DELAYS.

Abstract

This work is devoted to introduce several formal descriptions used to model and simulate cell-shaped spaces. The paradigms are based on the DEVS and Cellular Automata formalisms, combined with transport delays and inertial delays. The specification formalisms have been defined for binary or three-states cell spaces, and have been extended to other domains. The delay concepts belong to the digital circuits domain, and have been adapted to the Cellular Automata paradigm, being one of the main contributions of the present work. The formalims allow the automatic definition for the cell spaces, easing the model verification, allowing the cost-effective development of simulators. A tool was built with the goal to implement the formalism, allowing to verify empirically the performance of the proposed solutions. Development times were considered, with special attention to the testing and mantainance costs. An abstract simulation mechanism was also proposed, with the goal to improve the execution times of the cell spaces based on the flattening of the hierarahical models. This approach allowed to improve the execution times up to one order of magnitude. It must be noticed that a model can be built automatically using its specification, easing the verification problems and allowing rapid development of the simulators. The use of a formal mechanism allowe to reduce the development times, due to the improvements obtained in the testing and maintainance phases. The use of this formal mechanism also allows automatic verification of the model structure, permitting the programmer to focus only in the development of the models to be implemented.

Keywords: Discrete Events simulation, Cellular Automata, Transport Delays, Inertial Delays, Modelling Methodologies, Object-Oriented simulation.

Resumen

En un plano conceptual, este trabajo se dedica a presentar diversas descripciones formales para modelar y simular espacios de celdas. Los paradigmas propuestos constituyen extensiones a los formalismos DEVS y Autómatas Celulares, agregando la posibilidad de tener demoras de transporte o inerciales. Los formalismos de especificación han sido definidos para espacios de celdas binarios o de tres estados, y se han extendido para otros dominios. Los conceptos de demoras pertenecen al dominio de los circuitos digitales, y han sido adaptados a los autómatas celulares, constituyendo una de las contribuciones principales del trabajo. Los formalismos permiten la definición automática de los espacios, y facilita la verificación de los modelos, permitiendo el desarrollo eficiente y efectivo de simuladores. Desde un punto de vista práctico, se construyó un entorno de modelización y simulación basado en los formalismos propuestos. Esta permitió verificar de forma empírica la eficiencia de las soluciones propuestas, con respecto a los tiempos de desarrollo y chequeo. Se propuso a su vez un mecanismo de simulación achatado de los espacios de celdas que permitió mejorar los tiempos de ejecución en hasta un orden de magnitud. Debe hacerse notar que puede realizarse la construcción automática de un modelo a partir de su especificación, lo que simplifica el problema de verificación del modelo y permite un desarrollo rápido de los simuladores, debido a las mejoras obtenidas en las fases de chequeo y mantenimiento.

Palabras clave: Simulación de eventos discretos, Autómatas Celulares, Demoras de Transporte, Demoras Inerciales, Metodologías de Modelización, Simulación Orientada a Objetos.

Agradecimientos

Debo agradecer a:

Profesores Hugo Scolnik (FCEN/U.B.A., Argentina) y François Vernadat (LIPM, Metz, Francia), por haber juzgado el trabajo;

Profesores Jean Claude Bertrand (Director del IUSPIM, Université d'Aix-Marseille III, Francia), Alfredo Olivero (FCEN/U.B.A., Argentina), Sergio Junco (U.N.R., Argentina), y M.Conf. Claudia Frydman (Université d'Aix-Marseille III, Francia), por haber participado de este Jurado;

Agradezco al Profesor Norbert Giambiasi, que ha dirigido esta Tesis, de quien he adquirido toda clase de conocimientos en mis estadías en Marsella;

Agradezco, en particular:

A la Dra. Claudia Frydman, por su soporte, colaboración y sugerencias en todo el trabajo realizado;

Al Lic. Roberto J.G. Bevilacqua, sin el cual nunca habría llegado hasta este momento;

A los Lic. Amir Barylko y Jorge Beyoglonián, que trabajaron arduamente en la construcción de la herramienta CD++;

A la Lic. Irene Loiseau, que me empujó a esta instancia;

A todos los que soportaron mis idas y venidas; entre ellos, muy especialmente a Raúl V., así como a Sergio Z., Alejandro T. y al Lic. J.M. Santos (que siempre tuvo un consejo sabio);

Agradezco, fundamentalmente y ante todo, a mis padres (Carlos, que todavía me guía, y Noemí, que siempre me acompaña), a mis hermanos María Julia y Héctor Joaquín (y toda su prole);

Finalmente, agradezco a Diana Ruth, que siempre estuvo donde hacía falta.

1. INTRODUCCIÓN

En la actualidad existe una gran variedad de técnicas de simulación por computadora usadas para estudiar sistemas complejos. La utilización de técnicas de modelado formales pueden mejorar la construcción de los simuladores, ayudando a verificar el comportamiento de los modelos de implementación.

El impacto de la tecnología de computadoras ha tenido gran influencia en el desarrollo de técnicas de simulación, y en la actualidad existe hardware, interfaces con el usuario y herramientas de programación que influenciaron los métodos teóricos existentes. Existen diversos paradigmas que han sido desarrollados para especificar formalmente sistemas dinámicos; cada uno de ellos está mejor adaptado que los otros para modelar distintos tipos de sistemas. Algunos de estos paradigmas han sido desarrollados para modelar sistemas de eventos discretos (Discrete Events Dynamic Systems - DEDS), que tienen algunas características especiales que hacen que su modelado sea distinto que el de sistemas de variables continuas (usualmente descriptos por medio de ecuaciones diferenciales).

Un paradigma de modelado de sistemas de eventos discretos debe permitir la construcción de modelos usando variables discretas y tiempo continuo. Sin embargo, muchos de los paradigmas existentes usados para modelar estos sistemas usan bases de tiempo discretas. La utilización de un paradigma de tiempo continuo tiene diversas ventajas. Por un lado, permite mejor representación del tiempo, lo que facilita obtener mayor precisión en las simulaciones sin necesidad de usar segmentos discretos de tiempo que sean muy pequeños. En consecuencia, en la mayoría de los casos el número de ciclos de simulación puede reducirse, y los requerimientos de tiempo del procesador disminuyen.

Este trabajo introduce varios formalismos que permiten describir modelos celulares. El nuevo paradigma está basado en los formalismos DEVS [Zei76] y de autómatas celulares [Wol86]. Será llamado Cell-DEVS, y utilizará demoras de transporte o inerciales [WG98].

El objetivo primordial de este nuevo formalismo es la provisión de un mecanismo de simulación de sistemas basados en celdas que permita que un modelador pueda mejorar sus tiempos de desarrollo y chequeo. En un plano práctico, el objetivo es obtener un entorno de modelización y simulación de eventos discretos que permita a un usuario no especialista describir fácilmente un modelo de celdas a tres estados y con distintos tipos de demoras (utilización de bases de modelos, descripciones jerárquicas, modelos modulares) y realizar una simulación. En particular, se tratará de modelar sistemas de tráfico urbano complejos.

Se permitirá que el usuario defina el fenómeno a modelar por medio de un lenguaje de especificación de muy alto nivel. De esta forma se asegura, además, la validación automática de las simulaciones, que son independientes de los modelos: los formalismos subyacentes implican que un error en la simulación sólo dependerá de la especificación del modelo a simular. De esta forma se pueden plantear de forma simple y segura diversos modelos para analizar el comportamiento del sistema real ante cambios.

La propuesta se basa en considerar que cada una de las celdas del campo que forma el autómata es un modelo básico DEVS, y a su vez se trata de incluir distintos tipos de demoras en las celdas. Se trata de permitir la modelización y simulación eficiente de sistemas de eventos discretos que pueden ser descriptos como espacios de celdas.

Se define un lenguaje de especificación de modelos de celdas, cuya implementación es incorporada a la jerarquía de modelos propuesta por Zeigler. De esta forma, un modelador no experto podrá realizar experiencias variadas de modelización y simular sus modelos sin necesidad de tener conocimientos de informática, y sin conocer detalles de implementación de la técnica de simulación usada. Por medio de esta técnica se permite, a su vez, validar automáticamente los modelos incorporados, lo que permitirá proveer seguridad en las simulaciones. En la actualidad el lenguaje ya ha sido definido y se está estudiando su aplicabilidad, para incorporar posibles modificaciones.

Además de establecer los mecanismos necesarios para simular una célula, fue necesario definir un modelo acoplado celular. A su vez se definió una técnica de simulación abstracta para estos modelos, y se analizaron técnicas para realizar el "achatamiento" de los modelos jerárquicos previo a su incorporación a las bases de modelos (debido a motivos de eficiencia en la simulación, ya que la jerarquía de modelos impone un overhead importante).

Habiendo finalizado con estas definiciones, se propuso una extensión al formalismo de celdas DEVS que permite incluir construcciones del tipo "transport-delay". A su vez se propuso la extensión del formalismo (que en un principio

sólo permite estados binarios) para incluir un tercer estado. También se integraron los formalismos con construcciones del tipo de demora inercial (inertial delay), que permiten representar semánticas con desalojo.

Los trabajos se basaron en el formalismo de Zeigler ya que en este los intervalos de tiempo entre ocurrencias de eventos son variables, lo que tiene algunas ventajas: en los formalismos con una única granularidad es difícil describir los modelos donde hay muchos procesos operando en distintas escalas de tiempo, y la simulación no es eficiente ya que los estados deben actualizarse en el momento con menor incremento de tiempo, lo cual desperdicia recursos de las computadora cuando se aplica a los procesos mas lentos. Los modelos a tiempo continuo tienen la ventaja de permitir mayor precisión de los modelos (tanto conceptuales como de implementación), así como una reducción sustancial en los requerimientos de recursos de la computadora (especialmente, tiempo de procesamiento).

Por otro lado, los autómatas celulares son modelos a tiempo discreto, lo que reduce la precisión, la eficiencia de las simulaciones y el grado de paralelismo del modelo. Sin embargo, las transiciones de estado de cada celda suelen ser independientes de otras del autómata, y un autómata puede comportarse de forma muy inhomogénea. Estos problemas pueden evitarse por medio del uso de un modelo de autómatas celulares con relojes asincrónicos, y manejado por eventos. Si se usan relojes independientes, el tiempo de iteración para cada celda puede ser diferente.

Estos formalismos fueron integrados con construcciones del tipo de demora inercial (inertial delay), que permiten representar semánticas con desalojo. El significado de la construcción es que, si no se mantiene un valor de entrada para una función durante el tiempo suficiente (la demora inercial), el cambio de estado no debe realizarse. En cambio, si dicho valor se mantiene durante ese tiempo, el estado cambia luego de la demora. La construcción también se usa en el campo de diseño de hardware, ya que ciertos circuitos sólo cambian de estado dependiendo de la entrada de energía y de la duración de esta entrada. Aquí también puede usarse la construcción en otra clase de fenómenos complejos, ayudando al modelador a representar comportamientos de esta clase.

Los formalismos propuestos fueron incorporados a la jerarquía propuesta por Zeigler, y los simuladores abstractos que permitirán simular este nuevo tipo de modelos fueron implementados. La idea es generar, en base a parámetros, un modelo celular completo.

También se ha realizado un estudio exhaustivo de soluciones existentes que permitan realizar simulación distribuida de los citados modelos. En muchos casos, las simulaciones de sistemas reales son poco eficientes. El principal problema de velocidad está relacionado con la complejidad de los modelos a especificar, y a su vez tiene causas estadísticas, ya que para ser estadísticamente significativa, una simulación debe generar un número suficiente de evoluciones típicas del sistema.

Un medio obvio de obtener una simulación mas rápida es dedicar más recursos. En particular, se puede acelerar una simulación usando un sistema multiprocesador en vez de un solo procesador. Como la mayoría de las simulaciones son de sistemas que tienen muchos componentes operando en paralelo, parece razonable suponer que la simulación también explote el paralelismo inherente al sistema. La simulación paralela de eventos discretos (PDES), muchas veces llamada simulación distribuida, se refiere a la ejecución de un programa de simulación de eventos discretos en una computadora paralela. El uso de múltiples procesadores parece ser una aproximación promisoria para mejorar la velocidad.

Para lograr estos objetivos, desde el punto de vista teórico se propone terminar con la definición formal de los nuevos paradigmas analizados previamente, y su incorporación a la jerarquía de clases propuesta por Zeigler. También se hará evaluación experimental de la propuesta teórica de sincronización distribuida mencionada previamente en entornos Time-Warp.

El trabajo está organizado en ocho capítulos. En el capítulo 2 se hace una breve reseña de los formalismos utilizados. Se comienza con una reseña de temas básicos relacionados con el área de simulación de eventos discretos. Luego, se ataca una descripción de los formalismos DEVS y de Autómatas Celulares. Finalmente, se presentan algunos algoritmos de simulación paralela y distribuida.

El capítulo 3 está dedicado a la definición del formalismo Cell-DEVS. Aquí se comienza mostrando una definición formal de autómatas celulares (sincrónicos y asincrónicos). A continuación se definen los modelos atómicos Cell-DEVS (con demoras de transporte, con demoras inerciales, y de tres estados). Luego se ataca la especificación formal de los modelos Cell-DEVS acoplados (tanto modelos cerrados, como modelos con conexiones genéricas). Finalmente, se redefine el formalismo DEVS para permitir la inclusión de modelos acoplados Cell-DEVS en las definiciones originales.

El capítulo 4 encara los mecanismos de simulación abstracta utilizados. Se comienza analizando modelos de implementación Cell-DEVS. A continuación se analiza el mecanismo de simulación abstracta utilizado, seguido por los mecanismos de simulación achatada propuestos. Finalmente se analiza la definición de la ejecución paralela de los modelos implementados.

El capítulo 5 muestra con detalle un ejemplo de modelización y simulación de estos espacios de celdas. Se encara un modelo complejo de simulación de tráfico, mostrando su definición formal siguiendo el paradigma, y estudiando la ejecución del simulador asociado.

En el capítulo 6 se presentan herramientas de simulación desarrolladas durante este trabajo. Para comenzar, se presenta GAD, una herramienta genérica que permite simular modelos DEVS tradicionales, poniendo énfasis en su diseño y los mecanismos de simulación utilizados. A continuación se presenta una extensión de esta herramienta, llamada CD++, que permite la simulación de espacios de Cell-DEVS con demoras de transporte e inerciales.

El capítulo 7 enfoca los resultados obtenidos, mostrando distintas experiencias realizadas con autómatas celulares sincrónicos y asincrónicos, y comparando los resultados obtenidos al ejecutar modelos similares basados en modelos DEVS tradicionales y modelos Cell-DEVS.

El último capítulo presenta las conclusiones de este trabajo, y se remarcan los principales trabajos relacionados que están siendo llevados a cabo en la actualidad por el autor.

2. BACKGROUND

Al estudiar sistemas físicos complejos, suele ocurrir que la previsión de su comportamiento, así como lo es su comprensión global es difícil (o imposible). Las distintas metodologías y herramientas de simulación han permitido cumplir estos objetivos con una aproximación efectiva con respecto a los costos.

Las ventajas de la simulación son múltiples: puede reducirse el tiempo de desarrollo del sistema, las decisiones a tomar pueden chequearse artificialmente, un mismo modelo puede usarse muchas veces, etc. La simulación es de empleo mas simple que ciertas técnicas analíticas y precisa menos simplificaciones. Por medio de la simulación se pueden deducir consecuencias: como se plantean las reglas por las cuales se rige el sistema, durante la ejecución se pueden experimentar distintos casos especiales que serían imposibles de realizar en un laboratorio. Durante la simulación también se puede interactuar con el sistema, haciendo que reciba distintos eventos, pudiendo analizar los resultados de esta interacción.

El objetivo de esta sección será hacer una introducción a algunos conceptos generales relacionados con las técnicas de modelado y simulación.

2.1.1 Sistemas, modelos y paradigmas

De aquí en adelante se llamará **sistema** a una entidad real o artificial (una parte de una realidad), restringida por un entorno. Puede decirse que es un conjunto ordenado de objetos relacionados que atraviesan ciertas actividades, interactuando para cumplir ciertos objetivos.

Para lo que nos concierne, se usarán dos interpretaciones de la palabra sistema:

a) Un **sistema real** es una combinación de elementos con relaciones de estructura que se influencian mutuamente.
b) Un **sistema dinámico** es una construcción formal que da conceptos generales de modelado para distintas clases de disciplinas [Gia96].

Se llamará **modelo** a una representación inteligible (abstracta y consistente) de un sistema. En muchos casos no se puede resolver un problema directamente sobre un sistema real, por ende se razona sobre modelos. El proceso de pensar y razonar acerca de un sistema resaltando la reacción de un modelo se llama **modelado de sistemas**. Se llamará **paradigma** a un conjunto de conceptos, leyes y medios que sirven para definir un conjunto de modelos. Los modelos se construyen sobre un paradigma particular.

Ejemplo 1

Consideremos el diseño de un circuito digital. Si se analiza la medición de corrientes y tensiones del circuito se está razonando sobre un modelo eléctrico; en cambio, si se hace un estudio de las funciones booleanas que realiza el circuito, se razona sobre un modelo lógico. En este caso, el sistema se analiza usando dos paradigmas: el paradigma de Boole (para analizar el modelo lógico) y las ecuaciones diferenciales (para analizar el flujo de corrientes y tensiones en el circuito).

Para estudiar sistemas complejos, la idea es partir haciendo un modelo del sistema, y analizar las propiedades del sistema real estudiando el modelo. El resultado final será aplicar las soluciones o mejoras obtenidas sobre el sistema real modelado. En este proceso, el uso de técnicas formales facilita el desarrollo de los modelos ejecutables, debido a que el comportamiento de un modelo conceptual puede ser validado contra el sistema real, y a su vez, la respuesta del modelo ejecutable puede ser verificada contra la especificación. Un paradigma formal también debe considerar la dinámica del sistema, permitiendo modelar su comportamiento. También es bueno que se provea un mecanismo de descomposición para reflejar la naturaleza jerárquica de la mayoría de los fenómenos físicos. Un formalismo también debería facilitar la traducción de una especificación a un modelo ejecutable en una computadora.

2.1.2 Modelos y simulación

¿Porqué se hacen modelos de los sistemas? ¿Porqué se usa simulación? El motivo es que en muchos casos no se puede experimentar directamente sobre el sistema a estudiar, o se desea evitar costos, peligro, etc. En la actualidad existe una gran variedad de aplicaciones muy complejas para las que se usan modelos y/o simulación, que van desde manufactura hasta diseño de circuitos para computadoras, pasando por aplicaciones bélicas y estudio de experimentos complejos. Las características comunes a estos sistemas son su complejidad y la falta de herramientas de evaluación de desempeño adecuadas.

Distinguiremos, entonces, dos grandes grupos de métodos para modelar sistemas complejos:

a) **Analíticos**: en este caso los modelos están basados en razonamiento. Suelen ser simbólicos, y permiten obtener soluciones generales al problema considerado. La solución se construye usando las reglas de inferencia reconocidas como correctas en el paradigma usado para describir el modelo, y se obtienen bajo una forma general (en el sentido de que están bajo una forma simbólica). Las soluciones particulares serán obtenidas reemplazando los valores simbólicos por sus valores numéricos. La idea es que, una vez obtenida la solución, se aplica una variable al modelo obtenido y de esta forma se pueden encontrar soluciones particulares. Un formalismo analítico muy difundido son las ecuaciones diferenciales.

Un problema de estos métodos reside en que, al considerar sistemas complejos, éstos (con pocas excepciones) serán analíticamente intratables y numéricamente prohibitivos de evaluar. Por ende, para poder usar estos métodos para los problemas que existen en el mundo real, se debe simplificar el modelo a un nivel tal, que las soluciones obtenidas pueden alejarse de la realidad.

b) **Basados en simulación**: en ellos no existen soluciones generales sino que buscan soluciones particulares para el problema, por ende, se permite tratar cierta complejidad. El uso de simulación permite experimentación controlada, compresión del tiempo (una simulación puede ejecutar en mucho menos tiempo que el sistema real modelado), y análisis de sensibilidad. Otra gran ventaja es que su uso no afecta al sistema real, que puede seguir utilizándose (o no existir). Finalmente, la simulación es una herramienta efectiva de entrenamiento. Algunos problemas que existen en el uso de simulación son sus tiempo de desarrollo, que los resultados pueden tener divergencia con la realidad (precisan validación), y que para reproducir el comportamiento del sistema simulado se precisa colección extensiva de datos.

Por ende, si un problema es simple, es conveniente el uso de métodos analíticos ya que permiten obtener soluciones generales con seguridad. En cambio, si es complejo, usando simulación se pueden probar distintas condiciones de entrada que no serían posibles de probar analíticamente, y obtener resultados de salida significativos. Así se obtienen datos que pueden estudiarse para analizar algún comportamiento que interese.

Definición 1

La simulación es la reproducción del comportamiento dinámico de un sistema real en base a un modelo con el fin de llegar a conclusiones aplicables al mundo real [Gia96].

En general, el desarrollo de un simulador envuelve las siguientes etapas:

a) Planteo del problema: en esta fase se toma el sistema real a estudiar, y se trata de comprenderlo. Para ello, primero se trata de identificar el problema a resolver y se describe su operación en términos de objetos y actividades dentro de un marco físico. Luego, se trata de identificar las variables de entrada y salida del sistema, y se las categoriza. Las variables de entrada pueden ser de **decisión** (controlables) o **parámetros** (no controlables). También se trata de definir **medidas de desempeño** del sistema (en función de las variables de salida), y una **función objetivo** (una combinación de algunas de las medidas). Habiendo finalizado esta especificación, se trata de construir una estructura preliminar del modelo, interrelacionando las variables del sistema y las medidas de desempeño, introduciendo las presunciones y simplificaciones que se consideren adecuadas. Finalmente se construye una estructura del modelo más detallada, identificando todos los objetivos con sus atributos e interfaces.

b) Recolección y análisis de los datos de entrada: en esta fase se estudia el sistema real para obtener datos de entrada. Para ello se hace observación directa y recolección de los atributos seleccionados en la etapa anterior. Al estudiar el

flujo de las entidades a través del sistema, se trata de identificarlas con valores de tiempo. Otra cuestión importante en esta fase es seleccionar un tamaño de muestra estadísticamente válido, y un formato de datos procesable por computadora. Finalmente se decide qué datos serán tratados como aleatorios y cuáles se asumirán como determinísticos.

c) Modelización: en esta fase se construye un modelo del sistema con los aspectos que se quieren simular. Para ello se puede seguir una aproximación de **flujo físico**, basada en el flujo de entidades a través del sistema con sus puntos de procesamiento y reglas de decisión; o una aproximación de **eventos** (o cambios de estados), basado en la definición de variables de estado internas seguida por una descripción de la operación del sistema cuando ocurre un evento. El modelo se construye definiendo objetos, atributos, y métodos en el paradigma elegido.

d) Implementación: en esta etapa se elige un lenguaje de programación, y se construye una simulación del modelo que pueda ejecutarse en una computadora.

e) Verificación y validación del modelo: durante los pasos anteriores se construyeron tres modelos: el **conceptual** (especificación), el **lógico** (diseño) y el **ejecutable** (código). La verificación es un asunto de consistencia interna entre los tres modelos. La validación enfoca la correspondencia entre el modelo y la realidad. En base a los resultados obtenidos durante la validación y verificación, el modelo y su implementación deben refinarse.

f) Experimentación: en esta fase se hace evaluación estadística de las salidas del simulador para determinar algún nivel de precisión de las medidas de desempeño. Si el objeto en interés pasa por un comportamiento de período transitorio, se debe tener cuidado de hacer el análisis sobre estados estacionarios. Se hace diseño de experimentos de simulación basados en la repetición de la simulación con las variables de decisión en varios niveles.

g) Análisis de datos de salida: en la última fase se analizan las salidas de la simulación para comprender el comportamiento deseado del sistema. Estas salidas se usan para obtener respuestas con relación al comportamiento del sistema original.

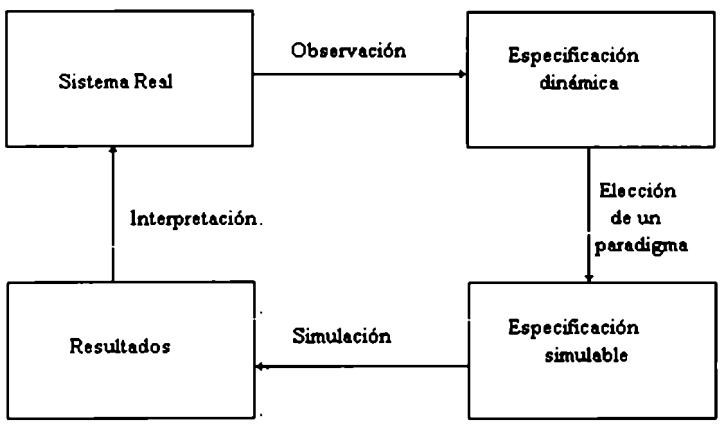


Figura 1. Pasos en el estudio de un sistema por medio de modelado [Gia96].

En función del nivel de intervención de la persona que decide sobre el sistema real, se definen cuatro niveles de uso de modelos para simulación:

- de **exploración**: se trata de adquirir mejor comprensión de las operaciones del sistema real;
- de **predicción**: se desea predecir el comportamiento del sistema real en el futuro;
- de **mejoras**: se trata de optimizar el desempeño del sistema real, analizando distintas alternativas; y
- de **concepción**: el sistema real no existe, y se trata de chequear distintas opciones antes de construirlo.

En cualquier caso, nos interesa tener **modelos ejecutables**, o sea, descripciones inteligibles sobre las que se pueden ejecutar algoritmos en tiempo finito. Para poder construir un modelo ejecutable, el sistema en consideración (y el modelo usado para representarlo) debe obedecer a dos principios fundamentales:

- **Causalidad**: el futuro no puede influenciar sobre el pasado. Es decir, que el estado del sistema en el instante actual **t** es independiente de todo lo que se pueda producir en tiempo posterior a **t**.

- **Determinismo**: el futuro del sistema puede determinarse a partir de su estado presente y de su pasado. Es decir, en todo instante t existe un valor positivo **e** tal que el comportamiento del sistema puede calcularse hasta t+e.

2.1.3 Categorías de paradigmas de modelado

Durante siglos el estudio y desarrollo de sistemas dinámicos estuvo centrado en el estudio de modelos basados en ecuaciones diferenciales ordinarias y parciales. Las ecuaciones diferenciales (que suelen establecer relaciones entre magnitudes físicas), suelen ser imposibles de resolver por completo, y se las debe acotar para llegar a una solución. Existe diversidad de métodos numéricos que permiten encontrar soluciones por aproximación a estas ecuaciones.

Por otro lado, la tecnología moderna ha permitido que el hombre cree sistemas dinámicos que no pueden ser descriptos fácilmente por medio de ecuaciones diferenciales ordinarias o parciales. Como ejemplos de tales sistemas podemos mencionar líneas de producción, redes de computadoras, sistemas de control de tráfico (en aire o tierra), sistemas de control militar, etc. La evolución en el tiempo de estos sistemas depende de interacciones complejas de varios eventos discretos y de su temporalidad (por ejemplo, llegada o partida de un trabajo; comienzo o finalización de una tarea, etc.). El "estado" de tales sistemas no cambia continuamente como en los sistemas físicos, sino en instantes discretos de tiempo [Ho89].

Para estudiar estos sistemas, se ha propuesto una gran variedad de paradigmas de modelado, que pueden clasificarse de acuerdo a distintos criterios:

- Con respecto a la **base de tiempo**, hay paradigmas a *tiempo continuo*, donde se supone que el tiempo evoluciona de forma continua (es un numero real), y a *tiempo discreto*, donde el tiempo avanza por saltos de un valor entero a otro (el tiempo es un entero).

- Con respecto a los conjuntos de **valores de las variables** descriptivas del modelo, hay paradigmas de *estados o eventos discretos* (las variables toman sus valores en un conjunto discreto), *continuos* (las variables son números reales), y *mixtos* (ambas posibilidades) [Gia96].

Ejemplo 2

. Ecuaciones diferenciales: paradigma de variables continuas a tiempo continuo.
. Ecuaciones en diferencias: paradigma de variables continuas a tiempo discreto.
. Máquinas de estados finitos: paradigma de variables discretas a tiempo discreto.
. DEVS: paradigma de variables discretas a tiempo continuo. Un modelo de eventos discretos cambia de estados por causa de un evento.

Con respecto a la caracterización del problema a modelar, los modelos pueden ser **prescriptivos** si formulan y optimizan el problema (en general son métodos analíticos) o **descriptivos** si describen el comportamiento del sistema (suelen ser métodos numéricos).

Por otro lado, un modelo se dice que es **determinístico** si todas las variables tienen certeza completa y están determinadas por sus estados iniciales y entradas. El modelo se dice **probabilístico** en el caso que una respuesta pueda tomar un rango de valores dado el estado inicial y sus entradas (si usa variables aleatorias se dice que el modelo es **estocástico**). En un modelo probabilístico, los cambios de estado del modelo se producen por medio de leyes aleatorias: las entradas al modelo son aleatorias (siendo el modelo determinista), ó el tiempo de llegada de los eventos es aleatorio [Zei96b].

De acuerdo a su entorno, los modelos son **autónomos** (no existen entradas) o **no autónomos** (existen entradas). Los autónomos sólo evolucionan en función del tiempo.

2.1.4 Simulación de Sistemas de Eventos Discretos

Los sistemas donde las variables son discretas a tiempo continuo reciben el nombre de *Sistemas Dinámicos de Eventos Discretos* (DEDS - Discrete Events Dynamic Systems) en oposición a los *Sistemas Dinámicos de Variables Continuas* (CVDS - Continuous Variable Dynamic Systems), que se describen por medio de ecuaciones diferenciales. Un paradigma de simulación para DEDS asume que el sistema simulado sólo cambia de estados en puntos discretos en tiempo simulado, o sea que el modelo cambia de estado ante ocurrencia de un **evento**.

Ejemplo 3

Un caso típico de un DEDS es una red de computadoras. Un modelo para analizar una red puede incluir variables de estado para indicar la longitud de las colas de mensajes, estado de los enlaces de comunicaciones (ocupado o libre), etc. Los eventos típicos pueden incluir la llegada de un mensaje en algún nodo en la red, la transmisión de un mensaje a otro nodo, fallas de componentes, etc.

Comparado con el modelado de sistemas de variables continuas (fundamentales en la física), el modelado de DEDS es un fenómeno relativamente reciente. A pesar de que pertenece al dominio de la Investigación Operativa (desde el punto de vista de que la Investigación de Operaciones puede pensarse como la técnica de operaciones y eventos de sistemas hechos por el hombre), este área recibió una fuerte influencia de las teorías de Sistemas de Control. En particular, los conceptos de dinámica (constantes de tiempo, tiempo de respuesta, frecuencia, controlabilidad) son importantes en el desarrollo de modelos y herramientas para DEDS [Ho89].

Es útil distinguir entre dos tipos de DEDS: los de topología fija y dinámica. La interacción entre componentes de topología fija puede describirse por un grafo en el cual los nodos representan componentes y los arcos representan los caminos posibles de interacción. Tales sistemas suelen representarse como redes de encolado con un conjunto fijo de estaciones, y clientes que se mueven entre estaciones. En ellos, las rutas posibles de los clientes son fijas (por ejemplo, sistemas de manufactura o redes de comunicación). En sistemas de topología dinámica las interacciones son arbitrarias y dinámicas. Aquí, los componentes de estos sistemas pueden moverse en cualquier dirección e interactuar con los demás (por ejemplo, en escenarios de campos de batalla o sistemas de radio móvil). Los sistemas de topología dinámica no suelen ser estrictamente DEDS, ya que los componentes se pueden mover de forma continua. Sin embargo, para el propósito de la simulación, el movimiento físico modelado puede discretizarse en el tiempo.

Para implementar una simulación de eventos discretos se debe incluir:

a) Un conjunto de variables de estado discretas.
b) Un planificador que contiene una lista cronológica de eventos (o mensajes) a tratar. Un evento es un cambio de estado que debe efectuarse a una hora ti $\in \boldsymbol{R}$ (el tiempo es continuo).
c) Un reloj global que indica el instante actual de simulación (hora de ocurrencia del evento presente).

2.1.5 Paradigmas de especificación de modelos DEDS

La aparición de DEDS ha provocado la necesidad de desarrollar paradigmas que permitan analizarlos formalmente. En los últimos años se ha atacado este problema, y en la actualidad existe una gran variedad de paradigmas. No puede decirse que ninguno de los formalismos existente sea mejor que otro para representar la variedad de comportamientos existentes en los sistemas reales, ya que dependiendo de diversos factores, algunos son mas naturales y producen simulaciones mas eficientes. Incluso existen paradigmas genéricos, pero no hay consenso para decidir qué modelo tiene el mayor potencial.

Existe una variedad de motivos para la existencia de formalismos de modelado para DEDS [Ho89]:

i) En estos sistemas, los eventos son discretos. Ello implica que, a pesar que se pueden analizar DEDS usando aproximaciones continuas (por ejemplo, usando modelos de difusión o analizando exclusivamente tasas en vez de las partes constituyentes), no es conveniente hacerlo. Como los estados reales de un DEDS son inherentemente discretos, es difícil evitar la naturaleza discreta del problema.

ii) A pesar que los componentes son eventos discretos, la mayoría de las mediciones de desempeño son de naturaleza continua, y suelen estar formuladas en términos de este tipo de variables (por ejemplo, rendimiento, tiempo de espera, ganancia, etc.).

iii) En los DEDS suelen ocurrir disturbios no planificados y/o caídas, que tienen características probabilísticas. Esto no implica que una aproximación totalmente determinista no tenga utilidad, sino que deben hacerse extensiones estocásticas.

iv) Los sistemas artificiales operan en planos jerárquicos, por ende, es necesario que un paradigma modele esta jerarquía. De esta forma, los detalles necesarios para modelar y controlar pueden ser distintos en cada nivel, conduciendo a un modelado más natural del sistema deseado.

v) Los modelos deben capturar el comportamiento dinámico y transitorio de los DEDS.

vi) Se desea que el paradigma sea ejecutable, ya que en algunos de ellos (como las máquinas de estados finitos o las cadenas de Markov) el número de estados discretos del modelo puede explotar de forma combinatoria, provocando requerimientos de cálculo no factibles. Por ende, el paradigma debe estar orientado a desempeño, distinguiendo entre modelos conceptuales y ejecutables.

2.1.6 Especificación de modelos simulables

La especificación de modelos ejecutables (ó simulables) implica la interacción de tres objetos básicos:

a. El **sistema real** en existencia, que se ve como una fuente de datos de comportamiento.

b. El **modelo**, que es un conjunto de instrucciones para generar datos comparables a los observables en el sistema real. El objetivo de la construcción de un modelo es el desarrollo de una representación simplificada y observable del comportamiento y/o estructura del sistema real. Un modelo es simplificador, un filtro de la realidad: no la representa ni de forma exacta ni de forma completa. Al crear el modelo, deben construirse relaciones de entrada/salida de forma tal que al hacer una variación de entrada, se pueda observar una variación de salida simular a la que ocurre en el sistema real. La definición del modelo debe considerar sus condiciones reales de utilización, incluyendo, además, las condiciones de funcionamiento del sistema real, y sus dominios de aplicación.

c. El **simulador**, que ejecuta las instrucciones del modelo para generar su comportamiento.

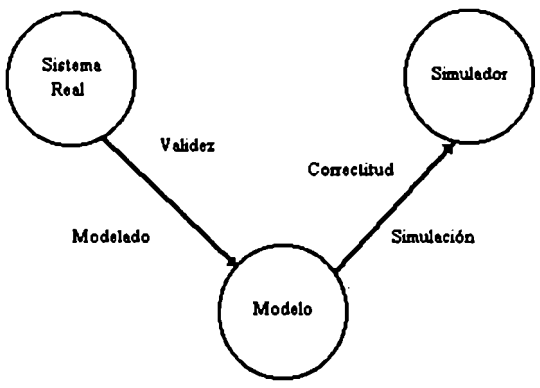


Figura 2 - Relaciones de modelado y simulación [Zei90].

Estos objetos básicos están ligados por dos relaciones:

- La relación de **modelado**, que relaciona al sistema real y al modelo, y define cómo el modelo representa al sistema o la entidad modelada. En términos generales, un modelo puede considerarse válido si los datos generados coinciden con los producidos por el sistema real en un marco experimental de interés (**validez** del modelo conceptual).

- La relación de **simulación** relaciona al modelo y al simulador, y representa cuán fielmente el simulador puede llevar a cabo las instrucciones del modelo. Concierne a la exactitud con la cual la computadora trata las instrucciones del modelo. Esta exactitud se llama **correctitud** del programa o del modelo simulable [Zei90].

Al hacer un modelo de un sistema real se debe, además, poner en consideración otros elementos:

a) Marco experimental: es un conjunto restringido de las circunstancias sobre las cuales se observa el sistema (el modelo se restringe a este marco). El marco debe estar determinado por el modelador, y define las condiciones de utilización del modelo. Refleja los objetivos del modelador que impactan en la construcción, experimentación y validación del modelo.

b) Computadora: al implementar el modelo en la computadora hay una reducción extra de la capacidad. Es una simplificación más del modelo, ya que el modelo conceptual se pasa a un modelo informático, lo que implica un nuevo filtro debido a la limitación de los lenguajes de descripción y a la eficacia del proceso de ejecución (duración, memoria, precisión de las variables). Por ende, en pasaje también se pierde información.

2.1.7 Comportamiento y estructura

La teoría matemática de sistemas provee un marco para representar y estudiar sistemas dinámicos, distinguiendo entre la estructura (constitución interna) de un sistema y su comportamiento (su manifestación exterior). Al analizar un modelo, se puede decir informalmente que la estructura del mismo es su conjunto de instrucciones, y que su comportamiento es el conjunto de todos los datos posibles que pueden ser generados ejecutando sus instrucciones. Al considerar el comportamiento de un sistema, se pueden usar representaciones causales y empíricas. Las empíricas son registros reales de datos (historia en el tiempo de las variables) tomadas del sistema real, mientras que las causales se integran en unidades que pueden generar datos empíricos.

Si el modelo sólo permite observación del comportamiento, se lo llama **modelo de comportamiento** o **caja negra**, y al analizarlo, lo único que se considera son sus entradas/salidas. Si, por otro lado, el modelo representa la estructura del sistema modelado, se lo llama **modelo de estructura** o **caja blanca**. En este caso, se explicitan los componentes del sistema real (que pueden ser modelos de comportamiento simples). Aquí entran en juego conceptos de descomposición (cómo dividir un sistema en componentes) y acoplamiento (cómo combinar los componentes para reconstituirlo). Una tercera relación fundamental es la taxonomía que concierne las variantes admisibles de un componente, y sus especializaciones.

Ejemplo 4

Consideremos la representación de un modelo de un circuito simple. Para este caso se puede usar, por un lado, un modelo lógico basado en la representación de la función booleana correspondiente al circuito. Por otro lado, se puede usar un modelo consistente de componentes lógicos básicos, que se combinan para obtener la misma función booleana.

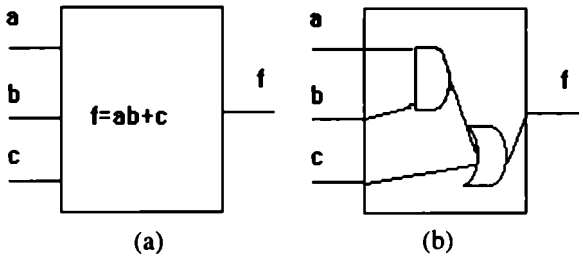


Figura 3. Ejemplo de un modelo de comportamiento (a) y de estructura (b) para el mismo sistema real.

En este caso, el primer modelo es de comportamiento, y el segundo es de estructura. En general, para desarrollar simulaciones es conveniente usar estos últimos, ya que permiten construir un modelo complejo en base a submodelos más simples. Definiendo los componentes y las uniones entre los mismos, se puede construir un modelo complejo en base a componentes simples.

En la gran mayoría de los casos, la complejidad del sistema a modelar implica que los modelos y sus implementaciones en una computadora sufran simplificaciones. Existen dos fuentes distintas de complejidad de los modelos:

- Cualitativa: esta complejidad se relaciona con la semántica del paradigma de modelado utilizado. Toma en cuenta la naturaleza variable de los componentes del sistema.
- Cuantitativa: depende del número de los componentes del modelo. Está ligado al sistema modelado y no al paradigma usado.

Para reducir la complejidad existente en el sistema modelado, puede atacarse alguna de ambas fuentes.

Ejemplo 5

Volviendo al caso de un modelo de circuitos lógicos, puede decirse que la complejidad cualitativa es baja, ya que existe un número pequeño y simple de componentes ({0,1}, {AND, OR, NOT}). Mientras tanto, la complejidad cuantitativa depende del circuito a modelar (podrían ser miles de circuitos para un sistema dado).

Supongamos que para diseñar un circuito de N compuertas lógicas, se cambia el paradigma de modelado, y en vez de variables booleaneas se utiliza "transformación de registros". Esto permite reducir la complejidad cuantitativa, ya que en este caso, se manipulan enteros, operaciones aritméticas, etc. Pero por otro lado, esto implica un aumento en la complejidad cualitativa (en el caso de un registro, se pueden modelar cientos de compuertas con un solo operador por medio de un paradigma más complejo). Otra forma de reducir la complejidad cuantitativa es por medio de la descripción jerárquica, usando elementos de base que se unen formando modelos de mayor nivel.

2.2 FORMALISMO DEVS

Muchas de las abstracciones existentes para modelar sistemas de eventos discretos tratan de describir el fenómeno combinando diversas técnicas, haciendo muy difícil el desarrollo de las simulaciones. Para evitar estos problemas, Zeigler [Zei76] propuso un formalismo de especificación conocido como DEVS (Discrete EVents Systems Specification). Esta aproximación propone una teoría de modelado de sistemas de eventos discretos. Permite descripción **modular** de los fenómenos a modelar (aproximación modular) y ataca la complejidad usando una aproximación **jerárquica**. El formalismo es independiente de todo mecanismo de simulaclión.

El paradigma (que es de tiempo continuo y variables discretas) provee una forma de especificar un modelo describe como un conjunto consistente de una base de tiempo, entradas, salidas, variables de estado y funciones para calcular los siguientes estados y salidas. Además de un medio de construir modelos simulables, provee una representación formal para manipular matemáticamente sistemas de eventos discretos. Esta aproximación se integró posteriormente con nociones de programación orientada a objetos [Zei84, Zei90, Zei97].

Habiéndose liberado de sus orígenes de computación, este formalismo es más que un medio de construir modelos simulables, y provee una representación formal de DEDS que puede ser manipulada matemáticamente (así como las ecuaciones diferenciales lo hacen en sistemas continuos).

El formalismo DEVS puede verse como una forma de especificar sistemas cuyas entradas, estados y salidas son constantes de a trozos, y cuyas transiciones se identifican como eventos discretos. El formalismo define cómo generar nuevos valores para las variables y los momentos en los que estos valores deben cambiar. Los intervalos de tiempo entre ocurrencias son variables, lo que trae algunas ventajas: en los formalismos con una única granularidad es difícil describir los modelos donde hay muchos procesos operando en distintas escalas de tiempo, y la simulación no es eficiente ya que los estados deben actualizarse en el momento con menor incremento de tiempo, lo cual desperdicia tiempo cuando se aplica a los procesos mas lentos.

Un modelo DEVS se construye en base a un conjunto de modelos básicos, que se combinan para formar modelos acoplados. Los modelos pueden ser de comportamiento (**atómicos**), o de estructura (**acoplados**). Un modelo acoplado especifica cómo se conectan las entradas y salidas de los componentes, y los nuevos modelos también son modelos básicos modulares, que pueden usarse para integrar modelos de mayor nivel. El modelado jerárquico permite la creación

de una Base de Datos de modelos, permitiendo la reutilización de los modelos creados y chequeados. En esta forma mejora la seguridad de la simulación, reduciendo el tiempo de chequeo de las nuevas simulaciones, y aumentando la productividad.

2.2.1 Modelos atómicos

Un modelo atómico es una especificación de un modelo de comportamiento. Una especificación formal de estos modelos puede definirse como

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

$\mathbf{I} = < P^X, P^Y >$ representa la definición de la interfaz modular del modelo. En este caso, $\forall\ i \in \{X, Y\}$, P_j^i es una definición de un port (de entrada o salida respectivamente), donde $j \in N$, $j \in [1, \mu]$, $(\mu \in N, \mu < \infty)$, y

$$P_j^i = \{ (N_j^i, T_j^i) \ / N_j^i \in [i_1, i_\mu] \text{ (nombre del port), y } T_j^i = \text{Tipo del port} \};$$

X es el conjunto de eventos externos de entrada;

S es el conjunto de estados secuenciales;

Y es el conjunto de eventos externos generados para salida;

δ_{int}: **S**→**S** es la función de transición interna, que define los cambios de estado por causa de eventos internos;

δ_{ext}: **Q**xX→**S** es la función de transición externa, que define los cambios de estado por causa de eventos externos. Q es el conjunto de estados totales del sistema especificado como $Q = \{ (s,e) / s \in S, e \in [0, D(s)] \}$, donde **e** representa el tiempo transcurrido desde la última transición de estado con estado **s**;

λ: S → Y es la función de salida;

D: S→R^+ es la función de duración de un estado, donde D(s) es el tiempo que el modelo se queda en el estado s si no hay un evento externo.

Para especificar modelos DEVS es conveniente considerar que el modelo tiene una interfaz consistente de ports de entrada/salida que interactúan con el entorno (**I**). Cuando se reciben en los ports de entrada los eventos externos, la descripción del modelo debe determinar cómo responder. Por ende, un modelo atómico posee ports de entrada y salida por los cuales transita toda interacción con su entorno, lo cual facilita el encapsulamiento. Este concepto de interfaz con el entorno permitirá una construcción modular de modelos más complejos, facilitará la validación subconjunto por subconjunto, y la reutilización de las descripciones ya efectuadas. Luego, la descripción simulable de un modelo atómico podrá ser guardada en una biblioteca para hacer acoplamiento con otros modelos más complejos.

Un modelo atómico DEVS transita entre los estados (**S**) vía sus funciones de transición. Si no hay eventos externos, los estados pueden cambiar ejecutando la función de transición interna (δ_{int}), cuya activación está determinada por la función de pasaje de tiempo (**D**) aplicada al estado actual. Antes de cada transición interna, el modelo puede generar un evento de salida, ejecutando la función de salida (λ), que depende del estado previo a la transición. También puede haber un cambio de estado cuando hay un evento de entrada externa (δ_{ext}). La función de transición externa determina el nuevo estado basándose en el estado actual, el tiempo transcurrido en ese estado, y la entrada.

En la definición de su mecanismo de simulación, Zeigler consideró que todo modelo atómico siempre tiene definidas dos variables de estado: **fase** y σ. Si no hay eventos externos, el modelo se queda en la fase durante σ, que es el tiempo que resta hasta la próxima transición interna. Si hay eventos externos, la transición externa hace que el modelo cambie de fase y σ, preparándolo para la próxima transición interna. El estado en el que se entra luego de un evento externo depende de la entrada, el estado actual, y el tiempo transcurrido en este estado. Por ende, la función de avance de tiempo que controla el timing de las transiciones internas simplemente devuelve el valor de σ.

Ejemplo 6

En la Figura 4 puede verse la secuencia de eventos en un sistema modelado con DEVS: $s2=\delta_{int}(s1)$; $s3=\delta_{int}(s2)$, etc. Estas son transiciones internas, que ocurren cuando no hay eventos externos, y cuya ocurrencia está determinada por la función de duración D. Aquí, por ejemplo, D(s2) determina la hora de la próxima transición interna en base al estado s2. En la figura también se ven los eventos de salida generados: $y1=\lambda(s1)$, ..., etc. Para ejemplificar la ejecución de la función de transición externa, puede verse que cuando se recibe el evento de entrada externa x1, la transición externa resultante dependerá del estado actual (s4), el tiempo transcurrido (e4), y el evento de entrada (x1), para determinar un nuevo estado. Es decir que $s5 = \delta_{ext}(s4, e4, x1)$.

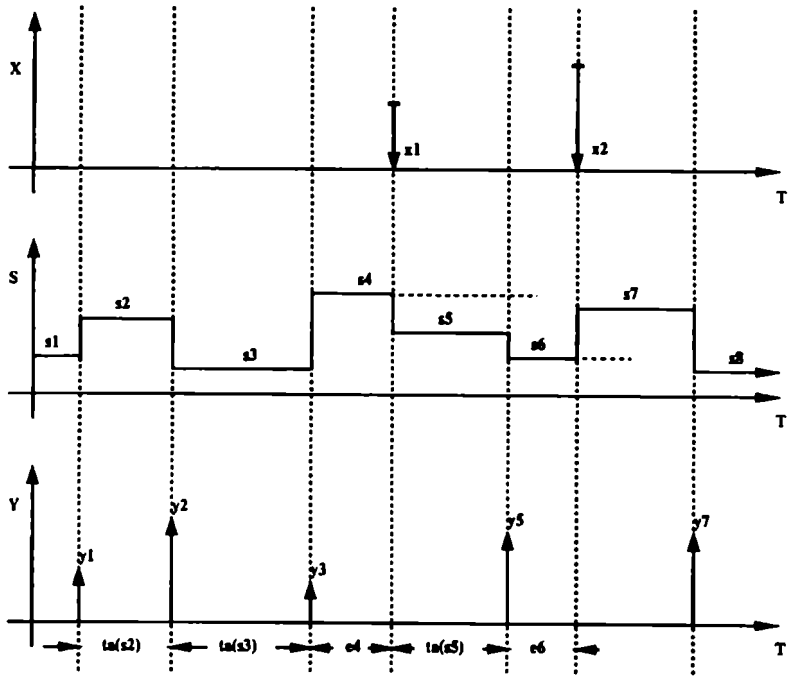
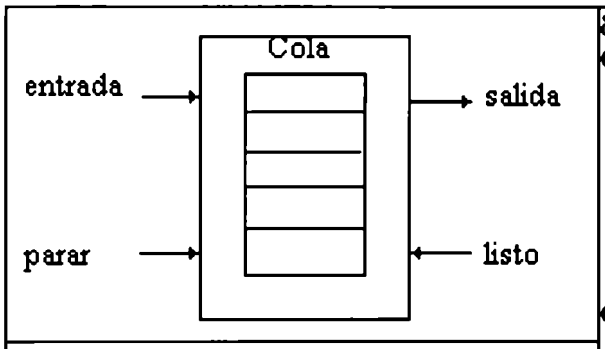


Figura 4 - Trayectorias en un modelo atómico DEVS [ChZ94].

Un estado **s** con **D(s)** = ∞ se dice pasivo (un estado no pasivo se dice activo), y la función de transición interna δ_{int} especifica el cambio de estado en el modelo, que ocurrirá luego del tiempo dado por D. Si hay un evento externo en el instante (**t+e**), se ejecuta la función de transición externa δ_{ext}, y el modelo cambia inmediatamente al estado $s'= \delta_{ext}(s,e,x)$. La variable **e** contiene el tiempo transcurrido en el estado actual, y se pone en cero luego del cambio de estado.

Ejemplo 7

Consideremos un modelo de buffering simple, como el ilustrado en la
Figura 5. Aquí pueden distinguirse tres ports de entrada: *entrada*, para recibir datos de entrada, *listo*, para recibir reconocimiento de un modelo ubicado en un nivel inferior, y *parar* para que un proceso de nivel superior pueda realizar control de flujo. El port *salida* envía ítems hacia los modelos de menor nivel. Se usan las variables **e** (tiempo transcurrido en la fase actual), y σ (tiempo restante en la fase actual), esenciales para lograr modularidad en modelos de eventos discretos (se asume que estas variables las maneja el simulador abstracto).




```
δint (s, e) {
   si fase = ENVIANDO
     /* Saca el primero de la cola luego
              del tiempo de preparación */
    Cola = resto(Cola)
    pasivar
}
```

```
λ (s) {
  si fase = ENVIANDO
            enviar primero(Cola) al port salida
}
```

```
δext (s, e, x) {
Cuando se recibe x en el port entrada
  insertar(x, cola)
  si hay_uno(cola) /* Debe transmitirse inmediatamente */
     e = tiempo_de_preparación /* Demora de preparación */
     mantener ENVIANDO durante e  /* Cambio de fase */
  sino
     continuar

Cuando se recibe listo en el port listo
  si !vacía(cola)          /* Tomar el siguiente elemento */
    e = tiempo_de_preparación
    mantener ENVIANDO durante e
  sino
    continuar                  /* Actualizar sigma */

Cuando se recibe x en el port parar con tiempo transcurrido e
   si fase = ENVIANDO y x = parar
       tiempo_restante =  sigma - e
       pasivar

   si fase = PASIVA y x = comenzar
      e = tiempo_restante
      mantener ENVIANDO por e
   sino
      continuar
}
```

Figura 5 - Un modelo de buffering simple

La especificación de transiciones externas tiene una definición para cada port. La primera dice que cuando se recibe un valor en el port *entrada*, se debe insertar en la cola. Al recibir un evento por el port *listo*, se presume que la tarea realizada por el modelo de nivel inferior ha terminado; por ende, si quedan más elementos en la cola, planifica la ejecución del próximo trabajo luego de un tiempo de preparación. Si es el único elemento en la misma, se debe transmitir inmediatamente, pasando a la fase ENVIANDO. Sino, el modelo debe avanzar (pasaje de tiempo). La frase *continuar* indica que el tiempo restante en la fase en la cual se encuentra el modelo no debe cambiar como resultado del procesamiento del evento externo. La sentencia *pasivar* (poner el modelo en PASIVO) representa una fase del modelo en la cual espera por eventos externos.

La transición interna tiene una sola fase, ENVIANDO, en la cual el modelo espera un *tiempo_de_preparación*, que provoca que se planifique una transición interna para hora_actual + tiempo_de_preparación. Esto sirve para modelar el tiempo entre que llega el mensaje y se puede retransmitirlo. Previamente se ejecuta la función de salida, que envía el primer valor en su cola al port *salida*, luego de lo cual la función de transición interna lo saca de la cola, y lo pone en estado pasivo.

La especificación del port de entrada *parar* dice que si se recibe la indicación de parar, el modelo abandona la fase ENVIANDO y aborte la transmisión actual. En ese instante se registra el tiempo restante de transmisión (es decir, el tiempo desde el comienzo de la fase actual hasta el próximo evento menos el tiempo transcurrido). Si en cambio, el evento en el port indica reinicio, la transmisión se reanuda, volviendo a tomar el tiempo hasta el fin de transmisión.

Ejemplo 8

La idea de este ejemplo es construir un modelo para comparar desempeño de sistemas mono/multiprocesador, basado en dos medidas: turnaround y throughput. Para ello se comienza por hacer un modelo atómico para estudiar el desempeño de arquitecturas simples. Para modelar un monoprocesador, se construye un modelo atómico llamado P, en el que solo se representa el tiempo que toma completar un trabajo.

variables de estado:
sigma = ∞, **fase** = Pasiva;
Id_Trabajo= < Nombre_Trabajo, Tpo_Procesamiento >
/* Indica el trabajo que se esta ejecutando actualmente */
Cola_de_Trabajos = Vacía /* Contiene los Id_Trabajo que están esperando tomar el procesador. */

Considerando la definición formal del modelo P, obtenemos:

X = { *Id_trabajo* };
Y = { *Id_trabajo* }
S = { { *Fase, sigma, Cola de trabajos, id_Trabajo* } }

δ_{ext} (Cola de trabajos, id_Trabajo, e, x) {
según fase
pasiva: sigma = Id_Trabajo.Tpo_Procesamiento;
fase = Ocupado;
id_Trabajo = Id_trabajo.Nombre_Trabajo;
ocupado: Insertar(Id_trabajo, *Cola de trabajos*);
sigma = sigma - e;
}

δ_{int} (Cola_de_Trabajos, id_Trabajo, e) {
según fase
ocupado: **si** (vacía(Cola_de_Trabajos))
fase = pasiva;
sigma = ∞;
sino
id_Trabajo = Get(Cola_Trabajo);
sigma = Id_Trabajo.Tpo_Procesamiento;
finsi
pasiva: /* Nunca Ocurre */
e = D(*sigma*) - *sigma;
}

λ (s) { enviar Id_Trabajo al port *salida* }

Figura 6. Un modelo atómico de desempeño de un procesador.

La llegada de trabajos se modela usando la función de transición externa. Si el procesador está libre, la fase se pone en PASIVA. Cuando llega un trabajo por el port *entrada*, se guarda su identificador y se simula que el procesador trabaja (mantener ocupado *tpo_procesamiento*, poniendo la **fase** en ACTIVA, y **sigma** en *tpo_procesamiento*). Si llega otro trabajo, lo ignora (usando la sentencia *continuar*, que actualiza sigma para reflejar el pasaje de tiempo). Cuando el procesador terminó de procesar, pone la identidad del trabajo en el port *salida*, y vuelve a la fase PASIVA. Este procesador se puede combinar con otros componentes para crear modelos de arquitecturas para analizar su desempeño. El modelo básico se puede refinar para representar aspectos mas complejos de la operación, como se mostrará en el capítulo 6.

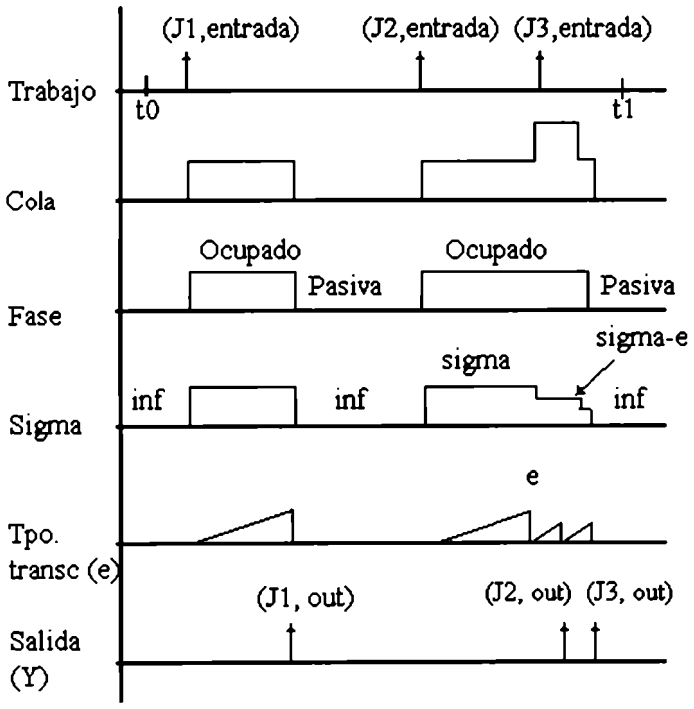


Figura 7. Ejecución del modelo del Ejemplo 8.

Es importante notar que no hay forma de generar una salida directamente desde un evento de entrada externa. Una salida solo puede ocurrir antes de una transición interna. Para que un evento externo provoque una salida sin demora, hay que "planificar" un estado interno con duración cero.

2.2.2 Modelos acoplados

Los modelos básicos en el formalismo DEVS, pueden integrarse para formar un modelo multicomponente o acoplados definido por la estructura:

$$CM = < I, X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, \text{select} >$$

$\mathbf{I} = < P^X, P^Y >$ representa la definición de la interfaz modular del modelo. En este caso, para $i \in \{X, Y\}$, P^i es la definición del conjunto de ports (de entrada y salida respectivamente), donde

$$P_j^i = \{ (N_j^i, T_j^i) \ / \ \forall j \in [1, \mu], (\mu \in N, \mu < \infty), N_j^i \in [i_1, i_\mu] \text{ (nombre del port), y } T_j^i = \text{Tipo del port} \};$$

X es el conjunto de eventos externos de entrada;

Y es el conjunto de eventos externos de salida;

$\mathbf{D} \in N$, $D < \infty$, es el conjunto de índices de modelos componentes; y $\forall i \in D$,

$\mathbf{M_i}$ es un modelo componente básico, definido como:

$$M_i = < I_i, X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, \lambda_i, D_i >$$

$\mathbf{I_i} \subseteq D$ es el conjunto de modelos influenciados por el modelo i; y para cada $\forall j \in I_i$,

$\mathbf{Z_{ij}}$: $Yi \rightarrow Xj$ es la función de traducción de salida i a j.

select: $D \rightarrow D$ es el selector ante eventos simultáneos (función de secuenciación o prioridad).

Un modelo multicomponente es equivalente a un modelo básico en el formalismo DEVS, y puede ser empleado en un modelo multicomponente mayor, ya que el formalismo es cerrado bajo acoplamiento. Un modelo DEVS que no se construye usando un modelo acoplado es un modelo atómico. Se llama **modelo básico** (o **de base**) a un modelo atómico (descripción jerárquica a dos niveles) o acoplado (descripción jerárquica a n niveles) que está almacenado en una biblioteca.

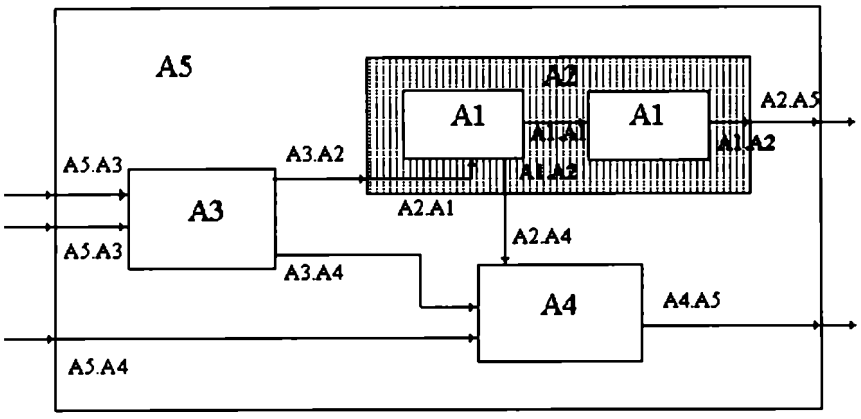


Figura 8 - Un modelo acoplado

Como vemos, un modelo acoplado define cómo conectar varios modelos básicos para formar un nuevo modelo. Este modelo puede ser empleado como un componente básico, permitiendo la construcción jerárquica de modelos complejos. El modelo acoplado contiene, por un lado, una interfaz modular (**I**), que le permite conectarse con otros modelos básicos. Luego, se definen los conjuntos de entrada y salida del componente (**X**, **Y**), como en otros modelos básicos. Por otro lado, se define un índice (**D**) para el conjunto de componentes ($\mathbf{M_i}$) que conforman el sistema acoplado.

Se determinan las influencias de cada componente ($\mathbf{I_i}$), y una especificación de acoplamiento ($\mathbf{Z_{ij}}$), en la cual la conexión de i a j se especifica designando a j como el influenciado por i. Z_{ij} provee una función de traducción de las salidas del modelo i en el conjunto de entradas del modelo j. Cuando ocurre un evento interno en i, en el mismo instante envía una señal al componente j. Finalmente, la función de selección, incluye las reglas empleadas para elegir cual de los componentes inminentes (los que tienen menor tiempo al próximo evento) ejecutará su próximo evento, si es que existen varios de ellos activos simultáneamente.

Ejemplo 9

Sea un modelo acoplado que consiste de un generador de trabajos para simular (GENR), y un transductor (TRANSDR), que se incluyen en la siguiente figura. El generador genera una secuencia de identificadores de trabajos espaciados en el tiempo. La función de transición interna vuelve al modelo a la fase ACTIVA luego de cada transición interna y la planifica para empezar una nueva transición en el momento dado por *tpo_entre_llegadas* (que en este caso es constante). Antes de que la transición interna ocurra, se produce la salida de un símbolo *id_trab*. Como GENR es un modelo DEVS, puede chequearse por separado. En principio, el generador es un modelo autónomo, por ende no necesita una función de transición externa. Sin embargo, se agrega un port de entrada *parar*, que cuando se estimula, frena la actividad del generador.

El transductor mide dos índices: el rendimiento (throughput) y el tiempo promedio de retorno (turnaround). Para calcular medidas de desempeño, el transductor ubica los *id_trab* que llegan en su port de entrada *llegada* en su *lista_llegados* junto con sus tiempos de llegada. Cuando un *id_trab* aparece en el port de entrada *resuelto*, se lo ubica en la *lista_resueltos* y también calcula las métricas. TRANSD mantiene su propio reloj local para medir los tiempos de llegada y de turnaround. El formalismo DEVS no permite que el reloj de la simulación este disponible para los componentes, por lo que los modelos deben mantener sus relojes acumulando información de tiempo transcurrido, usando **sigma** y **e**. Estos dos componentes básicos se acoplan para conformar el Marco Experimental EF, como se ve en la siguiente figura.

MODELO ATOMICO: GENR

variables de estado: sigma = ∞;
fase = activa
parámetros: tpo_entre_llegadas = 10

δ_{ext} (s, e) {
case port *entrada*
parar: pasivar
sino: error
}

δ_{int} (s, e, x) {
case fase
activa: mantener activa durante
tpo_entre_llegadas
pasiva: /* nunca ocurre */
}

λ (s) {
case fase
activa: **enviar** (nombre_al_azar, tiempo_al_azar)
al port *Salida*
pasiva: /* nunca ocurre */
}

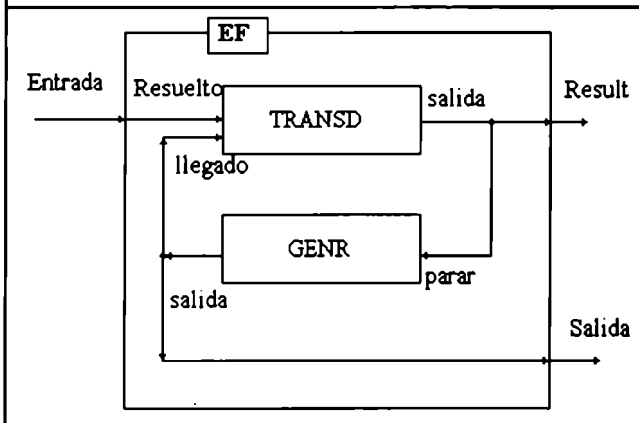
MODELO ATOMICO: TRANSD

variables de estado: sigma = intervalo de observación.
fase = activa; lista_llegada = lista_resueltos = ();
reloj = total_ta = 0;

δ_{ext} (s, e, x) {
Avanzar reloj hasta que coincida con el reloj global;
case port
llegada: **si** (NOT Inicializado)
Inicializado = Verdadero;
sigma = Intervalo_de_Observacion;
fase := Activa;
finsi
Agregar el trabajo a lista_llegada;
tiempo_de_llegada = reloj;
resuelto: buscar tiempo de llegada del trabajo;
total_ta = total_ta + reloj - tiempo de llegada
sacar el trabajo de la lista_llegada;
agregar el trabajo en la lista_resueltos;
}

δ_{int} (s, e) {
case fase
activa: pasiva /* fin del intervalo de observación. */
}

λ (s) {
case fase
activa: turnaround_promedio= total_ta/trab_res
throughput = trab_res / reloj;
enviar(turnaround_promedio al port salida);
enviar(throughput al port salida);
otro: no hay salida
}



MODELO ACOPLADO: EF

ARBOL DE COMPOSICION: raíz: EF
hojas: GENR, TRANSD.

ACOPLAMIENTO DE ENTRADA EXTERNA:
EF.Entrada → TRANSD.Resuelto

ACOPLAMIENTO DE SALIDA EXTERNA:
GENR.salida → EF.Salida
TRANSD.salida → EF.Result

ACOPLAMIENTO INTERNO:

GENR.salida → TRANSD.llegado
TRANSD.salida → GENR.parar

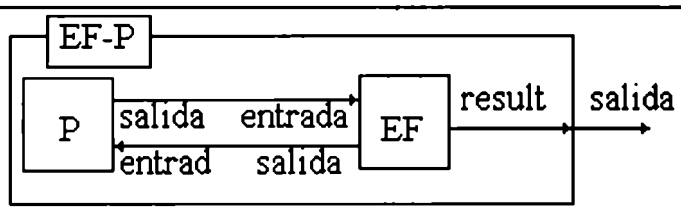
Figura 9 - Especificación del modelo acoplado EF y de sus componentes.

Como puede verse en el ejemplo, la especificación de la interconexión entre componentes tiene tres partes:

- Ports de entrada del modelo acoplado: especifican cómo identificar las entradas del modelo acoplado, estableciendo su conexión con los componentes;

- Ports de salida del modelo acoplado: especifican cómo identificar las salidas del modelo acoplado, permitiendo conectar componentes con la salida de cada modelo;
- Conexiones internas: especifica cómo se relacionan los ports de salida con los ports de entrada de los componentes internos.

El modelo acoplado del ejemplo se podría usar para experimentar con el modelo monoprocesador P presentado previamente. Para ello se lo debería acoplar directamente con el componente EF para formar el digrafo EF-P. En este caso, el modelo EF sirve como marco experimental, obteniéndose como resultado un modelo cerrado. Por este motivo, no hay acoplamiento de entrada externa en EF-P. El acoplamiento de salida externa hace que la salida del transductor esté disponible en el port *result* de EF-P. El acoplamiento interno es muy simple: los ports *salida* de cada uno de los componentes se acoplan a los ports *entrada* de los otros. De la misma manera, el marco experimental EF puede acoplarse a cualquier otro modelo, siempre y cuando éste use su port de entrada para recibir identificadores de trabajo, y produzca identificadores de trabajo en su port de salida. Usado de tal forma, el marco experimental generará trabajos para el modelo deseado, midiendo su throughput y turnaround.



MODELO ACOPLADO: EF-P

ARBOL DE COMPOSICION: raíz: EF-P
hojas: EF, P.

ACOPLAMIENTO DE SALIDA EXTERNA:
EF.result → EF-P.salida

ACOPLAMIENTO INTERNO:
P.salida → EF.entrada
EF.salida → P.entrada

LISTA DE PRIORIDAD: (P, EF)

Figura 10 - Especificación del modelo acoplado EF-P.

Para este caso, el modelo EF-P puede especificarse formalmente como:

$$EF\text{-}P = < I, X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

X = ∅; /* El modelo es cerrado */
Y = { throughput, turnaround };
D = { ef, p };

M_{ef} = EF;
M_p = P;

I_{EF} = P;
I_p = EF;

$Z_{ef,p}$ = { (EF.salida, P.entrada) };
$Z_{ef,EF\text{-}P}$ = { (EF.result, EF-P.salida) };
$Z_{p,ef}$ = { (P.salida, EF.entrada) };

y
select = (P, EF).

La función select contiene reglas para romper lazos con la planificación. Por ejemplo, si en EF-P el generador y el procesador tienen tiempos entre llegadas y de procesamiento iguales (o armónicos), tendrán igual tiempo de próximo evento cuando ambos estén listos para mandar un trabajo. En particular, si el generador de *tpo_entre_llegadas* y el tiempo de procesamiento son iguales, y el generador ejecuta su próximo evento primero, todos los segundos trabajos se pierde. Esto ocurre porque la salida del generador encuentra un procesador ocupado, y se ignora. Para evitarlo, la función select puede definirse de tal forma que el procesador se elija como componente inminente.

2.2.3 *Principio de Simulación abstracta*

La principal característica de modularidad de los modelos DEVS está garantizada por el hecho que la simulación es llevada a cabo por un simulador abstracto, que es genérico. La idea básica subyacente en el concepto del simulador abstracto, es que la simulación de los modelos atómicos y acoplados sea ejecutada por distintos **procesadores** especializados, llamados **simuladores** y **coordinadores** [Zei90].

La simulación de un modelo especificado es disparada por la recepción de mensajes de coordinación, que invocan secuencialmente a las salidas, transiciones externas, y transiciones internas planificadas por las funciones de tiempo. En esencia, un simulador abstracto es una descripción algorítmica de como llevar a cabo las instrucciones implícitas en los modelos DEVS para generar su comportamiento.

Para simular un modelo de estructura jerárquico, entonces, se utilizarán tres clases de procesadores (processors):

- **Simulador** (simulator): está encargado de controlar la ejecución de los modelos atómicos. Para ello, cada modelo atómico tendrá asociado un simulador, cuyo rol será coordinar la ejecución de distintas funciones del modelo.

- **Coordinador** (coordinator): es el encargado de tratar a los modelos acoplados. A cada modelo acoplado se le asociará un coordinador, con el rol de dirigir a los simuladores de los modelos que constituyen el acoplado.

- **Coordinador raíz** (Root coordinator): está encargado de controlar a los coordinadores de mayor nivel.

Cada procesador se comunica únicamente con otro del nivel inmediato superior (padre). Al tope de esta jerarquía se encuentra el coordinador raíz, cuya función principal es coordinar toda la simulación. Este procesador está encadenado con el coordinador del modelo acoplado exterior.

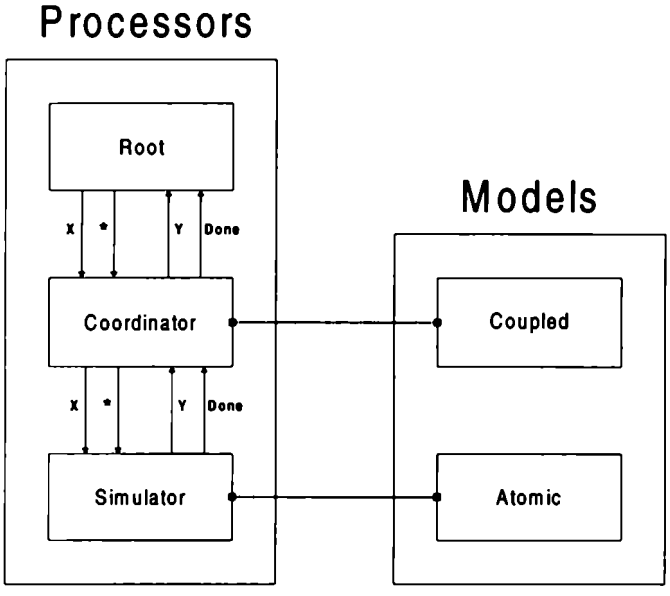


Figura 11. Relación entre Modelos y Procesadores.

La simulación avanza gracias a la interacción de mensajes que se intercambian entre los procesadores. Los mensajes pueden acarrear información con respecto a eventos internos y externos, así como necesidades de sincronización. Los mensajes tienen campos para el nombre del emisor, la hora (que puede ser un timestamp local o global, dependiendo del uso), y un contenido consistente del port destino y el valor asociado, ambos determinados por funciones de salida del modelo atómico. Hay 4 tipos de mensajes: *****, **X**, **Y**, y **listo**.

Un mensaje **X** representa la llegada de un evento externo. Incluye la hora global, y viene del padre. Hay dos acciones posibles ante un mensaje **X**:

- Un coordinador lo transmite a un procesador hijo, usando para ello un método de traducción (función Z_{ij});

- Un simulador llama a la función de transición externa del componente considerada, y responde con un mensaje **listo** indicando que la transición de estado ha terminado, y la hora prevista del próximo evento interno.

Un mensaje * llega a un procesador indicando que el próximo evento interno a tratar le concierne a su dominio.

- Si se trata de un coordinador, transmite el mismo mensaje al hijo que tenga el menor tiempo de próximo evento, llamado hijo **inminente**. La llegada de este mensaje implica que el próximo evento interno será llevado a cabo en su entorno.

- El simulador ejecuta la función de transición interna del modelo atómico asociado, provocando un cambio de estado. El simulador responderá con un mensaje **Y** (acarreando el nuevo estado del modelo, que debe ser transmitido al coordinador de nivel superior), y un mensaje **listo** (que indica que la transición fue llevada a cabo; y acarrea la hora del próximo evento).

Un mensaje **Y** acarrea los valores resultantes de la función de transición externa. Cuando un coordinador recibe un mensaje **Y** de su hijo inminente, consulta el esquema de acoplamiento de salida externa para ver si debe ser transmitido a su padre, y su esquema de acoplamiento interno para obtener los hijos y sus respectivos ports de entrada a los cuales hay que mandar el mensaje.

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

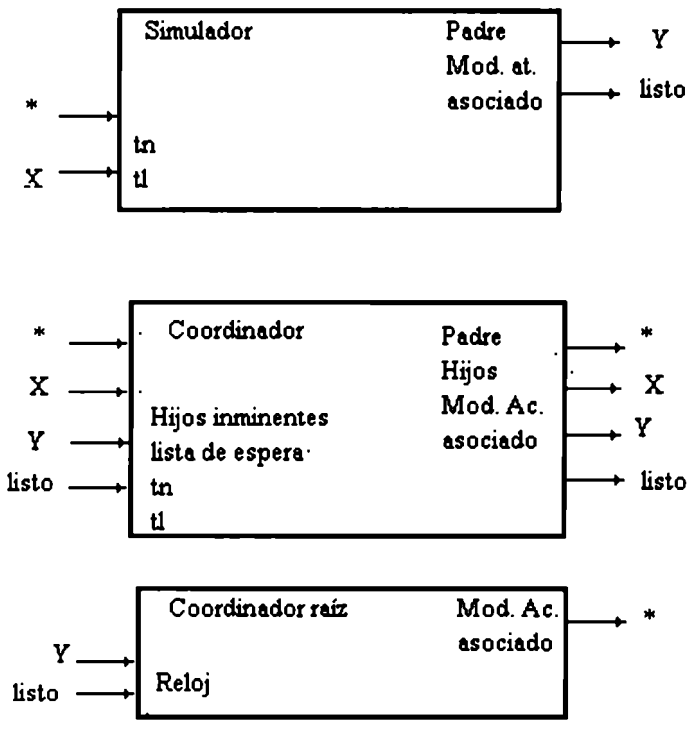


Figura 12 - Procesadores, variables usadas y sus mensajes.

Para llevar a cabo sus funciones, un simulador mantiene varias variables: la hora actual (t); la hora del último evento (tl); la hora del próximo evento interno (tn=tl+D(s)); el tiempo transcurrido desde el último evento (e = t-tl); y el tiempo restante hasta el próximo evento interno (σ = tn- t = D(s) - e). También mantiene enlaces al modelo atómico asociado, y a su coordinador padre.

Al llegar un mensaje *, se chequea si su hora coincide con tn (debería ser así porque un mensaje * sólo llega a un simulador si su modelo atómico asociado es el inminente), y si esto ocurre, aplica la función de salida, cuyo resultado se inserta en un mensaje **Y** que se envía al coordinador de nivel superior. Luego, el simulador activa la función de transición interna para actualizar el estado s, y actualiza las horas de eventos: tn=tl+D(s); tl = tn. Finalmente crea un mensaje **listo**, indicando que la transición de estado ha sido llevada a cabo, y que informa el nuevo tn al nivel superior.

La llegada de un mensaje **X** provoca, ante todo, que se chequee si la hora que lleva el mensaje **X** $\in$ [tl, tn] (esto debería ser así porque un mensaje **X** sólo debe llegar luego del ultimo evento procesado, y antes del próximo evento interno). Si es así, se actualiza el tiempo transcurrido (e = t-tl) y se aplica la función de transición externa con los valores actualizados de e y **X** ($s = \delta_{ext}(s,e,x)$). Además se actualizan las horas locales de eventos (tn=tl+D(s); tl = tn). Finalmente se crea un mensaje **listo**, que indica que la transición de estado ha sido llevada a cabo y que reporta el nuevo tn al nivel superior.

La simulación comienza inicializando los estados de los modelos atómicos, determinando sus tn. Estos tiempos se propagan hacia arriba por medio de mensajes **listo**, y establecen un camino de subcomponentes inminentes desde el modelo acoplado superior hasta un modelo atómico. Cuando el coordinador raíz recibe un mensaje **listo** de su hijo (el coordinador del modelo acoplado **superior**), le devuelve un * marcando su tn. Esto comienza la simulación, ya que el * será transmitido hacia abajo al simulador inminente. Esto resultara en una onda de mensajes **Y** hacia arriba, una ola de mensajes **X** hacia abajo, y una onda de mensajes **listo** hacia arriba, el ultimo de los cuales transmitido al coordinador raíz inicia la siguiente ronda de simulación (procesamiento del próximo evento interno).

El siguiente evento en el sistema ocurrirá en el menor de todos los tiempos planificados, el momento $t+\sigma$ (donde σ es el mínimo de los tiempos remanentes, $D_i(s_i)-e_i$, entre los componentes i), eligiendo uno usando la función select si fuera necesario. Llamemos i* a este componente elegido (el inminente). En el instante $t+\sigma$, justo antes que i* cambie de estado, se calcula la salida haciendo $y^*=\lambda i^*(si^*)$. Esta salida se manda a cada una de las influencias de i* en forma de una entrada traducida: para el influenciado j, la entrada x_{i^*j} es $Z_{i^*j}(y^*)$. El tiempo transcurrido en cualquier componente i en el tiempo $t+\sigma$ es $e_i'=e_i+\sigma$.

Un influenciado j responde al evento externo generado por i* aplicando su función de transición externa, para obtener el siguiente estado $s_j' = \delta_{ext}(s_j, e_j', x_{i^*j})$. Además pone su tiempo transcurrido en 0. Los otros componentes que no están en el conjunto de influencia no son afectados por la activación de i* excepto porque su reloj de tiempo transcurrido se incrementará en σ. Finalmente, el componente i* ejecuta su transición interna, pasando al estado $s_{i^*}'=\delta_{int}(s_{i^*})$, y poniendo su tiempo transcurrido en 0.

Ejemplo 10

Analicemos la interacción de los procesadores del simulador abstracto para el modelo EF-P (los círculos representan los procesadores y los rectángulos los modelos asociados).

El ciclo de simulación comienza con un mensaje * que es enviado por el coordinador raíz al coordinador del modelo exterior. El coordinador C:EF-P recibirá el mensaje *, y lo transmitirá a su hijo inminente. Para C:EF-P, el hijo inminente puede ser EF (cuando sea hora de generar un nuevo trabajo), o P (cuando un trabajo haya sido procesado).

Si se elige, C:EF, el coordinador le transmite el mensaje *, ya que es su hijo inminente. Para el coordinador C:EF, GENR será siempre el inminente hasta que sea el momento de parar la simulación (determinado por TRANSD, que será entonces el inminente).

Por ende, el mensaje se transmite hacia S:GENR, que recibe el mensaje *. Como resultado, arma un mensaje **Y** cuyo contenido tiene como port *salida*, y con un valor *id_trab*. Este mensaje es enviado a su padre C:EF. S:GENR también provoca la ejecución de la función de transición interna del modelo GENR, y una vez ejecutada manda un mensaje **listo** con el tn para GENR, con el fin de determinar la hora del próximo evento.

Cuando C:EF recibe el mensaje **Y** recién generado, lo traduce a un mensaje **X**, que se transmite al port *llegada* de su influenciado, TRANSD. El mensaje también se traduce a un mensaje **Y** en el port *salida* de su padre EF.

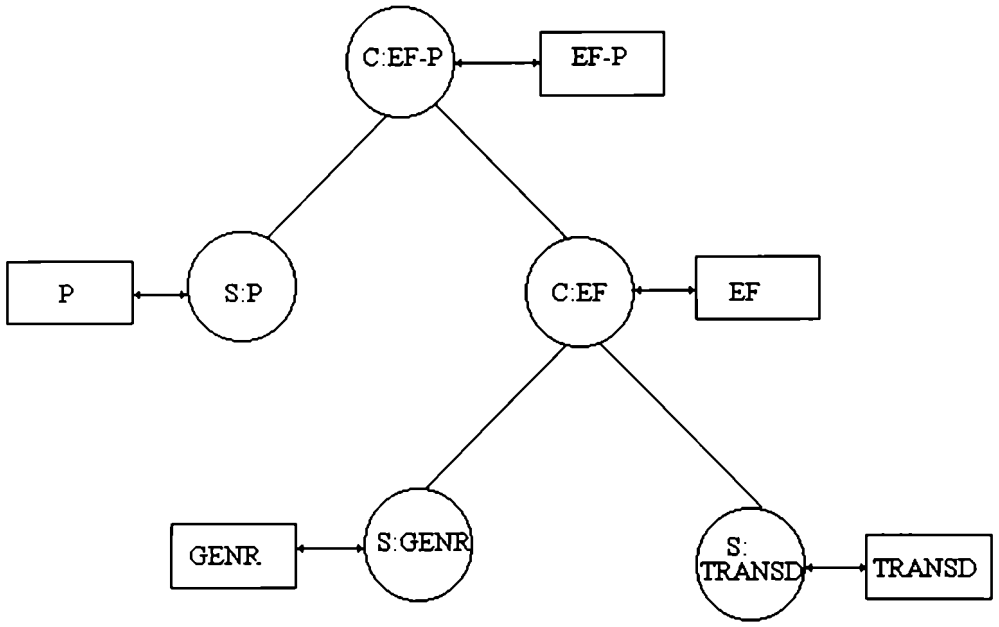


Figura 13. Simulador abstracto para el modelo EF-P.

Como EF es el hijo inminente de EF-P, el mensaje **Y** es enviado a C:EF-P, donde será enviado como un mensaje **X** al simulador de P, el influenciado de EF. En este caso, si P esta pasivo, se pone activo. Cuando el tiempo de procesamiento haya terminado, S:P generará un mensaje y (un *id_trab* en el port *salida*) que será transmitido a su padre C:EF-P, y posteriormente como un mensaje **X** a C:EF, el influenciado de P.

Para C:EF, el único receptor de un mensaje **X** en el port *entrada* es TRANSD, por ende, al recibir el mensaje X, lo traduce transformando la entrada por el port *entrada* a una nueva entrada por el port *resuelto*. Luego, el mensaje que manda a S:TRANSD es el mismo que recibe, excepto que el port se cambia a *resuelto*. Luego de retransmitir el mensaje **X** a todos los receptores, los pone en su lista de espera.

Veamos qué pasa con un mensaje **listo**. Por ejemplo, si los hijos de C:EF son {(GENR 10),(TRANSD 1000)}, GENR es el inminente, y el tn de C:EF es 10. Luego, C:EF envía un mensaje **listo** con tiempo 10 a su padre, C:EF-P. Este, habiendo recibido todos sus **listo**, envía un **listo** al coordinador raíz. Asumiendo que sus hijos sean {(EF 10), (P 15)}, el **listo** de C:EF-P contiene tn=10. El **listo** se manda al raíz y vuelve apareciendo como un * a C:EF-P con hora 10. Esto inicia un nuevo ciclo de simulación, en este caso con GENR generando un segundo trabajo.

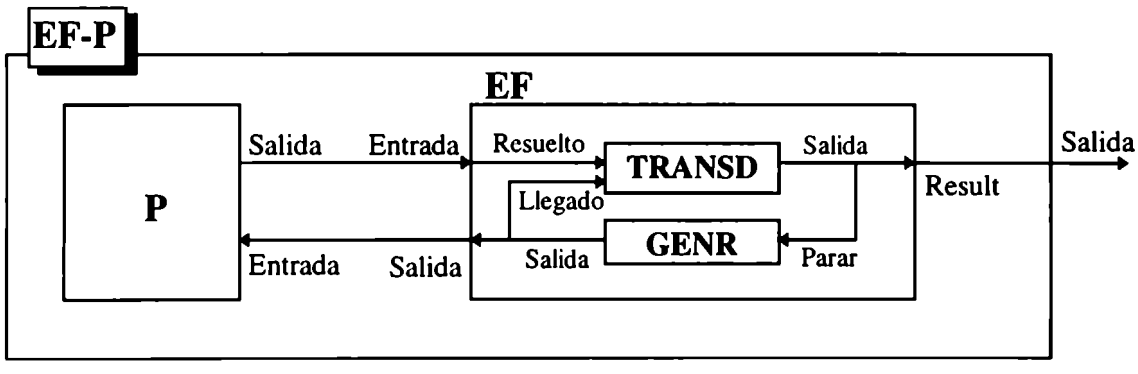


Figura 14. Detalle del modelo acoplado EF-P.

Este mecanismo de simulación abstracta es genérico, lo que implica que sirve para trabajar con cualquier modelo. La interfaz entre un modelo y su simulador está definida por métodos que representan consultas y operaciones que el simulador le hace al modelo, y los detalles de la estructura interna del modelo se ocultan en esta interfaz. Cualquier modelo puede simularse por un simulador si tiene definidos los métodos de la interfaz.

2.3 AUTÓMATAS CELULARES

Los autómatas celulares son modelos basados en un formalismo de variables y tiempo discretos, utilizados para modelar sistemas dinámicos complejos, creados originalmente por John von Neumann y Stephan Ulam [Tof94]. Conceptualmente, un autómata celular se forma poniendo copias de una misma "máquina secuencial" (el mismo aparato de cálculo) en puntos de una red n-dimensional. Todas las máquinas ejecutan las mismas operaciones en paralelo y simultáneamente en pasos de tiempo discreto: es un **sistema discreto** y posee invariancia espacial y en tiempo.

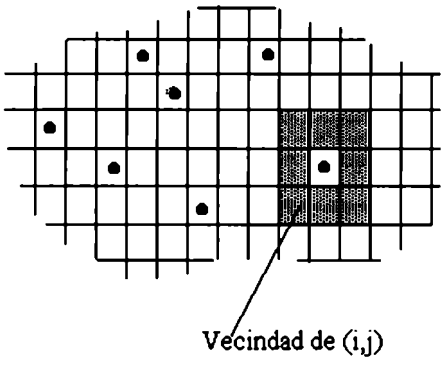


Figura 15. Un fragmento de autómata celular bidimensional.

Cada punto de la grilla espacial infinita se denomina **celda**; están ubicadas geométricamente y pueden tener un conjunto finito de valores para el estado, es decir, cada celda tiene un estado elegido de un alfabeto finito. Las celdas se conectan entre sí de forma uniforme, y su estado se actualiza simultáneamente y de forma independiente de los de las demás en pasos de tiempo discreto. El mecanismo de actualización precisa definir la **vecindad** de una celda, que es un conjunto de celdas cercanas (y en general es homogénea para todas las celdas)

Luego, un autómata celular se caracteriza especificando tres parámetros: un conjunto de estados **S**, una vecindad **N**, y una función de transición **T**. En cada punto de la grilla hay una celda con estado en **S**, los vecinos de una celda se determinan con la vecindad **N**, y el estado global en un instante **t** cambiará de estado en el instante **t+1** aplicando para ello, la función de transición **T** a los estados (simultáneamente en todas las celdas). Los vecinos de una celda son los que influencian su próximo estado (sus influencias), es decir que el estado de una celda en un momento dado sólo depende de su propio estado en el instante de tiempo previo, y los estados de sus vecinos en ese mismo momento.

La interacción entre los componentes está dada, para cada celda, por sus vecinos y la función local que especifica cómo influencian los vecinos a la celda . Esto determina que los vecinos son **influencias** y la celda es **influenciada**. El espacio de celdas es compacto y la función de transición es continua. Esto hace a los CA un punto ideal entre la dinámica continua y la teoría de complejidad, ya que están definidos de forma discreta pero exhiben dinámicas continuas.

Para comprender el sistema modelado por el autómata celular, se elige un estado inicial, y se observa la secuencia resultante de estados. En muchos casos, el sistema entra en equilibrio, independientemente de las características de los valores iniciales. En general, cambios en las condiciones iniciales pueden provocar cambios en los resultados.

Aunque los autómatas celulares representan un modelo de resolución elegante para un gran número de problemas complejos, la investigación de las propiedades básicas de la evolución de tiempo y estado de los autómatas celulares es de desarrollo reciente. Por ejemplo, en [Wol84] pudo mostrarse que la mayoría de los autómatas celulares pueden clasificarse en cuatro clases:

1. La evolución de todos los estados iniciales conduce, luego de un tiempo finito, a un estado homogéneo, donde todos las celdas tienen el mismo valor.
2. La evolución conduce a un conjunto de estructuras separadas estables o periódicas.
3. La evolución conduce a un patrón caótico.
4. La evolución conduce a estructuras complejas localizadas, generalmente metaestables.

El formalismo fue modificado por una variedad de autores, permitiendo incluir distintas propiedades. Entre ellas se destacan:

- **Autonomía**: algunas extensiones permiten que un autómata tenga entradas, especificando para ello, los **vecinos externos** de cada celda.

- **Homogeneidad**: se puede permitir que cada celda pueda tener distintas reglas, siendo en estos casos un autómata celular inhomogéneo. El modelo permite distintas reglas y conexiones (con aplicación a la biología teórica).

- **Uniformidad:** otras extensiones permiten que los vecinos no sean las celdas más cercanas, para lo cual se deben especificar los **vecinos internos** de cada celda individualmente. También, para cada celda se deben especificar sus variables de estado y la función de transición local.

- **Computabilidad**: para que pueda ser simulado, el modelo debe acotarse a un numero finito de celdas en cada paso. La forma mas simple de hacerlo es limitando el modelo a un área finita, lo que hace que se pierda homogeneidad, ya que debe determinarse qué se hace con los bordes. Para esto suelen usarse dos aproximaciones: o los estados de los bordes se especifican desde afuera, o se conectan los extremos entre sí, implementando un autómata toroidal.

- **Determinismo:** un autómata celular asincrónico estocástico puede obtenerse definiendo un experimento aleatorio e incluyendo variables aleatorias en el espacio de muestra del experimento. Para ello, la función de transición local debe ser aumentada con una implementación de la variable aleatoria.

- **Sincronismo:** se han definido autómatas celulares asincrónicos, cuyos conceptos generales se introducirán en la siguiente sección.

2.3.1 Autómatas celulares asincrónicos.

Como fuera mencionado, se asume que en el espacio de celdas, las celdas están activas simultáneamente (es un sistema paralelo), pero hay una restricción importante en la actividad simultánea, ya que se asume que todas cambian de estado simultáneamente.

Por otro lado, el gran número de celdas hace que en general se estén haciendo cálculos innecesarios. Al depender de los resultados de funciones de transición locales, puede ocurrir que se estén calculando celdas que no cambiarán su estado. Estas celdas, a las que llamaremos **latentes**, son simples de detectar, ya que una celda no puede cambiar de estado en su transición si ninguno de sus vecinos lo ha hecho. Llamemos un evento en el autómata a un cambio de estado en una celda. Por ende, se puede decir que en un autómata celular los eventos pueden ocurrir en un numero reducido de celdas (aunque es posible que ocurra en todas simultáneamente), y que puede detectarse si ocurrirá un evento en una celda mirando si hubo eventos en sus vecinos.

Esta solución permite definir autómatas celulares asincrónicos, cuya principal ventaja es su evolución en tiempo continuo. En lugar de usar una base de tiempo discreta, se consideran eventos instantáneos que pueden ocurrir asincrónicamente en instantes no predecibles. El uso de una base de tiempo continua permite lograr mayor precisión y saltear períodos de inactividad en la simulación, mejorando el uso de los recursos de la computadora.

Para lograr simular estos modelos, se debe mantener una lista de últimos eventos, que contiene las celdas cuyos eventos ocurrieron recientemente, junto con la hora de ocurrencia de los mismos. En base a la misma, se puede determinar en que celdas ocurrirán los siguientes eventos. Luego de simular cada una de las transiciones, se chequea si hubo un cambio de estado, y si es así la celda se agrega en la lista de últimos eventos. Esta aproximación facilita, además, la provisión de soluciones de simulación de autómatas celulares en paralelo, ya que cada la sincronización de cada celda es independiente de las demás.

En este caso, para iniciar la simulación será necesario detectar los estados latentes iniciales, agregando a las celdas no latentes en la lista de próximos eventos. El uso de esta técnica permite, además, simular un modelo con un numero infinito de celdas, siempre y cuando el número de celdas no latentes sea finito.

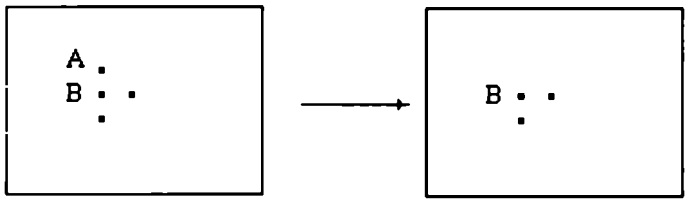
Ejemplo 11

Supongamos que se quiere simular el movimiento de un solo cuerpo en un plano, que avanza, siempre hacia la derecha. En este caso, cada celda del autómata tiene dos estados posibles (presencia o ausencia). S={0,1}. Una celda pasiva nunca se pondrá activa por si misma; esta representada por (0,∞). Como el cuerpo se mueve hacia la derecha, N = {(0,0), (1,0)}, que sumados a (i,j) dan las dos coordenadas de la celda (i,j) y su vecino derecho.

La función de transición es muy simple:
. Poner estado secuencial en 0. Pasivar.
. Prender el estado del vecino (1), y planificar una transición en TIEMPO_DE_MOVIMIENTO.

A pesar de estas ventajas, esta aproximación precisa sincronización explícita de las celdas a activar, y en algunos casos, el tiempo de procesamiento extra gastado en administración de la lista puede no tener buenos resultados. En algunos casos, este overhead puede superar las mejoras de performance de las aproximaciones asincrónicas. Por ejemplo, en un modelo donde la mayoría de las celdas son no latentes, no es bueno usar esta estrategia. Esto motiva que los esquemas de actualización sincrónica/asincrónica puedan conducir a resultados muy diferentes, y un modelador debe ser cuidadoso al utilizar cualquiera de ellos.

Por otro lado, es posible realizar una actualización asincrónica pura del autómata, ya que se pierden las propiedades básicas del mismo. Supongamos, por ejemplo, un modelo simple del juego de la Vida [Gar70]. Supongamos la situación presentada en la figura anterior. Supongamos que aquí se selecciona la celda A, se ejecuta su función de transición, y cambia su estado como puede verse en la figura. Cuando se ejecute la transición de la celda B, su estado pasará a ser 0, aunque su estado correcto debería haber sido una celda viva. Por otro lado, si primero se hubiera ejecutado la celda B, se habría obtenido el resultado correcto.



Lista de eventos:
<*, t, A>
<*, t, B>

Lista de eventos:
<*, t, B>

Figura 16. Ejecución de una transición en el juego de la Vida.

Un autómata celular asincrónico que ejecute de esta forma será, por ende, no determinístico. Para evitar estos problemas, todas las celdas activas que figuren en la lista de eventos deben ejecutar sobre el mismo espacio de celdas de base, antes de hacerle cualquier modificación.

Ejemplo 12

Consideremos ahora una extensión del modelo anterior para estudiar congestión de tráfico. El modelo previo no especifica qué pasa cuando hay colisiones. Supongamos que, en lugar de un individuo en un plano, se tienen automóviles en una dirección, y cuando uno obstruye el camino, quien quiere avanzar pasa de carril (preferentemente hacia la izquierda). El conjunto de estados S={0,1} representa la ausencia/presencia de un automóvil. El conjunto N está formado ahora por todas las celdas arriba y abajo del centro, para que el automotor se mueva en esas direcciones.

Si N esta ordenado en el orden legal de sobrepaso, la función de transición interna debe hacer:
. Encontrar la primera celda no ocupada
. Prender esta celda, y planificar un evento en TIEMPO_DE_MOVIMIENTO.
. Poner el estado propio en 0, pasivar, dejar las otras celdas no afectadas (dejar su estado sin cambiar, y restarles su tiempo_restante(σ^*), para actualizar los valores de tiempo).
. Si no hay vecinos, replanificar en después de TIEMPO_DE_REINTENTO. Dejar las otras celdas igual.

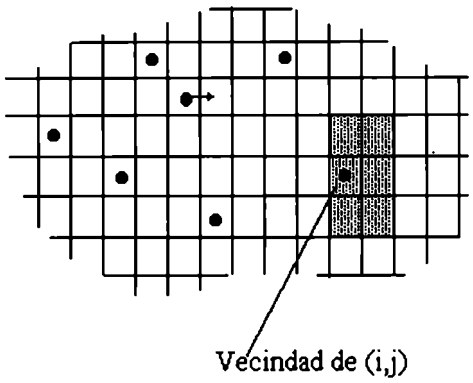


Figura 17. Vecindad del modelo de tráfico en una dirección.

Aquí la función select determina cuál de dos automotores que quieren ocupar la misma celda al mismo tiempo será la que lo hace. Un estado inicial en este modelo pone un conjunto finito de celdas en estado 1, con tiempo time_left σ entre 0 y TIEMPO_DE_MOVIMIENTO; todas las demás están pasivas. El patrón se moverá hacia la derecha, con los automotores haciendo maniobras para tapar obstrucciones. Notar que para hacer la simulación mas realista, TIEMPO_DE_MOVIMIENTO y TIEMPO_DE_REINTENTO pueden hacerse variables aleatorias, de forma tal que los automotores se muevan al azar en vez de a velocidad constante.

2.3.2 Ejemplos de aplicación

Algunos ejemplos clásicos del uso de autómatas celulares incluyen el modelado de dinámica en fluídos y gases, colonias de animales, modelos ecológicos, y modelos del tráfico en una ciudad, entre otros (para mayor información sobre aplicación de autómatas celulares para simulación de sistemas físicos complejos, consultar [Gut95], [Tof94], [Wol84], [Wol86]). En esta sección se mencionarán algunos ejemplos en los que se ha utilizado este formalismo. El objetivo es mostrar el mecanismo de modelización, y no analizar los resultados obtenidos, que dependerán de cada aplicación en particular. Para más detalles sobre cada uno de los ejemplos, puede consultarse [BLMW97], [DW98a], [DW98b].

El *Juego de la Vida* es un modelo simple muy difundido, que tiene el objetivo de analizar el comportamiento de un sistema usando autómatas celulares. Fue creado por John Horton Conway [Gar70], y captó la atención mundial de los científicos de las computadoras. Este modelo implica la interacción de submodelos basados en reglas simples de nacimiento y muerte. En cada momento una celda puede estar viva o muerta, y su estado cambiará de acuerdo a un conjunto de reglas que deben definir si una celda pasará de vacía a llena (nacimiento) o de llena a vacía (muerte). La vecindad está formada por las 9 celdas adyacentes (incluyendo a la celda analizada dentro de la vecindad). Si una celda está muerta, "nace" cuando tiene exactamente tres vecinos vivos. Si una celda está viva, se mantiene viva si tiene dos o tres vecinos vivos. Si tiene menos de dos o más de tres vecinos vivos, la celda muere. Estos modelos son impredecibles, y producen una increíble variedad de actividad. En algunos casos, las celdas vivas crecen demasiado rápido sin límite; en otros, tienden a desvanecerse.

El modelo *Vants* [Gut95] analiza la acción de un conjunto de hormigas sobre una sección de follaje. Un conjunto de hormigas avanzan sobre el campo apuntando todas hacia cierta dirección. Si una hormiga está en una sección de terreno con follaje, se alimenta (la celda pasa a un estado de semi-desfoliación), rota 90 grados y sigue avanzando. Si pasa por una celda semi-desfoliada, termina de destruirla, rota 270 grados, y sigue avanzando. Las celdas semi-desfoliadas vuelven a crecer cada determinado tiempo, mientras que las desfoliadas por completo nunca vuelven a crecer.

En [EDPWJ89] también se modelaron algunos aspectos importantes del comportamiento de una colonia de hormigas *pagonomyrmex barbatus*. Aquí, las hormigas pueden realizar tres acciones: buscar comida (persiguiendo su olor hasta llegar a la fuente), comer, o realizar *trophallaxis* (es decir, excitar a otras en el hormiguero para buscar comida). En este caso se emplearon celdas hexagonales. Las porciones de comida tienen el mismo aroma, por lo que se suman al llegar aromas de distintas fuentes a la misma celda y se dispersan a igual velocidad a igual cantidad de celdas. Pero cada una puede tener una cantidad de comida distinta. El aroma a comida se difunde en hexágonos vecinos en función del tiempo; en una celda se determina usando una función lineal de la distancia y el aroma de la celda que contiene la comida, y se define un valor crítico por debajo del cual el aroma a comida no es percibido por las hormigas. Una hormiga se mueve en alguna de las 6 direcciones: si una hormiga percibe olor a comida, se moverá hacia donde el aroma sea más fuerte

(dando preferencia al sentido de las agujas del reloj). Si no hay aroma en los vecinos, la hormiga se queda en la misma posición, se mueve en la misma dirección que el movimiento anterior, o se mueve a alguna de las dos posiciones hacia adelante y en diagonal respecto de la dirección de su última movida (todas con igual probabilidad). Si una hormiga regresa al hormiguero con comida, provoca que otras hormigas vayan a buscar comida (trophallaxis), y se modifica el número que indica la cantidad de hormigas que fueron en busca de comida. Si una hormiga regresa al hormiguero sin comida, sólo se modifica el número que indica la cantidad de hormigas que fueron en busca de comida.

En [GLNW94] se modeló una *red de telefonía celular* cubriendo un área geográfica donde se prestan los servicios. El área se divide en celdas, cada una con una estación que recibe la solicitudes, y que decide si acepta el pedido de línea. En tal caso, dice qué canal asignarle para la comunicación. Para esto la estación base se comunica con otras estaciones base, usando una red cableada independiente. Cuando se acepta una llamada, se le asigna un canal de comunicación con su estación, que maneja la conexión al destinatario. Cuando un llamado es rechazado, se dice que el mismo es bloqueado (o sea, se le niega un canal). Aquí, la asignación dinámica de los canales se modeló como un autómata formado por hexágonos que poseen una estación, y cuya vecindad se define como todos los hexágonos que están a una distancia discreta **r**. Un canal en uso en una celda dada no puede ser usada por ninguno de sus vecinos, definiendo a **r** como un parámetro fijo relacionado con el poder y atenuación de la señal. Para modelar la interferencia también se podría imponer la restricción de que no se puedan usar concurrentemente canales adyacentes dentro de la vecindad. Para aceptar una llamada, sólo se toma como base la cantidad de llamadas en progreso en cada celda. Sean **M** el número total de canales, **I** el número de celdas, y **estado(i)** el número de llamadas en progreso en la celda **I**, y $\sigma_i(\mathbf{k})$ el canal adquirido cuando el estado era **k**. Se usa la siguiente regla de transición local:

si estado(i)≤M .AND. cuando llega una llamada a la celda i el índice del canal n es igual a σ_i(estado(i)+1) **entonces**
  si estado(i) = M .OR. estado(i) < M .AND. el canal n=σ_i(estado(i)+1), está en uso en alguna celda vecina
    entonces Bloquear la llamada
  si el canal n no está siendo utilizado en ninguna de las celdas vecinas **entonces**
    La llamada es aceptada

El modelo *WaTor* [OS92, OSH93] consiste de un conjunto simple de reglas que describen el comportamiento de tiburones y peces en una porción del océano de forma rectangular. Los peces y tiburones pueden realizar alguna de las siguientes actividades: desplazarse dentro del océano, reproducirse, o dormir por algún tiempo. Los tiburones además, se alimentan de los peces, y si durante un cierto tiempo no comen, se mueren por inanición. Tanto un pez como un tiburón pueden moverse a una celda adyacente de su posición actual con dirección norte, este, oeste o sur. Existe un parámetro llamado *tiempo de activar* que indica una demora luego de haber ejecutado una función de transición. Para moverse, cada pez selecciona una posición al azar entre las celdas adyacentes vacías. Si no hay posiciones vacías entonces el pez no se mueve. Luego que pasó el tiempo correspondiente a *tiempo de activar,* la edad del pez se incrementa en esa cantidad de tiempo, y se reactiva. Si un pez alcanza la edad de reproducción, en el momento que decide moverse se coloca un nuevo pez con edad 0 en su antigua posición. Las reglas de movimiento de los tiburones son similares a las de los peces, excepto que la caza de peces tiene prioridad sobre el mero movimiento: primero buscan peces en sus posiciones vecinas, y si hay uno, lo comen (reinicializando un parámetro de inanición). Los tiburones se reproducen al igual que los peces, y finalmente, si un tiburón nada durante una cantidad de unidades de tiempo sin comer, se muere.

En [MZBG96] se modeló el proceso de desagüe de una cuenca fluvial formada por varias capas verticales (aire, superficie de agua, superficie de tierra, agua subterránea, capa de rocas). El agua de la cuenca circula hacia la desembocadura y parte de ella se infiltra hacia las capas subterráneas. Todo el terreno que abarca la cuenca se divide en celdas. La lluvia se representa como un flujo de entrada que reciben las celdas del reservorio, y se infiltra hacia abajo (capas subterráneas) y lateralmente (hacia celdas vecinas). Las entradas de los vecinos son excesos de cantidades de agua que representan flujo vertical o lateral. La transición de estado local es un paso de integración de la ecuación diferencial en la cual se basa el modelo hidrológico.

En [BEA97] se modeló el movimiento de peatones en lugares abiertos con aglomeración de personas. El flujo de peatones no está restringido a carriles, tienen movimientos flexibles, y puede originarse un conflicto cuando más de una persona quiere moverse al mismo lugar. Aquí, las celdas representan una división del suelo por donde caminan las personas. En cada momento entran peatones por alguno de los bordes, y avanzan hacia el lado opuesto, usando reglas locales que evitan colisiones y conflictos entre distintas personas. Se considera una densidad de peatones alta, y para evitar bloqueos, en determinadas ocasiones una persona puede forzar al ocupante de una posición a moverse. El peatón intenta moverse hacia adelante; sino, prueba con alguna de las dos diagonales hacia adelante; sino hace un movimiento al costado. Sino, empuja al ocupante, haciendo que éste dé un paso en dirección hacia adelante y avanza hacia esa celda. El peatón empujado se mueve hacia el costado más cercano al destino, y sino se mueve hacia cualquier costado.

Una última aplicación que nos interesa mencionar incluye los modelos de representación de tráfico urbano. Diversos autores han estudiado este problema desde distintos enfoques, considerando varios aspectos, algunos de los cuales serán resumidos a continuación.

Para modelar el tráfico en una ciudad completa, el estudio puede restringirse a un sector de la misma, lo que permite analizar distintas alternativas que se pueden considerar sólo en un área, y luego extender el modelo. El modelo a plantear dependerá del problema a resolver, y esto está en estrecha relación con los elementos que se toman en cuenta. Se han planteado autómatas celulares con distintas restricciones:

- En los modelos más simples, se propone circulación en mano única: las calles son paralelas, no se considera que un vehículo sobrepase a otro, y se asume que todos los vehículos tienen la misma velocidad [SG96, CCB97]. Esto no permite una verdadera distribución de velocidades máximas, por lo se crean embotellamientos de vehículos, que siguen a un vehículo lento. Para ello se consideraron segmentos compuestos por 3 sectores (3 distintos lugares donde se puede encontrar un vehículo, o un camino de longitud 3), definiendo las reglas de movimiento de un vehículo mediante una gramática con 2 símbolos en el lenguaje (segmento libre u ocupado). Las reglas definen el avance normal, el arranque (si hay un vehículo detrás de otro no se mueven en bloque, sino que avanza el primero), la espera (inversa del arranque) y estados estables (no varían).

- En otros casos se permite que los vehículos se pasen los unos a otros, con lo cual las calles tienen mano única pero dos carriles [SG96]. En este caso se consideran las velocidades de los vehículos, y un vehículo puede pasar a otro, para lo que deberá correrse a otro carril, acelerar, pasar, y volver al carril original. Para sobrepasar a un vehículo se debe considerar el número de sitios desocupados en frente del mismo, el número de sitios desocupados detrás sobre el carril opuesto, y la distancia mínima necesaria para pasar. Las dificultades al simular el cambio de carril residen en que la decisión del cambio no depende únicamente de conocer a los vehículos de la vecindad cercana, sino también saber sobre aquellos aun más lejanos.

- En otros casos, las calles son de doble mano, los vehículos pueden variar la velocidad, y pueden pasarse entre sí [SG96]. Para este caso un vehículo antes de adelantarse deberá chequear la distancia con otro vehículo que viene en dirección contraria. El vehículo debe tener una constante de aceleración mientras pasa a otro, y si detecta un vehículo cercano que viene en dirección contraria, retorna inmediatamente a su propio carril. Se consideran tres tipos diferentes de embotellamiento: uno es la secuencia de comenzar a andar y parar en uno de los dos carriles; el segundo se debe a los conductores arriesgados (que tratan de pasar sabiendo que no hay espacio para retornar a su propio carril); y por último, cuando dos conductores adyacentes (uno de cada carril) tratan de pasar simultáneamente, con lo cual se para todo el tráfico (no se modela marcha atrás).

Finalmente, estos planteos también se extendieron al caso de calles perpendiculares, considerando los cruces en las esquinas. Si se consideran calles perpendiculares pero con mano única, y un solo carril, existen modelos que consideran que:

- los vehículos pueden tener una sola dirección [CCB97]. Se consideran celdas de dos direcciones con intersección de caminos, que poseen dos bordes por los que se puede acceder y dos por los cuales se puede salir. Los vehículos se desplazan en una dirección como fuera mencionado previamente, y además se define el conjunto de estados en el que puede encontrarse la celda de cruce: vacío, yendo hacia la izquierda, o yendo hacia adelante.

- los vehículos pueden desplazarse y variar la dirección: por ejemplo, si un vehículo se dirigía en forma horizontal, puede decidir doblar en una esquina y desplazarse en forma vertical. Los vehículos son inyectados probabilísticamente por las fronteras izquierda y la frontera inferior del sistema, y fluyen hacia afuera determinísticamente desde las fronteras derecha y superior. Cada posición está ocupada por un vehículo que se dirige hacia arriba o por uno que se dirige hacia la derecha, y los vehículos pueden moverse de a un paso a la vez en una unidad de tiempo solo si la posición adyacente en la dirección que se dirige está vacía. El sistema contiene un semáforo que controla el tráfico permitiendo que los vehículos que van hacia arriba puedan moverse en pasos de tiempo pares, y los vehículos que se mueven hacia la derecha en pasos de tiempo impares.

- puede haber conflictos cuando dos vehículos desean acceder a una misma celda vacía, para lo que se asume presencia de semáforos en las intersecciones de las calles [Tad96], o que tiene prioridad el vehículo de la derecha [CCB97].

2.4 SIMULACION PARALELA

Hasta ahora se han analizado algunos formalismos con utilidad para simulación para sistemas de eventos discretos. En muchos casos, la ejecución de estas simulaciones es poco eficiente, cuyo motivo principal está relacionado con causas estadísticas (además de la complejidad de los modelos a especificar). Para que sea estadísticamente significativa, una simulación debe generar un número suficiente de evoluciones típicas del sistema.

Una forma de lograrlo es intentar usar métodos que no requieran la generación de un número grande de ciclos de simulación (usando, por ejemplo, técnicas de reducción de varianza, que tratan de que el número de replicaciones necesarias de ciclos sea proporcional a la varianza de la estimación aleatoria medida en un ciclo dado). Otro medio de obtener una simulación mas rápida es dedicarle mas recursos. En particular, se puede acelerar una simulación usando un sistema multiprocesador en vez de un solo procesador [RW89].

Como la mayoría de las simulaciones estudian sistemas que tienen muchos componentes operando en paralelo, parece razonable suponer que la simulación también explote el paralelismo inherente del sistema. La **simulación paralela de eventos discretos** (PDES), muchas veces llamada simulación distribuída, se refiere a la ejecución de un programa de simulación de eventos discretos en una computadora paralela. El usar múltiples procesadores parece ser una aproximación promisoria para mejorar la velocidad, ya que la mayoría de los sistemas simulados que consiste de muchos componentes operando en paralelo, y así se explota el paralelismo inherente en el sistema.

Se define **aceleración** al tiempo que le toma a un solo procesador hacer una simulación dividido el tiempo que toma al sistema multiprocesador hacer la misma simulación. La aceleración puede pensarse como el número efectivo de procesadores usados (la aceleración ideal con N procesadores es N). El problema con esta definición es especificar qué significa que un solo procesador haga la simulación. Presumiblemente se desea un procesador con la misma capacidad que los del multiprocesador, pero estos pueden no tener la suficiente memoria propia para manejar toda la simulación.

El objetivo de esta sección es presentar algunos mecanismos de PDES muy utilizados, tratando de analizar las soluciones que provean mayor grado de aceleración.

2.4.1 Formas de descomposición paralela

Hay diversas formas de descomponer una simulación para poder procesarla en múltiples procesadores, cuyos méritos y desventajas serán discutidos en esta sección.

Una aproximación obvia es hacer **replicaciones independientes** de una simulación existente en procesadores separados, y luego recolectar los resultados para el análisis estadístico. También permite hacer corridas independientes simultáneas con distintos parámetros. Esta aproximación es muy eficiente porque no hace falta coordinación entre los procesadores, por ende, con N procesadores la aceleración es virtualmente N. En general, distribuir los experimentos será mas eficiente si el sistema llega rápidamente a un estado estable y los tiempos de simulación son largos. La distribución de los experimentos puede no ser factible debido a restricciones de memoria, ya que precisa que todos los procesadores tengan la suficiente como para contener toda la simulación. Otra desventaja es que esta aproximación no permite la descomposición de problemas complejos.

Otra forma es aplicar un **compilador paralelizante** a un programa secuencial. Estos tratan de encontrar secuencias de código que se pueden ejecutar en paralelo, y los planifican en distintos procesadores. Es una buena solución porque es transparente al programador, y puede aplicarse a todo el software de simulación secuencial existente, pero ignora la estructura del problema y explota una parte pequeña del paralelismo existente.

Otra solución es usar **funciones distribuídas,** que asignan distintas tareas de soporte a procesadores individuales (por ejemplo, para la generación de variables aleatorias, recolección de estadísticas, entrada/salida, manipulación de archivos, generación de gráficos, supervisión, etc.). Esta aproximación no tiene abrazo mortal y es transparente al usuario, pero no explota el paralelismo, ofreciendo una cantidad limitada de aceleración.

Recordemos que en un modelo discreto sólo se cambia de estado en puntos discretos de tiempo simulado (un evento), y que se suele usar una lista de eventos con todos los eventos pendientes planificados, así como una variable global reloj para saber el progreso de la simulación. Al usar una solución basada en **eventos distribuídos** se mantiene una lista global de eventos, y cuando hay un procesador disponible se procesa el próximo evento en tiempo simulado. Aquí hace

falta usar algún protocolo para preservar la consistencia, ya que el siguiente evento en la lista puede estar afectado por los eventos procesados en la actualidad. Esta aproximación es particularmente adecuada para sistemas de memoria compartida, ya que la lista de eventos puede ser accedida por todos los procesadores. La aproximación de eventos distribuídos puede ser razonable cuando hay un número pequeño de procesos y un gran número de información global usada por componentes del sistema.

La última clase de soluciones se basa en **descomponer el modelo** en componentes débilmente acoplados. Mientras que el sistema simulado no requiera mucha información y control global, esta aproximación parece ser promisoria, ya que tiene la habilidad de tomar ventaja del paralelismo inherente en el modelo. Sin embargo, requiere cuidado en la sincronización. Los procesos se comunican vía pasaje de mensajes (aunque no necesariamente en una máquina de pasaje de mensajes), que incluyen timestamps representando el tiempo simulado de un evento. El sistema se suele modelar como un grafo dirigido en el que los nodos representan procesos, y los enlaces representan las posibles interacciones.

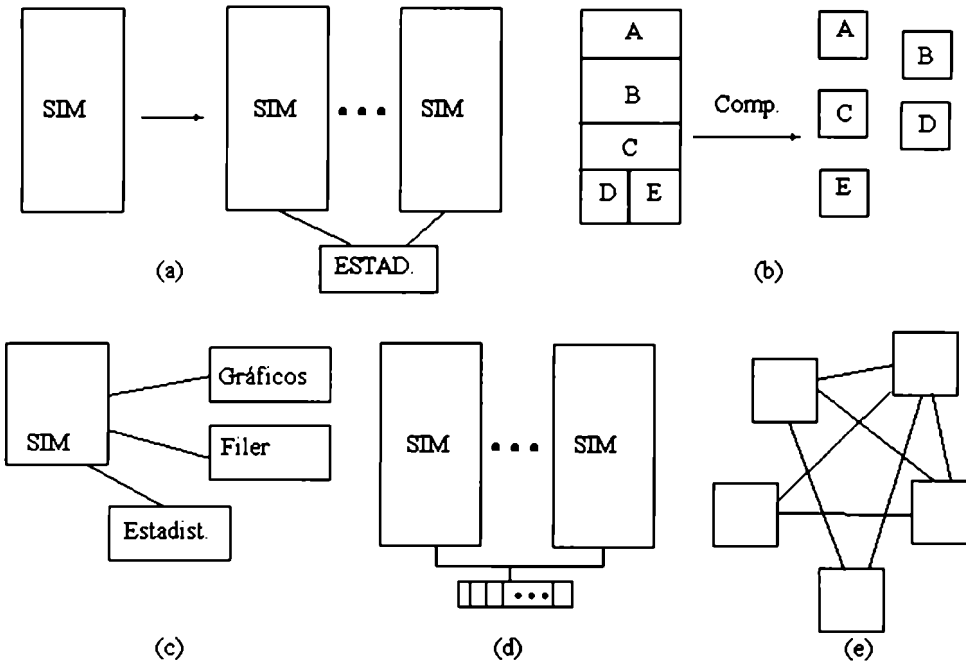


Figura 18. Mecanismos de descomposición. (a) Replicación independiente; (b) Compilador paralelizante; (c) funciones distribuídas; (d) eventos distribuídos; (e) descomposición del modelo.

2.4.2 Sincronización en modelos distribuídos

Al descomponer un modelo y distribuir sus componentes, se debe proveer algún esquema de sincronización. Las formas para atacar estas cuestiones dependerán de si la simulación es dirigida por **tiempo** o por **eventos**, y si es **sincrónica** o **asincrónica** [RW89].

a) Simulación dirigida por tiempo: Aquí el tiempo simulado avanza en incrementos fijos (ticks), en los que cada proceso simula sus componentes. Un tick corto garantiza precisión, pero implica mayor duración de la simulación y mayor tiempo de cómputo. Si la simulación es **sincrónica**, todos los procesos deben simular un tick antes que se pueda comenzar a simular el siguiente. Cuando es **asincrónica**, un proceso puede simular el siguiente tick tan pronto como sus predecesores hayan terminado el último, lo que permite mayor concurrencia pero tiene mayor costo de comunicación (un procesador debe recibir un mensaje de cada uno de sus predecesores para cada tick). La simulación dirigida por tiempo parece ser menos eficiente que la manejada por eventos, ya que puede haber ticks durante los cuales no haya eventos para simular; sin embargo se reduce el overhead de sincronización, lo que la hace adecuada en sistemas con topología dinámica o alto grado de concurrencia.

b) Simulación dirigida por eventos: En ellas, el tiempo se incrementa de un evento al siguiente, por lo tanto puede tener mayor aceleración potencial. En el caso de ser **sincrónica**, se usa un reloj global que mantiene el mínimo tiempo de próximo evento para todos los procesos, y si es **asincrónica**, el reloj local para cada proceso tiene el menor tiempo de evento para ese proceso. Tiene alto desempeño potencial, ya que los procesos pasan menos tiempo esperando a otros, y eventos independientes pueden ser simulados simultáneamente, aunque ocurran en distinto tiempo simulado. Estas solución se ha difundido, ya que parece ser la que provee mayor grado de aceleración, y en las siguientes secciones se analizarán algunos de los resultados existentes.

Ejemplo 13

Sea una red como la de figura. En el caso (a), se considera simulación dirigida por tiempo, y se parte de un reloj global sincronizado (5), por lo que todos los procesos avanzan simultáneamente al siguiente tick (6). El grado de paralelismo mejora en (b), con una solución asincrónica. Aquí, cada proceso avanza un tick si recibe una señal de cada predecesor (entre paréntesis en la figura), mejorando la aceleración. La situación mejora aún más en el caso de una simulación dirigida por eventos, ya que el reloj global (GVT - Global Virtual Time) avanza por eventos, y no de a ticks. En el caso de simulación sincrónica, los procesos deben sincronizarse entre sí hasta todos tener el mismo tiempo. El mayor grado de aceleración se logra en el caso (d), con simulación dirigida por eventos y asincrónica, donde las únicas dependencias están dadas por los datos de la simulación en sí.

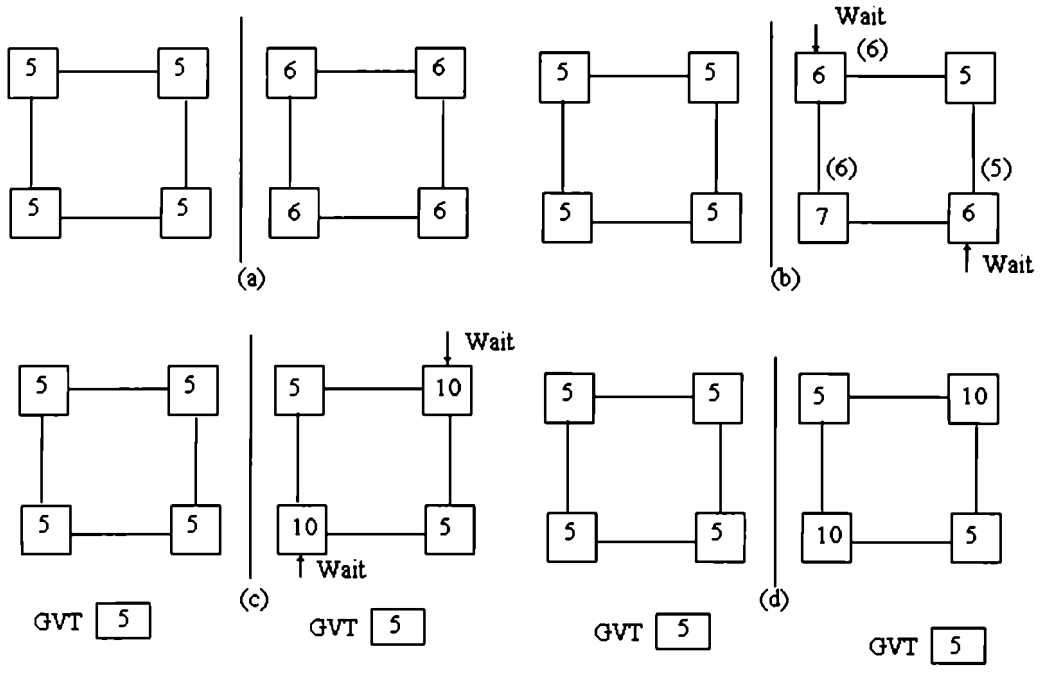


Figura 19. Simulación paralela dirigida por tiempo: (a) Sincrónica; (b) Asincrónica. Dirigida por eventos: (c) Sincrónica; (d) Asincrónica.

2.4.3 Procesos físicos y lógicos

Al hacer simulación paralela asincrónica, es crucial que el simulador (que es de eventos discretos) siempre seleccione el evento con menor timestamp. Si se eligiera alguno con timestamp mayor, podría modificar variables de estado usadas posteriormente, provocando errores de causalidad. La mayoría de las estrategias existentes evitan tales secuencias usando una metodología orientada a procesos que prohibe que un proceso acceda a variables de estado compartidas. El sistema modelado, se considera como compuesto por un conjunto de **procesos físicos** que interactúan en distintos puntos en tiempo simulado, y el simulador se construye como un conjunto de **procesos lógicos** (uno por proceso físico). Todas las interacciones entre procesos físicos se modelan con mensajes enviados entre los procesos lógicos correspondientes, que representan ocurrencia de eventos.

Sean N procesos lógicos P_0,..., P_{N-1}; siendo $Clock_i$ el tiempo simulado hasta el que ha avanzado P_i. Si se procesa un evento, el reloj del proceso se adelanta hasta el timestamp del mensaje asociado con el mismo. Si P_i puede mandar un mensaje a P_j durante la simulación, se dirá que hay un **enlace** entre P_i y P_j. Como cada proceso lógico contiene una parte

del estado, las variables de estado se dividen en un conjunto disjunto, y ningún proceso lógico debe acceder directamente a más de un estado (lo que podría provocar bajo desempeño debido al overhead impuesto por el pasaje de mensajes) [Fuj90].

Una simulación cumple la **restricción de causalidad local** si cada proceso lógico procesa eventos en orden no decreciente de timestamps. El adherir a esta restricción es suficiente aunque no necesario para garantizar que no haya errores de causalidad. Puede no ser necesario, porque dos eventos de un mismo proceso lógico pueden ser independientes, por ende, procesarlos fuera de secuencia no provoca errores de causalidad.

Ejemplo 14

Sean dos eventos, E1 en el proceso lógico LP1 con timestamp 10, y E2 en LP2 con timestamp 20. Si E1 planifica un nuevo evento E3 para LP2 que contiene un timestamp menor que 20, E3 podría afectar a E2, por ende se necesita ejecución secuencial de los tres eventos. Si no existe información acerca de qué eventos podrían ser planificados, la conclusión es que el único evento seguro es el que tiene el menor timestamp, provocando ejecución secuencial.

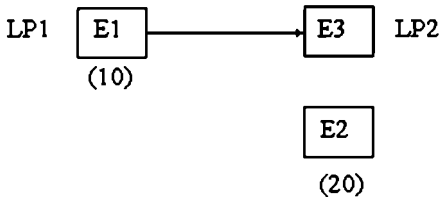


Figura 20. Error de causalidad por sincronización entre procesos lógicos [Fuj90].

La responsabilidad del mecanismo de simulación es asegurar que las restricciones de causa-efecto del sistema simulado, no sean violadas al ejecutar en paralelo. Los mecanismos PDES tratan de proveer soluciones a esta clase de problema. En general se usan dos aproximaciones: las **conservadoras** o las **optimistas**. Las conservadoras **evitan** los errores de causalidad, determinando cuándo es seguro procesar un evento. Por otro lado, las optimistas usan una solución de **detección y recuperación**: los errores se detectan, y se invoca un mecanismo de rollback para recuperarse. Las siguientes secciones detallarán estas propuestas.

2.4.4 Mecanismos conservadores

Los primeros mecanismos propuestos para PDES asincrónica fueron conservadores: se propusieron algoritmos que tratan de determinar cuándo es seguro procesar un evento. Una forma de lograrlo es teniendo en cuenta que si un proceso tiene un evento no procesado con un timestamp dado (y ningún otro con timestamp menor) y se puede determinar que es imposible recibir otro evento con timestamp menor, el evento se puede procesar de forma segura (ya que no habrá una violación de la restricción de causalidad local).

Ejemplo 15

Sean cuatro procesos lógicos como los de la figura. Supongamos que P1 envía el mensaje (3, m1) a P2; P3 manda el mensaje (2,m2) a P2, y P3 no tiene mensajes sin procesar como se ve en la figura (a). Entonces, P2 procesa el mensaje (2,m2) y produce (5, m3) que se manda a P4, como se ve en la fig. (b). En este punto, P2 no puede procesar el mensaje (3,m1) porque P2 está bloqueado por P3; pero P3 también está bloqueado por P2: por ende hay un abrazo mortal.

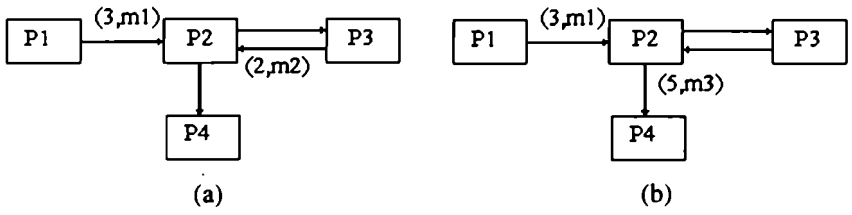


Figura 21. Deadlock en aproximaciones conservadoras.

En [ChM79], se presentó el siguiente algoritmo. Si un proceso lógico tiene mensajes no procesados en todos los enlaces de entrada, pone su reloj local en el mínimo timestamp entre todos ellos, y procesa todos los mensajes que tengan ese timestamp. Todos los mensajes futuros tendrán timestamps posteriores, ya que llegan en orden cronológico. Si el enlace con menor timestamp no tiene mensajes, el proceso lógico se bloquea (puede recibir un mensaje con timestamp menor a todos los otros). Esto puede provocar abrazo mortal si hay un ciclo de procesos lógicos bloqueados. En general, si hay relativamente pocos mensajes con eventos no procesados en relación con el número de enlaces en la red, o si los eventos no procesados se agrupan en una porción de la red, el abrazo mortal puede ocurrir muy frecuentemente [ChM81].

Para evitar el abrazo mortal se trata de especificar **estáticamente** enlaces que indican qué procesos pueden comunicarse con otros [ChM79]. Como la secuencia de timestamps en un enlace es creciente, el timestamp del último mensaje en un enlace de entrada es un límite inferior en todo timestamp subsiguiente. Esta información se puede aprovechar, y usar mensajes **nulos,** que no corresponden a ninguna actividad en el sistema físico. Estos mensajes son una promesa de que no se enviará un mensaje con timestamp menor. Para determinar los timestamps de estos mensajes, puede usarse el reloj de cada enlace de entrada, que provee un límite inferior en el timestamp del siguiente evento sin procesar. Siempre que un proceso termina de procesar un evento, envía un mensaje nulo a cada uno de sus ports de salida indicando el límite. De esta forma el receptor del mensaje nulo puede calcular nuevos límites en sus enlaces de salida, enviar esta información a sus vecinos, etc. De esta forma se rompe el ciclo de procesos bloqueados y se evita el abrazo mortal.

Ejemplo 16

Sean los tres procesos lógicos de la figura (a), que entran en deadlock. Ahora supongamos que cada proceso lógico progresa en avances de 9 unidades de tiempo. En la figura (b) se ve la emisión de mensajes nulos para este caso. Como P2 tiene dos enlaces de entrada con hora local 21, procesa el primer mensaje del enlace elegido y elimina el deadlock.

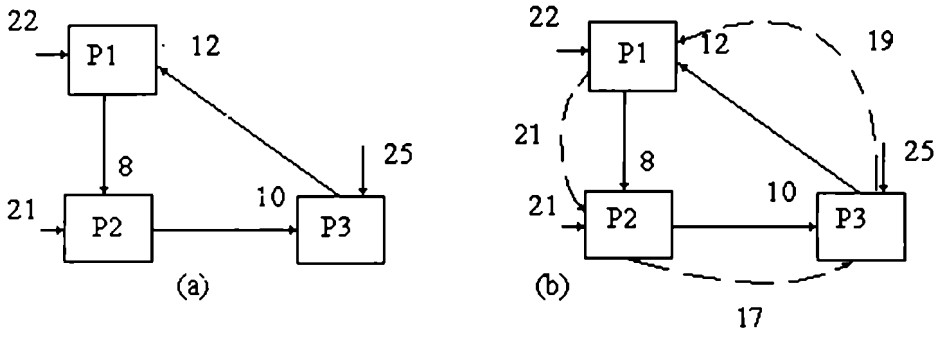


Figura 22. Evitando Deadlock con mensajes nulos.

Intuitivamente, no puede existir abrazo mortal, ya que para que exista, debe haber un ciclo de procesos bloqueados (por ejemplo, P1 bloquea a P2, P2 bloquea a P3, etc., hasta Pn, y Pn bloquea P1). Sea ti el tiempo local del reloj de Pi. Luego, si Pi está bloqueando a Pj, Pj debe estar esperando un mensaje de Pi y su tiempo de reloj local, tj debe ser igual al reloj del enlace de Pi a Pj. Pero si Pi también está bloqueado, debe haber actualizado los relojes de sus enlaces de salida para que sean mayores o iguales al reloj local de Pi, ti. Luego, si tj>=ti, tl >= tl. Por lo menos un proceso en el ciclo, digamos Pi, tiene relojes del enlace de salida estrictamente mayores que su reloj local, por ejemplo, ti>ti+1. Luego, tl>tl; una contradicción. Por ende, el mecanismo evita el abrazo mortal mientras no haya ciclos en los que el incremento de timestamp colectivo de un mensaje que atraviesa el ciclo sea cero.

Se llamará **precálculo** (lookahead) a la habilidad de predecir qué pasará (o, más importante, qué NO ocurrirá) en el futuro simulado, lo que puede usarse para determinar qué eventos son seguros de procesar. El precálculo es necesario para evitar abrazo mortal, cuando se determinan los timestamps asignados a mensajes nulos.

Los mensajes nulos, en lugar de enviarse después de cada evento, pueden transmitirse por demanda (como en el protocolo SRADS [Rey82]). Aquí, siempre que un proceso está por bloquearse pide el siguiente mensaje (nulo o no) al emisor conectado al enlace de entrada. Esto permite reducir la cantidad de mensajes nulos, aunque acarrea mayor overhead para los mensajes nulos. Esta solución es útil para sistemas sincrónicos con interacciones infrecuentes.

Una aproximación alternativa usa un mecanismo para detectar cuando la simulación entró en abrazo mortal, y otro para romperlo. El abrazo mortal puede romperse porque los mensajes con menor timestamp siempre son seguros de procesar. Cuando un proceso se bloquea, manda un mensaje (llamado "**probe**") con el timestamp de su reloj local a alguno de sus

predecesores para obtener información de sus relojes. El receptor le envía su reloj local, si este es posterior al reloj local del proceso que lo pide; sino, envía probes a sus predecesores. Esto precisa que el mensaje contenga el camino atravesado, y los relojes locales de los procesos en el camino. Esta solución se puede modificar para detectar y recuperar deadlocks locales (en sólo una porción de la red), haciendo un preprocesamiento para identificar todas las subredes que podrían tener abrazo mortal, y aplicar la técnica en las subredes.

Ejemplo 17

En la Figura 23, supongamos que el precálculo para los procesadores P2 y P3 es 2. Los números arriba de los procesadores en la figura muestran sus relojes locales antes que se envíen los mensajes. P2 envía el mensaje nulo (4, null) a P3. Luego P3 actualiza su reloj local a 4 y envía el mensaje (6, null) a P2. P2 puede procesar el mensaje (3, m1) y actualizar su reloj a 3, produciendo el mensaje (8, m4), que se envía a P3.

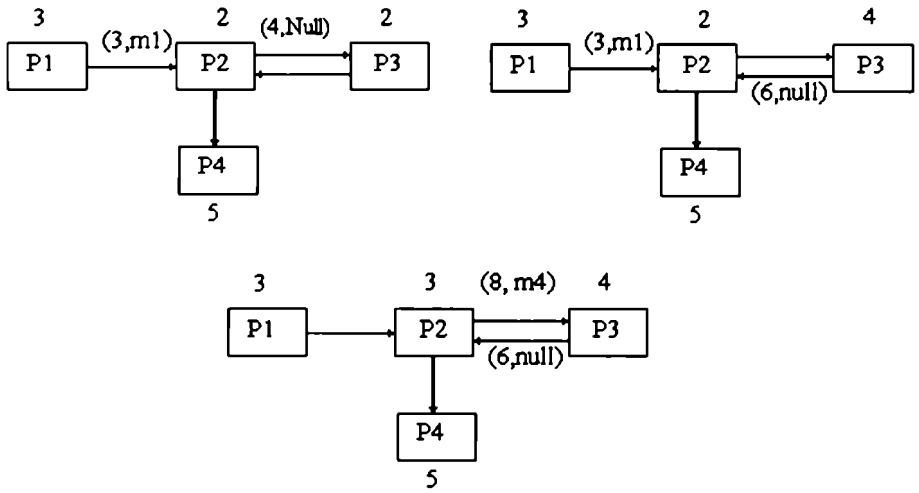


Figura 23. Uso de mensajes nulos para la Figura 21.

Lubachevsky [Lub89] usa un mecanismo con mensajes nulos y precálculo, y una ventana de tiempo simulado móvil, reduciendo el overhead existente para determinar un evento seguro. El límite inferior de la ventana es el menor timestamp de todo evento sin procesar, y sólo los eventos no procesados cuyo timestamp reside en la ventana pueden procesarse. Esta ventana reduce el "espacio de búsqueda" que se debe recorrer para determinar si un evento es seguro.

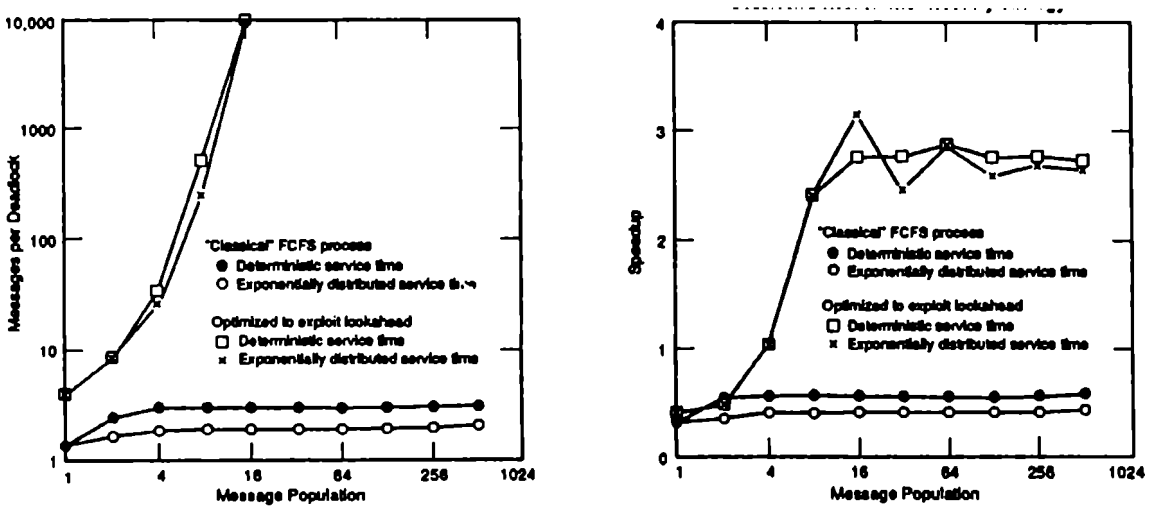


Figura 24. Desempeño de algoritmos conservadores (a) Eficiencia; (b) Aceleración [Fuj90].

Existen diversos estudios empíricos y analíticos del desempeño de los esquemas conservadores. En todos los casos pudo verse que el grado de precálculo juega un rol critico en el desempeño, ya que un proceso con precálculo puede habilitar a otros procesos para procesar de forma segura mensajes de eventos pendientes. En la Figura 24 se muestran medidas de desempeño al simular una red de encolado con un servidor central usando el algoritmo de detección y recuperación de abrazo mortal en un Butterfly BBN (cada proceso lógico ejecutando en un procesador separado con un número fijo de mensajes). Se muestra el número promedio de mensajes procesados entre abrazos mortales, y la aceleración relativa a

una solución secuencial. El detectar y recuperar abrazo mortal es generalmente superior, particularmente cuando hay un número pequeño de procesos en el sistema.

Por otro lado, las mediciones extensivas de los algoritmos para evitar, detectar y recuperar abrazos mortales ejecutando en un multiprocesador, obtuvieron desempeño bastante malo excepto en pocos casos especializados. Si hay ciclos, apenas se logra aceleración. La demora parece afectar el desempeño del esquema de mensajes nulos, pero no en el esquema de detección y recuperación. La conclusión general es que las técnicas de Chandy-Misra no son viables. La desventaja más obvia de estas soluciones es que no se explota por completo el paralelismo disponible, ya que si un evento puede afectar a otro, deben ejecutarse secuencialmente (forzando ejecución secuencial sin ser necesaria). Otro problema es la robustez: cambios menores en la aplicación pueden tener un efecto catastrófico en el desempeño. Finalmente, las configuraciones son estáticas: no se pueden crear dinámicamente nuevos procesos, y su interconexión también debe definirse estáticamente. Puede verse que es necesario que el programador provea conocimiento con respecto al comportamiento del proceso lógico, y aquí, estimaciones conservadoras resultan en desempeño pobre, y las optimistas pueden violar las restricciones de causalidad. Una de las desventajas más serias con los algoritmos pesimistas es que el programador debe cuidar los detalles del mecanismo de sincronización para lograr buen desempeño.

2.4.5 Mecanismos optimistas

Como fuera comentado, los métodos optimistas no evitan errores de causalidad sino que los detectan y recuperan, explotando paralelismo en donde podrían ocurrir errores de causalidad pero de hecho no ocurren. El protocolo optimista más conocido es el mecanismo Time Warp [Jef85]. Aquí, si un proceso lógico recibe un mensaje de un evento con hora menor a la del reloj local, se invoca una rutina de recuperación (**rollback**). El evento que provoca el rollback se llama **dispersor**. La recuperación implica deshacer los efectos de todos los eventos procesados prematuramente por los procesos que reciben el dispersor (es decir, los eventos procesados con timestamp mayor que el del dispersor).

Para implementar el mecanismo, los mensajes usan dos timestamps: el de emisión y de recepción. El reloj local (LVT - Local Virtual Time) de un proceso es el mínimo tiempo de recepción de todos los mensajes no procesados. Mientras haya alguna entrada, un proceso pueden ejecutar eventos, y su LVT puede adelantarse a los de sus predecesores, pudiendo recibir un mensaje de un evento del pasado. En este caso, el proceso hace rollback, y la simulación se rehace teniendo en cuenta el nuevo evento.

Un rollback, además de modificar el estado, debe enviar mensajes a otros procesos. Para lo primero se debe almacenar el estado del proceso (en el rollback se restaura un vector de estado viejo). Para lo segundo, se envía un mensaje negativo o **antimensaje** que elimina al original cuando llega a su destino. Los mensajes de eventos se llaman mensajes **positivos**. Si se recibe un antimensaje de un mensaje positivo que fue procesado, debe restaurarse el estado, y eliminar el mensaje positivo. Para cancelar los mensajes erróneos, deben almacenarse los estados de cada proceso desde el último tiempo "correcto" (llamado GVT - Global Virtual Time). Este es el mínimo de los LVTs de todos los procesos y los tiempos de emisión de todos los mensajes enviados pero no procesados.

Asumiendo memoria infinita, el algoritmo Time Warp no entrará en abrazo mortal, porque los procesos individuales no entran en abrazo mortal mientras tengan entradas, y el GVT siempre crece. Además tiene mayor aceleración potencial que las aproximaciones conservadoras pero al costo de mayor requerimiento de memoria. No necesita conocer la topología de las posibles interacciones entre procesos, y como ningún evento con timestamp menor que el GVT puede anularse, el almacenamiento usado por tales eventos puede ser utilizado. Las operaciones irrevocables (como las de entrada/salida) no pueden hacerse hasta que el GVT borre el tiempo simulado en el cual ocurrió la operación. Este proceso se llama **colección de fósiles.** Una desventaja es que requiere mucha memoria. Si se calcula el GVT seguido, se pueden sacar mensajes de la entrada y salida, pero habrá mayor overhead en tiempo de procesamiento. Otra forma de ahorrar memoria es descartar los estados con timestamps mayores que el LVT que surgen a partir de los rollbacks.

La cancelación **perezosa** [Gaf88] es una optimización para reparar el daño causado por los cálculos incorrectos en lugar de repetirlos por completo. El mecanismo de Time Warp descripto usa cancelación **agresiva**: si se hace rollback hasta un instante dado, se mandan inmediatamente los antimensajes correspondientes. Pero un dispersor puede no alterar el cálculo de los eventos de los que hace rollback, y los mensajes (positivos) generados por estos eventos pueden ser los mismos. Con cancelación perezosa, los procesos no mandan inmediatamente los antimensajes para todo rollback, sino que miran si la reejecución del cálculo regenera los mismos mensajes. En este caso no hay necesidad de cancelar el mensaje. Un antimensaje sólo se manda luego que el LVT pasa su tiempo sin regenerarlo.

En la mayoría de los casos la cancelación perezosa es más rápida (hay evidencia empírica) pero precisa más memoria. También permite que los cálculos se ejecuten en menos tiempo que el caso de camino crítico, ya que los cálculos con entrada incorrecta (o parcialmente correcta) pueden generar resultados correctos. La cancelación perezosa toma ventaja del precálculo aunque la aplicación no haya sido programada explícitamente para explotarlo, a diferencia de las aproximaciones conservadoras que requieren que se especifique explícitamente, pero tiene mayor overhead.

Ejemplo 18

En la figura (a), el LVT está en el tiempo simulado 29, y llega un dispersor con tiempo 22. Entonces, se inserta en la cola, el LVT pasa a ser 22, y se envían, como antimensajes, los mensajes de la cola de salida 25 y 33, dispersando el evento de llegada.

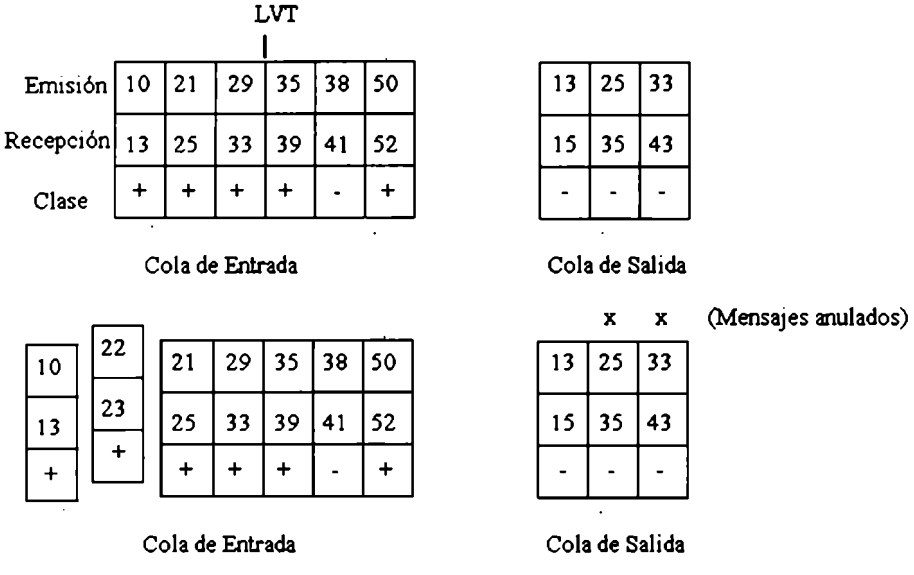


Figura 25. Llegada de un antimensaje usando Time Warp.

El protocolo de ventanas de tiempo también fue propuesto para protocolos optimistas. Aquí se usan para prevenir que los cálculos incorrectos se propaguen en el futuro simulado. La aproximación de ventana de tiempo en movimiento usa una ventana de tiempo fijo de tamaño W. Aquí sólo se eligen para procesar los eventos con timestamps en [T, T+W], donde T es el evento con menor timestamp [SBW88]. Las críticas al método dicen que no puede distinguir buenos cálculos de los malos, por ende pueden impedir el progreso de cálculos correctos. Más aún, los cálculos incorrectos que están adelantados en el futuro simulado ya están discriminados por el mecanismo de planificación de Time Warp, que da precedencia a las actividades con timestamps pequeños. Finalmente, no está claro cómo debe determinar el tamaño de la ventana.

En [MWM88], se propone un esquema en el que un mensaje dispersor provoca que un proceso envíe mensajes de control especiales para detener la dispersión de cálculos erróneos. Los procesos que pueden ser "infectados" por cálculos erróneos se notifican cuando se detecta un error. La desventaja es que algunos cálculos correctos pueden congelarse innecesariamente, y que el conjunto de procesadores afectado por cálculos erróneos podría ser significativamente más grande que el conjunto real; por ende, pueden mandarse algunos mensajes de control innecesarios. Finalmente, se debe conocer la velocidad a la que puede dispersarse el cálculo erróneo y el tiempo necesario para transmitir los mensajes de control. El determinar estos límites puede ser difícil.

Es importante cancelar un cálculo incorrecto antes que este pueda dispersarse en el sistema, porque sino que los cálculos erróneos pueden dispersarse a través del sistema mientras hay antimensajes tratando de encontrarlos. Dicho comportamiento puede evitarse dando a los antimensajes mayor prioridad que los mensajes positivos. Un mecanismo, llamado de cancelación directa [Fuj89b] (para memoria compartida) hace que siempre que un evento E1 planifique otro E2, y deje un puntero de E1 a E2. Este puntero se usa si luego se decide que E2 debería cancelarse (usando cancelación perezosa o agresiva). Por contraste, los sistemas de Time Warp convencionales deben buscar para encontrar mensajes cancelados. Las ventajas de este mecanismo son dobles: reduce el overhead asociado con la cancelación de mensajes, y rápidamente encuentra cálculos erróneos para minimizar el daño.

Diversos análisis empíricos indican que mientras que el precálculo mejora el desempeño de los algoritmos pesimistas, no es un prerrequisito para Time Warp. La Figura 26 muestra una comparación entre Time Warp y los algoritmos de evitar, detectar y recuperar abrazo mortal. Usando cargas sintéticas se midió el efecto del precálculo, la distribución de incrementos de timestamp (localidad temporal), topología (específicamente localidad espacial), y granularidad de los cálculos en el desempeño. Se encontró que Time Warp puede lograr aceleración en proporción de la cantidad de paralelismo disponible en la carga bajo condiciones de prueba saturadas (mas paralelismo que procesadores) y no saturadas (menos paralelismo que procesadores). Aunque los resultados son alentadores, el overhead de salvado de estado puede degradar bastante el desempeño.

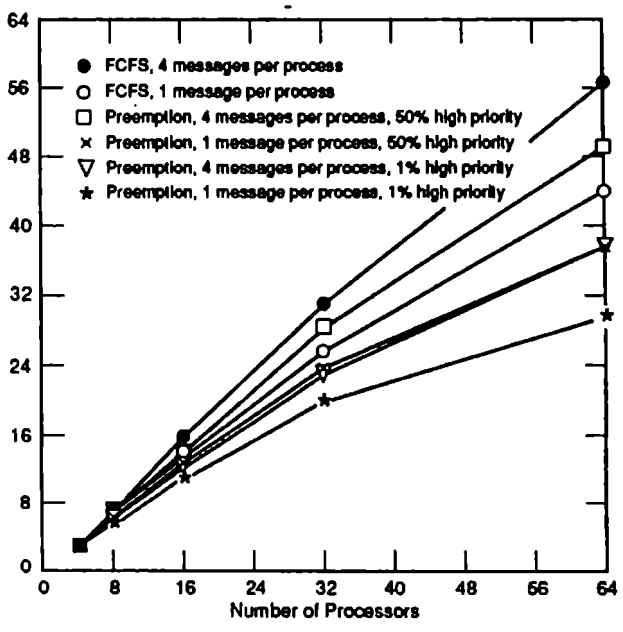


Figura 26. Aceleración de simulación usando Time Warp (256 procesos lógicos) [Fuj90].

Asumiendo un costo fijo para el rollback, se demostró que Time Warp puede mejorar los algoritmos de Chandy-Misra en una cantidad arbitraria en proporción al numero de procesadores disponibles. Intuitivamente, esto ocurre porque el paralelismo que se pierde ejecutándose de forma conservadora puede ser arbitrariamente grande, y la ejecución optimista puede explotar este paralelismo [LM90].

El rollback produce tres overheads: restaurar las colas de entrada, de estado, y de salida. El primero tiene costo despreciable, porque sólo hacen falta unas pocas instrucciones de maquina para poner el puntero en el próximo evento a ser procesado. El overhead principal en un rollback está en restaurar la cola de salida, lo que implica mandar antimensajes. Por otro lado, mandar antimensajes toma menos tiempo que mandar mensajes positivos: sólo se mandan mensajes preexistentes mientras que mandar un mensaje positivo requiere asignar un buffer y llenarlo con datos, además de otros costos.

Una cuestión crítica es si el sistema exhibirá comportamiento de thrashing (la mayoría del tiempo ejecutando cálculos incorrectos y haciendo rollback). Finalmente, el principal problema de los algoritmos optimistas, es que tienden a usar muchas más veces de memoria que sus contrapartes conservadoras. El uso de una política de planificación de procesos inadecuada al implementarlo puede ser catastrófico.

3. FORMALISMO CELL-DEVS

Este capítulo está dedicado a presentar un paradigma de descripción de modelos celulares que pueden definirse como modelos atómicos DEVS con distintos tipos de demoras. La idea es poder estudiar fenómenos diversos, permitiendo que cada submodelo pueda ser construido por medio de distintas técnicas de modelado (la que esté mejor adaptada para analizar el comportamiento deseado). En todos los casos se presentan formalismos para espacios de celdas de n dimensiones con distintos dominios para el conjunto de estados, pero se hace especialización en espacios bidimensionales binarios, que son el principal objeto del presente trabajo.

Algunas construcciones útiles que fueron consideradas para modelar las demoras que existen en sistemas reales fueron las de Transporte e Inerciales. Esas demoras no sólo son útiles para diseñar aplicaciones de hardware, sino para modelar otras clases de fenómenos físicos que puedan describirse como espacios de celdas.

El capítulo comienza por especificar distintos tipos de autómatas celulares. Luego se presenta un mecanismo de especificación para modelos acoplados cerrados. Finalmente se introduce una especificación formal de modelos Cell-DEVS que pueden acoplarse con otros modelos.

3.1 Definición formal de autómatas celulares

En esta sección se introduce una definición formal para los paradigmas de autómatas celulares que fueron presentados informalmente en la sección 2.3. Se comienza analizando autómatas celulares conceptuales, y luego se definen autómatas celulares ejecutables (sincrónicos y asincrónicos).

3.1.1 Autómatas celulares conceptuales

Un autómata celular conceptual puede definirse como:

$$CCA = < S, n, C, \eta, N, T, \tau, c.Z_0^+ >$$

donde

S es el alfabeto utilizado para representar el estado de cada celda;
n es la dimensión del espacio de celdas;
C es el conjunto de estados para el espacio de celdas;
η es el tamaño de la vecindad;
N es el conjunto de vecindad;
T es la función de transición global;
τ es la función de cómputo local; y
$c.Z_0^+$ es la base de tiempo (discreta) del autómata celular.

Analicemos con detalle la definición de cada uno de estos conjuntos:

$S \subseteq Z \wedge \#S < \infty$;

$n \in N$;

Notación 1

De aquí en más, se llamará $C_c \in S$ al estado de la celda c, siendo $c \in Z^n$, $c = (i_1,...,i_n)$ la posición de la celda c en el espacio n-dimensional. Aquí, $\forall k \in [1,n]$, $i_k \in Z$ es la posición de la celda en la k-ésima dimensión.

Para el caso de autómatas celulares conceptuales, $\forall k \in [1,n]$, $i_k \in [-\infty,\infty]$.

$\mathbf{C} = \{ C_c / c \in Z^n \wedge C_c \in S \}$;

Para el caso de autómatas celulares bidimensionales [WG98], $C = \{ C_{ij} / i,j \in Z \wedge C_{ij} \in S \}$;

$\eta \in N$;

$\mathbf{N} = \{ (v_{k1},...,v_{kn}) / \forall (k \in N, k \in [1, \eta]) \wedge (i \in N, i \in [1, n]), v_{ki} \in Z \}$;

$\mathbf{T}: C \times c.Z_0^+ \rightarrow C$;

$\tau: C_c \times N \times c.Z_0^+ \rightarrow C_c \ \forall c \in Z^n$. En este caso, $C_c[t+c] = \tau (C_{c+v1}[t],...,C_{c+v\eta}[t])$, donde $t \in c.Z_0^+ \wedge \forall (k \in N, k \in [1, \eta])$, $vk \in N \wedge c + vk = (i_1+v_{k1}, ..., i_n+v_{kn})$.

Para autómatas celulares de dos dimensiones, la función de cálculo local [WG98] puede definirse como:

$\tau: S^\eta \times c.Z_0^+ \rightarrow S$. En este caso, $C_{ij}[t+c] = \tau(C_{i0,j0}[t],..., C_{i\eta, j\eta}[t])$, con $(ik = i+v_{ki} \wedge jk = j+v_{kj}) \forall (i, j \in Z) \wedge ((v_{ki}, v_{kj}) \in N, k \in N, k \in [1, \eta])$.

$c.Z_0^+ = \{ 0, c, 2c, 3c, ... \} = \{ i / i \in N, i = c.j \wedge j \in N \}$

Como podemos ver en la definición, el modelo está compuesto de un espacio **n**-dimensional de celdas (**C**). El espacio de estados progresa en pasos de tiempo discreto, dado que la base de tiempo discreta está definida por $c.Z_0^+$ (un conjunto de valores enteros separados entre sí por una constante de tiempo).

En este caso se incluyen dos definiciones: una considera espacios de n dimensiones en los cuales los vecinos son homogéneos (pero pueden pertenecer a cualquier parte del espacio de celdas). Por otro lado, se incluyen definiciones para autómatas celulares de dos dimensiones con vecindades adyacentes a la celda origen. Esta especialización fue considerada debido a que esta clase de espacios de celdas es el objetivo primordial del presente trabajo.

El estado de cada celda en el espacio puede tomar un valor entre los de un alfabeto finito (**S**). El dominio del conjunto S puede ser cualquier conjunto finito discreto, como fuera analizado informalmente en la sección 2.3. De aquí en más se considerará esta definición presentada para el conjunto S, sin pérdida de generalidad en la definición del alfabeto de la celda. Esto se debe a que, a pesar de que puede usarse cualquier conjunto finito de símbolos, se puede hacer un mapeo de estos en el conjunto S definido en esta sección.

La vecindad está definida como una lista de η vecinos de n dimensiones. En la definición se usa un índice (k) que permite identificar el número de vecino, y un segundo índice (i) que indica la dimensión de una de las posiciones del vecino. En este caso, los vecinos son una n-upla de posiciones relativas a la celda origen.

Como vemos, las vecindades consideradas implican autómatas celulares homogéneos, aunque las definiciones pueden extenderse fácilmente para autómatas no homogéneos. En este caso, N se debe definir como un arreglo de listas de vecindades, especificado formalmente como:

$$N = \{N_c / c \in Z^n \}$$

$$N_c = \{ (v_{k1},...,v_{kn})_c / \forall (k \in N, k \in [1, \eta_c]) \wedge (i \in N, i \in [1, n]), v_{ki} \in Z \};$$

Si suponemos el caso de espacios bidimensionales,

$$N = \{N_{kl} / k, l \in Z \}$$

donde

$$N_{kl} = \{(i_p,j_p) / \forall p \in N, p \in [1,\eta_{kl}], i_p, j_p \in Z \wedge \eta_{kl} \in N \}$$

En este caso, cada celda tendrá una lista de vecinos de η_c (η_{kl}) elementos, compuesta por tuplas de índices relativos a la celda de origen.

La evolución del espacio de estados del autómata está definida por la ejecución de una función de transición global (**T**) que cambia el estado de todo el espacio de celdas. El comportamiento de esta función de transición depende del resultado de la ejecución de funciones de cálculo locales (τ) que corren en el entorno de la vecindad de una celda (**N**). Conceptualmente, el cálculo de estas funciones locales se hace de forma sincrónica y paralela en cada celda del espacio. En el caso de autómatas celulares bidimensionales, $C_{ij}[t]$ es el estado de la celda (i,j) en el tiempo simulado *t*, y τ denota la función de cálculo local para el autómata. Los parámetros (i0,j0),..., (iη,jη) definen la vecindad de la celda.

3.1.2 Autómatas celulares ejecutables sincrónicos

En el caso anterior, los índices del espacio de celdas podían incluir infinitas celdas. Como fuera comentado previamente, estamos interesados en modelos ejecutables. Para lo cual un autómata celular ejecutable sincrónico se define formalmente como:

$$CA = < S, n, \{t_1,...,t_n\}, C, \eta, N, B, T, \tau, c.Z_0^+ >$$

donde

S es el alfabeto utilizado para representar el estado de cada celda;

n es la dimensión del espacio de celdas;

$\{t_1,...,t_n\}$ es la cantidad de celdas en cada una de las dimensiones;

C es el conjunto de estados para el espacio de celdas;

η es el tamaño de la vecindad;

N es el conjunto de vecindad;

T es la función de transición global;

B es el conjunto de celdas de borde;

τ es la función de cómputo local; y

$c.Z_0^+$ es la base de tiempo.

En este caso,

$S \subseteq Z \wedge \#S < \infty$;

$n \in N$; $n < \infty$;

$\{t_1,...,t_n\} \in N$, con $t_k < \infty \; \forall \; k \in [1,n]$;

$C = \{ C_c / c \in I \wedge C_c \in S \}$, con

$$I = \{ (i_1,...,i_n) / i_k \in N \wedge i_k \in [1, t_k] \; \forall \; k \in [1, n] \} \qquad (1)$$

Para el caso de autómatas celulares bidimensionales [WG98], se tiene $C = \{ C_{ij} / i,j \in N \wedge i \in [1,f], j \in [1,c] \}$, donde $t_1 = f$ es la cantidad de filas y $t_2 = c$ es la cantidad de columnas;

$\eta \in N, \eta \leq \prod_{i=1}^{n} t_i$, es el tamaño de la vecindad;

$\mathbf{N} = \{ (v_{k1},...,v_{kn}) / \forall (k \in N, k \in [1, \eta]) \wedge (i \in N, i \in [1, n]), v_{ki} \in \mathbf{Z} \wedge t_i - | v_{ki} | \geq 0 \}$;

Para el caso de autómatas celulares bidimensionales con vecindades adyacentes a la celda de origen [WG98],

$\mathbf{N} = \{ (i_p, j_p) / i_p, j_p \in \mathbf{Z} \wedge i_p, j_p \in [-1, 1] \ \forall p \in N, p \in [1, \eta] \}$.

$\mathbf{B} = \{\varnothing\}$ si el espacio de celdas es toroidal; o
$\mathbf{B} = \{C_b / C_b \in C$ con $b \in \mathbf{I} \}$, con **I** definido como en (1). En este caso, el conjunto B está sujeto a la restricción que

$\tau_b \neq \tau_c = \tau \ \forall (C_c \notin B) \wedge (C_b \in B)$.

En general, si $B \neq \{\varnothing\}$, B suele definirse como:

$B = \{C_b / C_b \in C$ con $b \in L\}$, siendo $L = \{ (i_1,...,i_n) / i_j = 0 \vee i_j = t_j \ \forall j \in [1, n] \}$, y sujeto a que $\tau_b \neq \tau_c = \tau \ \forall c \notin L$.

Para el caso de espacios bidimensionales [WG98] se obtiene:

- $B = \{\varnothing\}$ si el espacio de celdas es toroidal; o
- $B = \{C_{ij} / C_{ij} \in C \wedge (i = 1 \vee i = f) \wedge (j = 1 \vee j = c) \}$ sino.

$\mathbf{T}: C \times c.\mathbf{Z}_0^+ \rightarrow C$;

$\tau: C_c \times N \times c.\mathbf{Z}_0^+ \rightarrow C_c$; donde $C_c[t+c] = \tau(C_{c \oplus v1}[t],...,C_{c \oplus vn}[t])$, (2)

con $t \in c.\mathbf{Z}_0^+ \wedge \forall (k \in N, k \in [1, \eta])$, $vk \in N \ \wedge c \oplus vk = (i_1 + v_{k1} \bmod(t_1), \ ..., \ i_n + v_{kn} \bmod(t_n))$ en el caso que $B = \{\varnothing\}$, y

$C_b[t+c] = \tau_b(C_b[t])$, $\forall b \in B$, con $t \in c.\mathbf{Z}_0^+$. En este caso, $\tau_b \neq \tau_c = \tau \ \forall c \notin B$, y para estos últimos, C_c se calcula como en (2).

Como puede verse, la definición de los autómatas celulares ejecutables difiere en algunos aspectos de la de los autómatas celulares conceptuales. Por un lado, se acota el tamaño del espacio de celdas en cada una de las dimensiones $(t_1,...,t_n)$, que en este caso también es finita. Por ende, los subíndices de las celdas también se acotan (números naturales finitos).

Otra restricción de los autómatas ejecutables es la pérdida de homogeneidad del espacio de celdas. Ello implica la inclusión de un conjunto de celdas de borde (**B**) que puede ser vacío (para los autómatas celulares toroidales), o un conjunto de celdas prefijado. Estas celdas tienen distinto comportamiento que el resto, por ende, usan distintas funciones de transición local, que se calculan de forma diferente en los bordes y en el resto del autómata.

Como en el caso de los autómatas celulares conceptuales, la semántica de la función de transición T implica la ejecución simultánea y paralela de las funciones de transición local en todo el autómata celular. Esta puede plantearse formalmente por medio de la siguiente regla:

$$C_c \in C \ \forall c = \{ (i_1,..., i_n) / i_j \in [1, t_j] \ \forall j \in [1,n] \}, \qquad t \in c.\mathbf{Z}_0^+$$

$$C[t+c] = T(C[t]), \text{ con } C_c[t+c] = \tau(C_c[t]) \ \forall c = \{ (i_1,..., i_n) / i_j \in [1, t_j] \ \forall j \in [1,n] \} \ ; \ t = t + c$$

Es decir, que partiendo de un espacio de celdas dado para el instante *t*, la función de transición global provoca un cambio en el espacio de celdas en el siguiente paso (de longitud c), que puede verse como el cálculo individual y paralelo de la función de transición local en cada celda.

3.1.3 Autómatas celulares ejecutables asincrónicos

Finalmente, para el caso de autómatas celulares asincrónicos, cuya especificación sea ejecutable, se puede considerar la siguiente definición formal:

$$ACA = < S, n, \{t_1,...,t_n\}, C, \eta, N, B, T, \tau, R_0^+ >$$

donde

S es el alfabeto utilizado para representar el estado de cada celda;

n es la dimensión del espacio de celdas;

$\{t_1,...,t_n\}$ es la cantidad de celdas en cada una de las dimensiones;

C es el conjunto de estados para el espacio de celdas;

η es el tamaño de la vecindad;

N es el conjunto de vecindad;

B es el conjunto de celdas de borde;

Nevs es una lista de próximos eventos;

T es la función de transición global;

τ es la función de cómputo local; y

R_0^+ es la base de tiempo (continua) del autómata celular.

En este caso,

$S \subseteq Z \wedge \#S < \infty$;

$n \in N$, $n < \infty$;

$\{t_1,...,t_n\} \in N$, siendo $t_k < \infty \; \forall \; k \in [1,n]$;

$C = \{ C_c / c \in I \wedge C_c \in S \}$, con **I** definido como en (1);

$\eta \in N, \eta \leq \prod_{i=1}^{n} t_i$;

$N = \{ (v_{k1},...,v_{kn}) / \forall (k \in N, k \in [1, \eta]) \wedge (i \in N, i \in [1, n]), v_{ki} \in Z \wedge t_i - | v_{ki} | \geq 0 \}$;

$B = \{\varnothing\}$ si el espacio de celdas es toroidal; o
$B = \{C_b / C_b \in C$ con $b \in I \}$, con **I** definido como en (1). En este caso, el conjunto B está sujeto a la restricción que $\tau_b \neq \tau_c = \tau \; \forall (C_c \notin B) \wedge (C_b \in B)$;

$T: C \times R_0^+ \rightarrow C$;

$\tau: C_c \times N \times R_0^+ \rightarrow C_c$; donde $C_c[t_p] = \tau(C_{c \oplus v1}[t],...,C_{c \oplus v\eta}[t])$, (3)
con $t \in R_0^+ \wedge \forall (k \in N, k \in [1, \eta]), vk \in N \wedge c \oplus vk = (i_1+v_{k1} \bmod(t_1), \; ..., \; i_n+v_{kn} \bmod(t_n))$ en el caso que $B = \{\varnothing\}$, y

$C_b[t_p] = \tau_b(C_b[t])$, $\forall\ b \in B$, con $t \in R_0^+$. En este caso, $\tau_b \neq \tau_c = \tau\ \forall\ c \notin B$, y para estos últimos, C_c se calcula como en (3).

Como puede verse, la mayoría de los conjuntos y funciones definidos son similares que para el caso sincrónico. Los cambios se deben a que aquí la base de tiempo es continua (es decir, que las variables de tiempo $t \in R_0^+$). Esta definición asincrónica provoca que el formalismo deba incluir una lista de próximos eventos, que debe utilizarse para almacenar la información correspondiente a los eventos a ser tratados. Esta lista se define como **Nevs** = $\{ (c, t) / c \in I \wedge t \in R_0^+ \}$, donde c es la posición de una celda en el espacio, **I** está definido como en (1), y t es la hora del evento correspondiente. Para el cálculo de la función de cómputo local se utilizarán los valores en esta lista de eventos. Aquí, $t_p = \min\{t_i\}_{i=1}^{n}$, con i, $p \in N / t_i \in R_0^+$, y $b = c_p \vee c = c_p$, con $(t_i, c_i) \in$ Nevs. Aquí, $\tau_b \neq \tau_c = \tau\ \forall\ c \notin I$ (1), y para estos últimos, C_c se calcula como en (3).

También debe considerarse que la semántica de la función de cálculo global (**T**) es diferente al caso anterior. Aquí, esta función implica la ejecución un grupo de celdas no latentes, llamadas inminentes. La ejecución de esta función se hace simultáneamente en todas las celdas inminentes para un tiempo simulado dado, tal como fuera analizado en la sección 2.3.1. La semántica de este comportamiento puede especificarse como:

$$C_c \in C\ \forall c \in I \quad (1), \qquad C_p = \{ (t_p, c_p) / (t_p, c_p) \in \text{Nevs} \} \wedge t_p = \min\{t_i\}_{i=1}^{n}$$

$$C[t_p] = T(C[t]), \quad \text{con } C_{cp}[t_p] = \tau(C_{cp}[t])\ \forall\ cp \in C_p \wedge C_c[t_p] = C_c[t]\ \forall\ c \notin C_p \ ; \quad t = t_p \wedge \text{Nevs} = \text{Nevs} \cup N_{cp}$$

En este caso, el autómata avanza tomando las celdas inminentes (C_p) de la lista de eventos, y calculando la función de transición para todas ellas. También se hace que el tiempo avance hasta la hora del último evento tratado (t_p). Finalmente, se agregan las celdas vecinas en la lista de eventos.

3.2 Modelos Cell-DEVS con demoras de transporte

En esta sección se presenta un paradigma de especificación utilizado para describir modelos celulares en forma de modelos atómicos DEVS. En este nuevo formalismo, que será llamado **Cell-DEVS**, cada celda será definida como un modelo atómico, y podrán utilizarse distintas clases de demoras al definir el comportamiento de cada celda. Las definiciones están basadas en una especificación formal previa para modelos con demoras de transporte ideales, incluida en [GG96].

En primer lugar, se incluirá una definición formal de distintos tipos de modelos atómicos modulares. Se considera la especificación de distintos formalismos: por un lado, modelos Cell-DEVS atómicos con demoras de transporte; por el otro, se incluyen modelos con demoras inerciales. Finalmente, se analizan modelos Cell-DEVS atómicos de tres estados con demoras, los que serán usados para definir una herramienta de simulación basada en este paradigma.

3.2.1 Especificación formal de modelos atómicos Cell-DEVS modulares con demoras de transporte.

Esta sección introduce la definición de modelos modulares Cell-DEVS con **demoras de transporte**. El comportamiento de estas demoras (usualmente empleadas en lenguajes de descripción de hardware) permite reflejar la propagación directa de señales en líneas de ancho de banda infinito.

Las celdas del modelo serán definidas como modelos atómicos, con el fin de acoplarlas con otras para formar un espacio de celdas completo. Se considerará que los modelos son cerrados, ya que cada celda sólo puede influenciar o ser influenciada por un vecino.

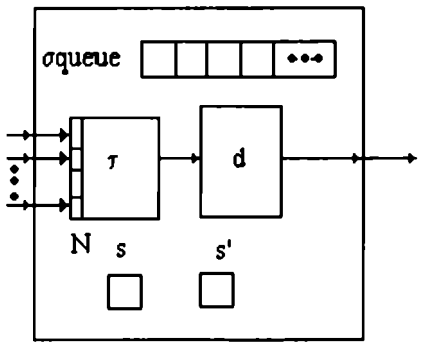


Figura 27. Descripción informal de una celda atómica con demoras de transporte.

La Figura 27 muestra informalmente los contenidos básicos de una celda. En este modelo atómico, una celda tiene η entradas en las que ocurren eventos externos. La celda memoriza los valores de todas las entradas, y cuando hay un nuevo evento externo, ejecuta la función booleana τ, que consume los valores de los eventos de entrada. Luego, el resultado del cálculo se demora durante **d** unidades de tiempo antes de ser transmitido a las celdas vecinas, planificando para ello un evento interno. Se debe usar una cola para mantener los valores de los resultados de los cálculos junto con su hora futura de planificación, ya que durante la demora pueden llegar nuevos eventos externos. Cuando llega la hora del evento interno, el valor se transmite por un port de salida.

Como los modelos influenciados sólo deben activarse cuando la celda influenciante cambia su estado, el resultado de la función de cálculo local sólo será transmitido si hay un cambio. Para permitir esta distinción, se guardan los valores presente y pasado de la celda.

Un modelo atómico como el descripto puede definirse formalmente como:

$$TDC = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

donde para $\#T < \infty \wedge T \in \{N, Z, R, \{0,1\}\}$;

$X \subseteq T$ es el conjunto de eventos externos de entrada;

$Y \subseteq T$ es el conjunto de eventos externos de salida;

$I = < \eta, \mu, P^X, P^Y >$ representa la definición de la interfaz modular del modelo. En este caso,
$\eta \in N, \eta < \infty$ es el tamaño de la vecindad,
$\mu \in N, \mu < \infty$ es la cantidad de ports de entrada/salida independientes de la vecindad, y
$\forall j \in [1, \eta], i \in \{X, Y\}, P_j^i$ es una definición de un port (de entrada o salida respectivamente), con
$P_j^i = \{ (N_j^i, T_j^i) / \forall j \in [1, \eta+\mu], N_j^i \in [i_1, i_{\eta+\mu}]$ (nombre del port), y $T_j^i \in I_i$ (Tipo del port)$\}$,
$I_i = \{ x / x \in X$ si $X \}$ ó $I_i = \{ x / x \in Y$ si $i = Y \}$;

$S \subseteq T$ incluye todos los valores posibles de estados secuenciales para la celda;

θ es la definición del estado de la celda, definido como

$\theta = \{$ (s, phase, σqueue, σ) /
$s \in S$ es el valor del estado para la celda,
phase ∈ {activa, pasiva},
$\sigma queue = \{ ((v_1,\sigma_1),...,(v_m,\sigma_m)) / m \in N \wedge m < \infty) \wedge \forall (i \in N, i \in [1,m]), v_i \in S \wedge \sigma_i \in R_0^+ \cup \infty \}$; y
$\sigma \in R_0^+ \cup \infty$
} ;

$\mathbf{N} \in S^{\eta+\mu}$, es el conjunto de estados de los eventos de entrada almacenados;

$\mathbf{d} \in \boldsymbol{R}_0^+$, $d < \infty$ es la demora de transporte de la celda;

δ_{int}: $\theta \rightarrow \theta$ es la función de transición interna;

δ_{ext}: $Q x X \rightarrow \theta$ es la función de transición externa, donde Q es el conjunto de estados definido como:

$$Q = \{ (s, e) / s \in \theta \times N \times d; e \in [0, D(s)] \};$$

τ: $N \rightarrow S$ es la función de cálculo local;

λ: $S \rightarrow Y$ es la función de salida; y

$\mathbf{D}$: $\theta \times N \times d \rightarrow \boldsymbol{R}_0^+ \cup \infty$, es la función de duración de vida del estado.

Esta definición es independiente de la técnica de simulación utilizada. Por ende, permite especificar el comportamiento del sistema independientemente de los detalles de implementación.

Cada celda tiene una interfaz bien definida, compuesta por un número fijo de ports numerados en orden ascendente. Por un lado, existe un conjunto de ports para establecer el acoplamiento interno del modelo de celdas, cada uno de los cuales estará conectado con un vecino. El número de estos ports de entrada y salida de cada celda será, por ende, igual al del tamaño de la vecindad. En el caso de precisarse otras entradas o salidas, se utilizarán los demás ports definidos.

Cada port en la interfaz tiene un nombre y un tipo. Los nombres están compuestos por un identificador (**X** para entradas; **Y** para salidas) y un número natural (número de port). Tanto los conjuntos de entrada y salida ($\boldsymbol{X}$ e $\boldsymbol{Y}$) como el conjunto de estados secuenciales, así como los tipos de los ports, pueden representar cualquier conjunto finito de símbolos, pero se ha elegido trabajar con un conjunto de tipos básicos con sus respectivas álgebras: $\boldsymbol{N}$, $\boldsymbol{Z}$, $\boldsymbol{R}$, y ***Booleanos***. Esta elección se debe a que la mayoría de las aplicaciones usan datos en estos dominios, y que muchos conjuntos de símbolos pueden mapearse en estos.

El estado de la celda se define como un conjunto compuesto por:

- El valor actual de la celda;
- La fase del modelo, que puede ser activa o pasiva.
- Una cola (σ**queue**) usada para mantener los tiempos de los próximos eventos y sus valores de entrada asociados; y
- La variable σ, que representa el tiempo simulado para el siguiente evento interno planificado;

El conjunto **N** representa el conjunto de valores de entrada de la celda, que tiene la forma de una η-upla $(s_1, ..., s_\eta)$ donde $s_i \in S$. Este conjunto registra los valores actuales usados para calcular el valor futuro a través de la función de cálculo local τ.

La función de duración de vida **D** se usa para controlar el tiempo de duración del estado de una celda. En este caso, **D**(s, phase, σqueue, σ, N, d) = t representa el tiempo durante el cual, si no hay eventos externos, el modelo atómico conservará el estado actual. Tanto la función de vida del estado como la demora de la celda tienen dominio en los números reales, ya que la base de tiempo seleccionada es continua.

Los objetivos de la ejecución de las funciones de transición (δ_{int}, δ_{ext}) y de la de salida (λ) son similares a los definidos para otros modelos DEVS. La función de transición interna se usa para definir cambios de estado debido a eventos internos, y la función de transición externa expresará la ocurrencia de eventos externos.

En cambio, la semántica de estas funciones será diferente a la de otros modelos DEVS. Esto se debe a que cada celda puede tener asociada una demora de transporte (**d**), que permite postergar la ejecución de la función de transición interna. La construcción de demora de transporte permite que el cambio de estado ante la ocurrencia de un evento externo sea demorado, y que el estado del sistema sólo cambie al consumirse el tiempo correspondiente a la demora.

Otra diferencia es que una vez ejecutada la función de transición interna, los modelos DEVS atómicos pasan a un estado pasivo hasta recibir nuevos eventos externos. En cambio, la introducción de demoras de transporte implica que una celda debe quedar activa durante la duración de la demora, ya que en ese lapso se pueden recibir nuevos eventos externos. Esto puede provocar que haya varios eventos internos planificados a futuro.

```
δext((s, phase, σqueue, σ, N, d), e, x) = (s, phase, σqueue, σ, N, d)
{
    /* Hay un evento de entrada: calcular τ, y si el nuevo valor es igual al anterior, no hacer nada */
    Actualizar los valores de N con el valor de x;
    s' = τ(N);          /* Calcula la función de cálculo local */

    if (s' != s) then          /* Los valores deben transmitirse sólo si cambia el estado */
        s = s';
        if (phase = pasiva) then    /* La celda está pasiva. Activarla. */
                phase = activa;
                σ = d;                      /* Demora de transporte */
        else
                if (d < σ) σ = d endif; /* Las demoras de una celda pueden ser variables, y podría
                        llegar un evento externo con menor demora que las ya almacenadas */
                for (ai ∈ σqueue) ai.σ = ai.σ - e;     /* La celda está activa. El tiempo transcurrido
                σ = σ - e;                      debe actualizar los valores futuros de σqueue */
        endif
        σqueue = insertar(σqueue, <s, d>); /* Inserción ordenada por σ */
    }
}

δint(s, phase, σqueue, σ) = (s, phase, σqueue, σ)
{       /* Los tiempos de σqueue deben actualizarse */
        for (ai ∈ σqueue) ai.σ = ai.σ - first(σqueue.σ);
        σqueue = tail (σqueue);

        if (empty(σqueue)) then /* No hay más eventos planificados */
                phase = pasiva;
                σ = ∞;
        else                    /* planificar el siguiente evento */
                phase = activa;
                σ = first(σqueue.σ);
        endif
}

λ(s) {
 return first(σqueue.value);
}
```

Figura 28. Definición de δ_{int}, δ_{ext} y λ para modelos TDC.

La semántica para la función de transición externa es la siguiente: la llegada de un evento externo indica que un vecino ha cambiado. Por ende, la celda debe activarse y calcular su función de cómputo local. La demora de transporte indica que se planificará la ejecución futura de la función de transición interna, para lo cual se almacenan los valores de la demora y de la entrada en la cola **σqueue**. Asimismo, si al hacer el cálculo el estado de la celda no cambia, sus vecinos tampoco pueden cambiar. Por ende, el estado actual no se encola para ser transmitido.

Una celda se pone en estado pasivo sólo cuando no tiene más eventos planificados, o sea, cuando la cola **σqueue** esté vacía. El valor de la variable σ será, por ende, igual al del primer valor de esta cola. Si la celda está activa, la llegada de un nuevo evento implica que, además de encolar el elemento, los valores de σ almacenados en la cola deben ser actualizados para reflejar el tiempo transcurrido (e).

La función de transición interna es función de la demora de la celda. Esta función (y la de salida) deben activarse cuando σ=0. Como la demora de transporte ha expirado, debe transmitirse el nuevo estado de la celda. La función de salida λ se ejecuta antes de la función δ_{int}, como en cualquier otra especificación DEVS, la que al activarse tomará el primer elemento de la cola de espera para ser transmitido. La función de transición interna sólo elimina el primer miembro de la cola (cuyo valor ya fue transmitido), y actualiza los tiempos de la cola de espera y de la variable de estado σ.

El comportamiento detallado de estas funciones puede verse en la Figura 28.

Como fuera mencionado, nos interesa la especialización en modelos celulares binarios de dos dimensiones. Estos modelos pueden definirse como:

$$TDC_b = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

con

$X = \{0, 1\}$;

$Y = \{0, 1\}$;

$I = <\eta, \mu, P^x, P^y>$ donde $(\eta \in N, n < \infty)$, $(\mu \in N, \mu < \infty)$, y $\forall j \in [1, \eta]$, $i \in \{X, Y\}$, P_j^i se define como:

$$P_j^i = \{ (N_j^i, T_j^i) \ / \ N_j^i \in [i_1, i_\eta], \text{ y } T_j^i \in \{0,1\} \ \};$$

$S = \{0, 1\}$;

$\theta = \{$ (s, phase, σqueue, σ) /
s $\in \{0,1\}$,
phase $\in$ {activa, pasiva},
σqueue = $\{ ((v_1,\sigma_1),...,(v_m,\sigma_m)) / (m \in N, m < \infty) \wedge \forall (i \in [1,m], i \in N), v_i \in \{0, 1\}, \quad \sigma_i \in R_0^+ \cup \infty, \}$, y
$\sigma \in R_0^+ \cup \infty$ $\}$;

$N \in \{0, 1\}^{\eta+\mu}$;

$d \in R_0^+$, $d < \infty$;

δ_{int}: $\theta \rightarrow \theta$;

δ_{ext}: $Q \times X \rightarrow \theta$, con $Q = \{ (s, e) / s \in \theta \times N \times d \wedge e \in [0, D(s)]\}$;

τ: $N \rightarrow S$;

λ: $S \rightarrow Y$; y

D: $\theta \times N \times d \rightarrow R_0^+ \cup \infty$.

Aquí vemos que, como las celdas definidas son binarias, los conjuntos de entrada y salida y el tipo de cada port de la celda también lo serán. En este caso, los valores de los eventos de entrada almacenados también son binarios, y la función de cálculo local es booleana.

También se considera que el espacio de celdas es cerrado, por ende, las celdas sólo se comunican con sus vecinos, y no es necesario expandir otros valores que no sean parte del estado de la celda. Esto provoca que sólo sea necesario usar los ports de conexión con la vecindad.

A continuación se mostrará que los modelos recién definidos son modelos DEVS. De aquí en más, las pruebas de las proposiciones sólo se enfocarán en los casos que nos interesan: espacios binarios bidimensionales. Las extensiones de las pruebas para espacios n-dimensionales en otros dominios son directas.

Lema 1
Los modelos TDC_b son modelos atómicos DEVS.

Prueba:

Como fuera analizado en la sección 2.2.1, un modelo atómico DEVS se define como:

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

I: interfaz del modelo atómico;
X: conjunto de eventos externos de entrada;
S: conjunto de estados secuenciales;
Y: conjunto de eventos externos generados para salida;
δ_{int}: S→S, función de transición interna;
δ_{ext}: QxX→S, función de transición externa Q = { (s,e) / s ∈ S, e ∈ [0, D(s)] };
λ: S→ Y, función de salida; y
D: S→R^+, función de duración de un estado.

y, por otro lado, consideremos

$$TDC_b = < X', Y', I', S', \theta', N', d', \delta_{int}', \delta_{ext}', \tau', \lambda', D' >$$

Se debe probar la equivalencia de los dos modelos. Por un lado, se tiene que:

I' ≡ I;
X' ≡ X'; y
Y' ≡ Y, por equivalencia directa de la definición de cada uno de los conjuntos.

Por otro lado, definamos a S = {0,1} x Sq x N' x d', con Sq = { (x, y) / x ∈ {0,1} ∧ y ∈ $R_0^+ \cup \infty$}, con #Sq < ∞;

Consideremos ahora, que todo modelo atómico DEVS tiene definidas dos variables de estado: fase y σ (como fuera analizado en la sección 2.2.1), y que en este caso σ puede calcularse como $\sigma = \min \{yi\}_{i=1}^{\#Sq}$ / (xi, yi) ∈ Sq. Entonces podemos decir que S ≡ θ' x N' x d', y por construcción de los conjuntos, S es un conjunto de estados secuenciales bien definido.

Por otro lado, δ_{int}': S → S porque θ' ⊆ S, y por definición, δ_{int}': θ' → θ'.

D': θ x N x d ' (≡ S) → $R_0^+ \cup \infty$

δ_{ext}': Q x X → S, ya que por definición de δ_{ext}': Q' x X' → θ', con Q' = { (s, e) / s ∈ θ' x N' x d' ∧ e ∈ [0, D'(s)]}, y por definición, Q' ⊆ Q, X' ≡ X ∧ θ' ⊆ S.

τ' es una función activada por δ_{ext}', por ende se la puede considerar como parte del comportamiento de la misma.

Finalmente,

λ': S → Y, porque Y' ≡ Y, θ' ⊆ S, y por definición, λ': θ' → Y.
Por ende, los modelos son equivalentes, y un TDC_b puede tratarse como un modelo atómico DEVS.

■

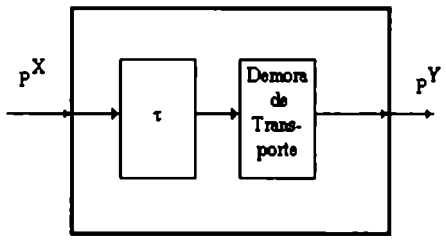


Figura 29. Función booleana con demora de transporte.

Para analizar el comportamiento de las funciones de transición para el caso de los modelos TDC_b, se presenta el ejemplo de la Figura 29. Esta celda (presentada previamente en [WFG97a] como una extensión del trabajo realizado en [GG96]) representa una función booleana ideal (la función identidad) con demoras de transporte.

Para este ejemplo consideremos una demora de transporte de 17 unidades de tiempo. En este caso se obtiene:

$I = <1, 0, P^X, P^Y>$,

donde $\eta = 1$, $\mu = 0$, $P^X = \{X1\}$, $P^Y = \{Y1\}$;

$X = Y = \{0,1\}$;
$d = 17$;
y $\tau(x) = x$.

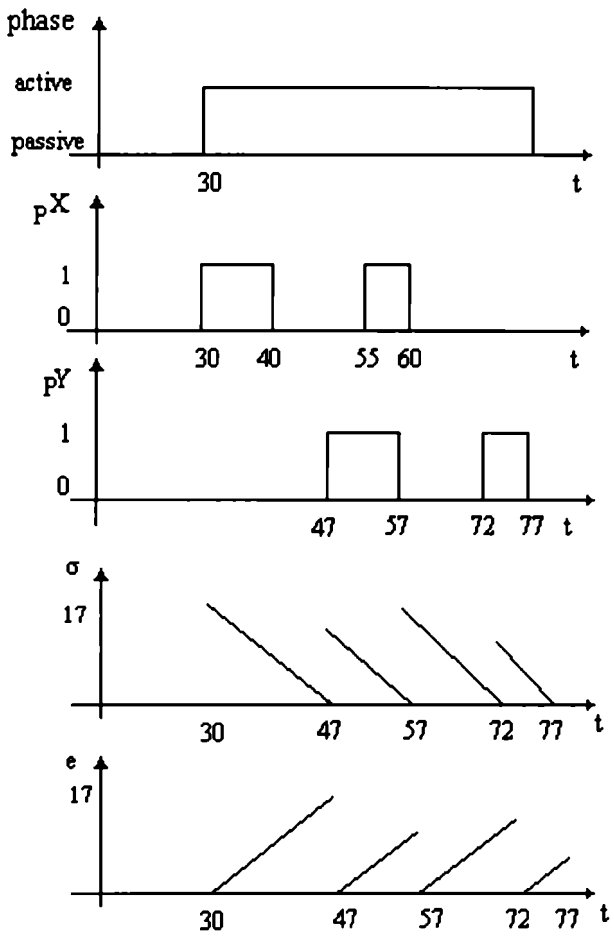


Figura 30. Trayectorias de entrada/salida con demoras de transporte para el modelo anterior.

Supongamos que el modelo recibe eventos externos de entrada con los valores que se muestran en la trayectoria para P^x en la Figura 29. Analizando la ejecución de las distintas funciones para el modelo, dichas entradas provocan el comportamiento que se observa en las demás trayectorias que se ven en la misma figura.

El comportamiento detallado de las variables de estado de la celda (para estas trayectorias) se analiza en la Tabla 1. La tabla muestra las variables de estados de la celda en cada transición (incluyendo su hora de activación). El signo * identifica la ejecución de funciones de transición interna, mientras que las líneas restantes representan la ejecución de transiciones externas.

Los estados presente y futuro se incluyen en las columnas marcadas con s y s' respectivamente. Los campos que contienen dos valores separados por una barra representan los valores de las variables antes y después de la ejecución de cada función de transición, Finalmente, los valores de salida del modelo serán obtenidos tomando el valor de la variable de s en las líneas correspondientes a las funciones de transición interna.

	t	s	s'	phase	σ	e	σqueue
	...	0	0	pasiva			
	30	0/1	1	activa	17	0	(1,17)
	40	1/0	0	activa	7	10	(1,7),(0,17)
*	47	0	0	activa	0 / 10	17 / 0	(0,10)
	55	0/1	1	activa	2	8	(0,2), (1,17)
*	57	1	1	activa	0 / 15	10 / 0	(1,15)
	60	1/0	1/0	activa	12	3	(1,12), (0,17)
*	72	0	0	activa	0 / 5	15	(0,5)
*	77	0	0	pasiva	∞	5	

Tabla 1. Secuencia de ejecución de un modelo celular con demoras de transporte.

3.2.2 Modelos celulares con demoras inerciales

En muchos casos, la introducción de **demoras inerciales** [GM76] permite el modelado natural de ciertos fenómenos cuya semántica incluye comportamiento con desalojo. La construcción descarta las entradas a una celda si su valor no se mantiene durante un período determinado, y por otro lado, si el flujo de entrada es constante durante ese tiempo (llamado demora inercial), el valor de entrada se registra.

Un paradigma de simulación que tenga la facilidad de representar esta clase de comportamiento, facilita la especificación de los sistemas a modelar y la construcción de los modelos ejecutables correspondientes. Esta construcción se ha usado, por ejemplo, en el dominio del modelado de circuitos digitales, ya que puede representar el comportamiento de componentes que cambian su estado interno dependiendo no sólo de los valores de tensión de entrada, sino de su duración.

Además del dominio del diseño de circuitos, la construcción puede ser útil para representar otros modelos físicos espaciales. Consideremos, por ejemplo, el estudio de la expansión del fuego en un bosque. En este caso, el fuego podría ser representado como un valor para una celda, que debería ser transmitido a las celdas vecinas. El fuego se expande a las vecinas sólo si el valor se mantiene durante la demora inercial (permitiendo representar, por ejemplo, la influencia del viento, lluvia o bomberos trabajando). La construcción también puede usarse para estudiar otros sistemas ecológicos. Por ejemplo, si se analiza un campo de maíz donde si las temperaturas son bajas durante cierto tiempo, las plantas se pueden destruir. En cambio, si antes de la demora inercial aumenta la temperatura, los brotes siguen vivos.

Otro ejemplo lo podemos encontrar en simulaciones de tráfico (como será detallado en la sección 5.1.2). En estos casos se puede usar una demora inercial para representar el cruce de una bocacalle. Si un automóvil está detenido durante determinado tiempo en una bocacalle, cruza. Si el tiempo transcurrido es menor y aparece otro automotor en la calle de la derecha (o, por ejemplo, decide doblar), el automóvil no avanza.

En estos todos estos casos, el uso de construcciones de demora inercial permite modelar comportamientos complejos de forma simple. La Figura 31 ejemplifica el comportamiento de una demora inercial para una celda atómica como la considerada en la sección 3.2.1.

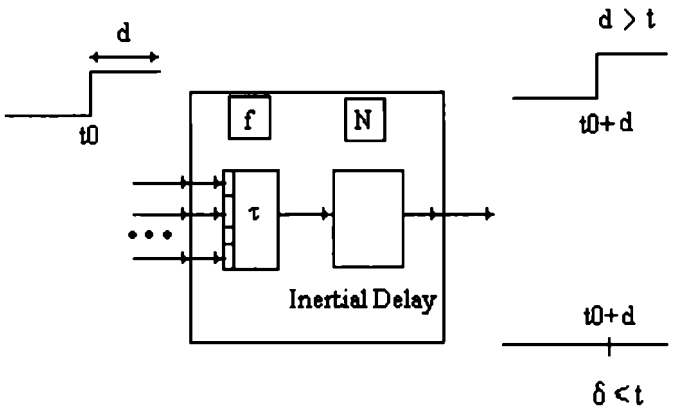


Figura 31. Comportamiento de un modelo atómico de celdas con demora inercial.

Un modelo atómico de celda con demoras inerciales puede especificarse como:

$$IDC = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

donde para $\#T < \infty \wedge T \in \{N, Z, R, \{0,1\}\}$;

$X \subseteq T$ es el conjunto de eventos externos de entrada;

$Y \subseteq T$ es el conjunto de eventos externos de salida;

$I = < \eta, \mu, P^X, P^Y >$ representa la definición de la interfaz modular del modelo. En este caso,
$\eta \in N, \eta < \infty$ es el tamaño de la vecindad,
$\mu \in N, \mu < \infty$ es la cantidad de ports de entrada/salida independientes de la vecindad, y
$\forall j \in [1, \eta], i \in \{X, Y\}, P_j^i$ es una definición de un port (de entrada o salida respectivamente),
$P_j^i = \{ (N_j^i, T_j^i) / \forall j \in [1, \eta+\mu], N_j^i \in [i_1, i_{\eta+\mu}]$ (nombre del port), y $T_j^i \in I_i$ (Tipo del port)$\}$,
$I_i = \{ x / x \in X$ si $i = X \}$ ó $I_i = \{ x / x \in Y$ si $i = Y \}$;

$S \subseteq T$ incluye todos los valores posibles de estados secuenciales para la celda;

$\theta = \{ (s, phase, f, \sigma) / s \in S$, phase $\in$ {activa, pasiva}, $f \in T$, y $\sigma \in R_0^+ \cup \infty \}$ es la definición del estado de la celda;

$N \in S^{\eta+\mu}$, es el conjunto de estados de los eventos de entrada almacenados;

$d \in R_0^+, d < \infty$ es la demora inercial de la celda;

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

$\delta_{ext}: Q \times X \rightarrow \theta$ es la función de transición externa, donde Q es el conjunto de estados definido como:

$$Q = \{ (s, e) / s \in \theta \times N \times d \wedge e \in [0, D(s)] \};$$

$\tau: N \rightarrow S$ es la función de cálculo local;

$\lambda: S \rightarrow Y$ es la función de salida; y

$D: \theta \times N \times d \rightarrow R_0^+ \cup \infty$, es la función de duración de vida del estado.

Como puede verse, la mayoría de los conjuntos y funciones fueron definidos de forma similar a los presentados para modelos con demoras de transporte, pero existen algunas diferencias. Por un lado, la definición para el estado de una celda ha sido modificada. En este caso, s, phase y σ tienen el mismo significado definido anteriormente, pero f representa el valor futuro factible para la celda. Si el valor de entrada de la celda se mantiene durante la demora inercial, f se convertirá en el estado actual de la celda.

La definición de la variable **d** corresponde ahora a una demora inercial, lo que motiva un cambio en la semántica de las funciones de transición, que ahora deben modelar demoras inerciales. Para este caso, el comportamiento de las funciones de transición se ha definido como sigue:

```
δext((s, phase, f, N, d, σ), e, x) = (s, phase, f, N, d, σ)
{
        Actualizar los valores de los eventos de entrada con el valor de x;
        s' = τ(N);

        if (s != s') then
          s= s';              /* El nuevo estado es el recién calculado */

            if (phase = pasiva) then
                phase = activa;
                σ = d;   /* Demora inercial */
            else
                σ = σ - e;
                if (σ > 0 .AND. f != s)  σ = d;  /* Remoción: el estado calculado es distinto al futuro */
            endif
        endif
        f = s;     /* El estado futuro es el recién obtenido */
}

δint(s, phase, f, σ) = (s, phase, f, σ)
{
        if (σ = 0) then
                phase = pasiva;
                σ = ∞;
        endif
}

λ(s) {
        return s;
}
```

Figura 32. Funciones de transición interna y externa para modelos con demoras inerciales.

Podemos ver que el objetivo de la función de transición externa es almacenar el valor actual para la celda, y detectar si la entrada se conserva durante la demora inercial. Si no es así, la entrada anterior es removida. Al llegar la hora planificada por la demora, se transmite el valor actual de la celda.

Para la especialización en modelos celulares binarios de dos dimensiones, los conjuntos pueden definirse como:

$$IDC_b = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

con

$X = \{0, 1\}$;

$Y = \{0, 1\}$;

$\mathbf{I} = < \eta, \mu, P^X, P^Y >$ donde ($\eta \in N, \eta < \infty$), ($\mu \in N, \mu < \infty$), y $\forall\ j \in [1, \eta], i \in \{X, Y\}$, P_j^i se define como:

$$P_j^i = \{ (N_j^i, T_j^i) \ / \ N_j^i \in [i_1, i_\eta], \text{ y } T_j^i \in \{0,1\} \ \};$$

$\mathbf{S} = \{0, 1\}$;

$\theta = \{ (s, \text{phase}, f, \sigma) \ / s \in \{0,1\}, \text{phase} \in \{\text{activa, pasiva}\}, f \in \{0,1\}, \text{y } \sigma \in \boldsymbol{R}_0^+ \cup \infty \}$;

$N \in \{0, 1\}^{\eta+\mu}$;

$\mathbf{d} \in \boldsymbol{R}_0^+, d < \infty$;

δ_{int}: $S \rightarrow S$;

δ_{ext}: $Q x X \rightarrow \theta$, con $Q = \{ (s, e) / s \in \theta \text{ x } N \text{ x } d \ \wedge e \in [0, D(s)]\}$;

τ: $N \rightarrow S$;

λ: $S \rightarrow Y$; y

$\mathbf{D}$: $\theta \text{ x } N \text{ x } d \rightarrow \boldsymbol{R}_0^+ \cup \infty$.

Lema 2
Los modelos IDC_b son modelos atómicos DEVS.

Prueba:

La equivalencia entre los dos tipos de modelos puede probarse como en el Lema 1. Para ello debe probarse que:

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

es equivalente a

$$IDC_b = < X', Y', I', S', \theta', N', d', \delta_{int}', \delta_{ext}', \ ', \lambda', D' >$$

Por un lado, se tiene que:

$I' \equiv I$;
$X' \equiv X$; y
$Y' \equiv Y$, por definición de cada uno de los conjuntos.

Por otro lado, si se define $S = \{0,1\} \text{ x } f \text{ x } N \text{ x } d$, y se considera que todo modelo DEVS tiene definidas dos variables de estado, fase y σ (como fuera analizado en la sección 2.2.1), entonces se puede decir que $S \equiv \theta' \text{ x } N' \text{ x } d'$, y por construcción de los conjuntos, S es un conjunto de estados secuenciales bien definido.

Por otro lado, δ_{int}': $S \rightarrow S$; D': $S \rightarrow \boldsymbol{R}_0^+ \cup \infty$ y δ_{ext}': $Q \text{ x } X \rightarrow S$, con los mismos argumentos que en el Lema 1.

τ' es una función activada por δ_{ext}', por ende se la puede considerar como parte del comportamiento de la misma. Finalmente, λ': $S \rightarrow Y$. Por ende, los modelos son equivalentes, y un IDC_b es un modelo atómico DEVS.

■

El comportamiento de este tipo de celdas atómicas puede estudiarse en el siguiente ejemplo. Se consideran las trayectorias de entrada/salida presentadas en la Figura 33 (basadas, a su vez, en la misma celda atómica usada en la sección 3.2.1), usando una demora inercial de 5 unidades de tiempo.

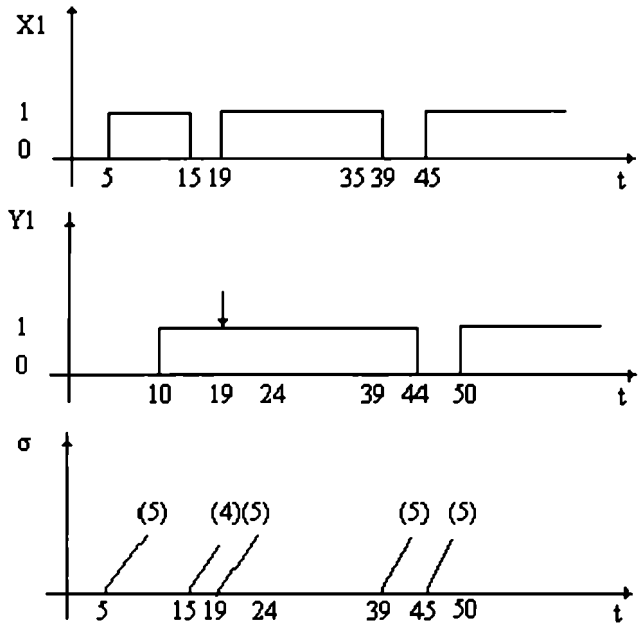


Figura 33. Trayectorias de entrada y salida con demoras inerciales.

El flujo de ejecución de las funciones puede analizarse fácilmente en la Tabla 2. Aquí la representación es la misma utilizada previamente en la Tabla 1. Las líneas marcadas con los signos "!" (y con flechas en la figura) representan el comportamiento del modelo ante desalojos.

	t	s	s'	phase	σ	e	x	f
	...	0	0	pasiva	∞			
	5	0/1	1	activa	5	0	1	1
*	10	1	1	pasiva	0/∞	5		
	15	1/0	0	activa	5	0	0	0
!	19	0/1	1	activa	1/5	4/0	1	0/1
*	24	1	1	pasiva	0/∞			
	39	1/0	0	activa	5	0	0	1/0
*	44	0	0	pasiva	0/∞			
	45	0/1	1	activa	5	0	1	0/1
*	50	1	1	pasiva	0/∞			

Tabla 2 - Secuencia de ejecución de la Figura 33.

El comportamiento de la demora inercial es tal que si hay una nueva entrada antes de que se consuma la demora, el último evento futuro que haya llegado puede ser removido. Esto sólo ocurrirá si el nuevo valor externo es distinto del valor almacenado previamente. En cambio, si ambos valores son iguales, la llegada del nuevo evento pasa inadvertida para el modelo, ya que su llegada equivale a tener un único evento externo (el que llegó primero) pero con una demora inercial mayor.

Un ejemplo de este comportamiento puede verse en la tabla anterior, en el tiempo simulado 24. En el instante 19, había llegado un nuevo evento externo con valor 1. Como el estado del sistema en ese momento era 1 (debido a que el evento externo previo fue removido), el estado de la celda no cambiará.

3.2.3 Modelos atómicos Cell-DEVS con tres estados

En las secciones anteriores se describieron formalmente la interfaz y comportamiento de modelos Cell-DEVS con distintas demoras. Como fue mencionado, se hizo especialización en modelos binarios, que son los de principal interés en este trabajo. Para este caso suele ser útil la introducción de un tercer estado reflejando un comportamiento indefinido para la celda. Esto ocurre porque los valores de demora suelen no ser conocidos con antelación, y sólo puede ser determinado con precisión un rango de valores de demora. Con dicho fin se hizo una especialización de modelos atómicos Cell-DEVS con tres estados, cuya finalidad es proveer un marco formal especializado para la construcción de la herramienta que se presentará en la sección 6.3.

Los modelos atómicos Cell-DEVS de tres estados (con distintos tipos de demoras) pueden describirse formalmente como:

$$UDC = < X, Y, I, delay, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

donde

$X = \{0, 1, \phi\}$ es el conjunto de eventos externos de entrada;

$Y = \{0, 1, \phi\}$ es el conjunto de eventos externos de salida;

$I = < \eta, \mu, P^x, P^y >$ donde $(\eta \in N, \eta < \infty)$, $(\mu \in N, \mu < \infty)$, y para $i \in \{X, Y\}$, $\forall\ j \in [1, \eta]$, P_j^i se define como:

$$P_j^i = \{ (N_j^i, T_j^i) \ / \ N_j^i \in [i_1, i_\eta], \text{ y } T_j^i \in \{0, 1, \phi\} \ \};$$

delay $\in$ {transporte, inercial, ninguno}.

$S = \{0, 1, \phi\}$ so n los valores secuenciales para la celda;

θ es la definición del estado de la celda, con
$\theta = \{$ (s, phase, σqueue, σ) /
$s \in \{0, 1, \phi\}$,
phase $\in$ {activa, pasiva},
$\sigma queue = \{ ((v_1,\sigma_1),...,(v_m,\sigma_m)) / m \in N \wedge \forall i \in [1,m], i \in N, \sigma_i \in R_0^+ \cup \infty, vi \in \{0, 1, \phi\} \}$, y
$\sigma \in R_0^+ \cup \infty$
$\}$ si delay = transporte, o

$\theta = \{ (s, phase, f, \sigma) / s \in \{0, 1, \phi\}$, phase $\in$ {activa, pasiva}, $f \in \{0, 1, \phi\}$, y $\sigma \in R_0^+ \cup \infty\}$ si delay = inercial;

$N \in \{0, 1, \phi\}^{\eta+\mu}$, es el conjunto de los estados binarios de entrada

$d \in R_0^+$, $d < \infty$ es la demora de la celda;

δ_{int}: $\theta \rightarrow \theta$ es la función de transición interna;

δ_{ext}: $Q \times X \rightarrow \theta$ es la función de transición externa, donde Q es el conjunto de estados definido por

$$Q = \{ (s, e) / s \in \theta \times N \times d \ \wedge e \in [0, D(s)] \};$$

τ: $N \rightarrow S$ es la función de cálculo local;

λ: $S \rightarrow Y$ es la función de salida; y

D: $\theta \times N \times d \rightarrow R_0^+ \cup \infty$, es la función de duración del estado.

La definición es similar a la de las secciones previas, pero incluyendo el tercer estado. Este estado se denotará como ϕ, y servirá para definir el comportamiento de aquellas celdas en las cuales el estado binario sea desconocido. Los cambios principales a la descripción están en los conjuntos de estado, de entrada y de salida, donde se agregó el estado desconocido. De acuerdo con la demora elegida, serán activadas las funciones de las secciones 3.2.1 o 3.2.2.

Consideremos ahora el comportamiento de la compuerta ideal del ejemplo presentado en la Figura 34. Supongamos una demora de transporte de 15 unidades de tiempo. La ejecución de las funciones de transición y el comportamiento de las trayectorias de entrada/salida puede verse en la siguiente figura:

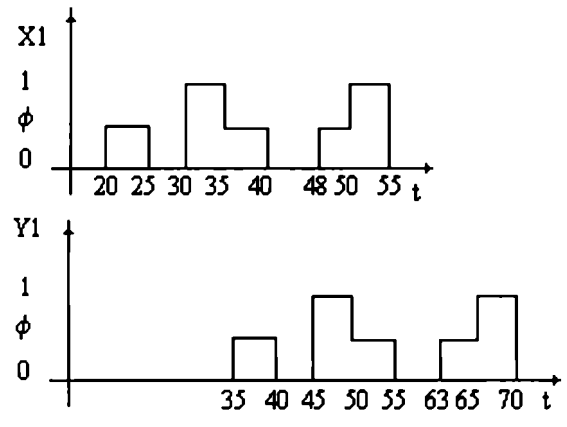


Figura 34. Trayectorias de entrada/salida para celdas de tres estados con demoras de transporte.

	t	s	s'	phase	σ	e	σqueue
	...	0	0	pasiva			
	20	0/ϕ	ϕ	activa	15	0	(15,ϕ)
	25	ϕ/0	0	activa	10	5	(10,ϕ), (15,0)
	30	0/1	1	activa	5	10	(5,ϕ), (10,0), (15, 1)
*	35	1	1	activa	0/5	15/0	(5,0), (10, 1)
	35	1/ϕ	ϕ	activa	5	0	(5,0), (10, 1), (15,ϕ)
*	40	ϕ	ϕ	activa	0/5	5/0	(5, 1), (10,ϕ)
	40	ϕ/0	0	activa	5	0	(5, ϕ), (10,ϕ), (15,0)
*	45	0	0	activa	0/5	5/0	(5,ϕ), (10,0)
	48	0/ϕ	ϕ	activa	2	3	(2,ϕ), (7,0), (15,ϕ)
*	50	ϕ	ϕ	activa	0/5	5/0	(5,0), (13,ϕ)
	50	ϕ/1	1	activa	5	0	(5,0), (13,ϕ), (15,1)
*	55	1	1	activa	0/5	5/0	(8, ϕ), (10,1)
	55	1/0	0	activa	8	0	(8, ϕ), (10, 1), (15, 0)
*	63	0	0	activa	0/5	8/0	(2, 1), (7, 0)
*	65	0	0	activa	0/5	5/0	(5,0)
*	70	0	0	pasiva	∞		

Tabla 3. Secuencia de ejecución para el modelo de celda de tres estados con demora de transporte.

3.3 Especificación formal de modelos acoplados Cell-DEVS cerrados con demoras de transporte.

Los modelos atómicos de celdas con distintas demoras de transporte presentados en las secciones anteriores pueden acoplarse con otros modelos para formar un modelo multicomponente basado en celdas. El objetivo de esta sección es el especificar una descripción formal para modelos Cell-DEVS acoplados, que serán definidos como un espacio consistente de celdas atómicas conectadas por una relación de vecindad, donde cada componente será una celda como las definidas en las secciones anteriores. En esta especificación se considerarán modelos cerrados (es decir, que no

pueden acoplarse con otros modelos de base). Por ende, al no usarse entradas ni salidas, tampoco habrá necesidad de definir una interfaz para el modelo.

Un modelo Cell-DEVS acopladas puede definirse como:

$$CC = < n, \{t_1,..., t_n\}, \eta, N, C, B, Z, select >$$

donde

n es la dimensión del espacio de celdas;

$\{t_1,...,t_n\}$ es la cantidad de celdas en cada una de las dimensiones;

η es el tamaño de la vecindad;

N es el conjunto de vecindad;

C es el espacio de celdas;

B es el conjunto de celdas del borde;

Z es la función de traducción; y

select es la función de selección ante eventos simultáneos.

En este caso,

$\mathbf{n} \in N, n < \infty$;

$\{t_1,...,t_n\} \in N$, con $t_k < \infty \; \forall \; k \in [1,n]$;

$$\eta \in N, \eta \leq \prod_{i=1}^{n} t_i;$$

$$\mathbf{N} = \{ (v_{k1},...,v_{kn}) / \forall (k \in N, k \in [1, \eta] \wedge i \in N, i \in [1, n]), v_{ki} \in \mathbf{Z} \wedge (t_i - | v_{ki} | \geq 0) \};$$

$\mathbf{C} = \{ C_c / c \in \mathbf{I} \wedge C_c = < I_c, X_c, Y_c, S_c, N_c, d_c, \delta_{int_c}, \delta ext_c, \tau_c, \lambda_c, D_c >$ es un componente TDC, IDC ó UDC $\}$, siendo

$$\mathbf{I} = \{ (i_1,...,i_n) / (i_k \in N \wedge i_k \in [1, t_k]) \; \forall \; k \in [1, n] \} \qquad (4)$$

$\mathbf{B} = \{\varnothing\}$ si el espacio de celdas es toroidal; o
$\mathbf{B} = \{C_b / C_b \in C$ con $b \in \mathbf{I} \}$, con **I** definido como en (4). En este caso, el conjunto B está sujeto a la restricción que $\tau_b \neq \tau_c = \tau \; \forall (C_c \notin B) \wedge (C_b \in B)$.

$B = \{C_b / C_b \in C$ con $b \in L\}$, siendo $L = \{ (i_1,...,i_n) / i_j = 0 \vee i_j = t_j \; \forall \; j \in [1, n] \}$, y sujeto a que $\tau_b \neq \tau_c = \tau \; \forall \; c \notin L$.

Definición

Si llamamos **C** a la posición de una celda, donde $\mathbf{C} = (i_1,...,i_n)$; y sea $V \in \mathbf{Z}^n$ la n-upla definida como $V = (v_1,..., v_n)$, con $v_i \in \mathbf{Z}$, $| v_i | \leq t_i$ entonces se define como

$$D = C +_t V \qquad (5)$$

a la n-upla $D = (j_1,...,j_n)$ compuesta por: $j_k = (i_k + v_k) \bmod (t_k) \; \forall \; k \in [1, n]$.

Notación 2

$\forall k \in N, k \in [1, \eta]$, se denotará $V_k = \{ (v_{k1},...,v_{kn}) / (v_{k1},...,v_{kn}) \in N \}$ al k-ésimo elemento de una lista de vecindad.

Notación 3

Se denotará P_c^{iq} al port $P_q^i \in I_c$, con $i \in \{X, Y\}$, y $(q \in N, q \in [1, \eta])$, donde $I_c \subseteq C_c$, y C_c es un componente TDC, IDC ó UDC.

Z: $I_C \rightarrow I_D$ es la función de traducción, definida (usando (5)) como:

$Z(P_C{}^{Y_q}) = P_D{}^{X_q}$, $\forall (q \in N, q \in [1,\eta])$, donde $\forall V_k \in N$, $D = C +_t V_k$; y

$Z(P_C{}^{X_q}) = P_D{}^{Y_q}$, $\forall (q \in N, q \in [1,\eta])$, donde $\forall V_k \in N$, $D = C -_t V_k$.

select = $\{ (s_1, ..., s_n) / s_i \in N, \forall i \in [0, n] \}$, con la restricción que select $\subseteq t_1xt_2x...xt_n \rightarrow t_1xt_2x...xt_n$.

En este caso, el espacio de celdas **C** es un modelo acoplado definido como un arreglo de modelos atómicos Cell-DEVS de tamaño fijo (t_1 x ... x t_n).

Cada celda tiene un conjunto de η vecinas, definidas por el conjunto de vecindad (**N**). Este conjunto está representado como una lista de tuplas de dimensión n que definen la posición relativa entre la celda origen y sus vecinos. Estos índices no pueden exceder el límite del espacio de celdas.

El conjunto **B** define el borde del espacio de celdas, y puede ser de dos tipos distintos. Si $B = \{\varnothing\}$, toda celda en el espacio tendrá el mismo comportamiento: las celdas en un borde se conectan con las que están el borde opuesto usando la relación de vecindad inversa. En cambio, si el conjunto de borde es no vacío, las celdas tendrán un comportamiento diferente a las otras del modelo (por ejemplo, pueden generar su estado, o consumir los estados de los vecinos).

La función **Z** permite definir el acoplamiento entre celdas en el modelo. Esta función traduce las salidas del q-ésimo port de salida en la celda C_c en valores para el q-ésimo port de entrada de la celda C_D (y viceversa). Cada port de salida corresponderá a un vecino, y cada port de entrada estará asociado con una celda en la vecindad inversa [Zei76].

Los nombres de los ports se generan usando la siguiente notación: $P_C{}^{X_q}$ se refiere al q-ésimo port de entrada de la celda C_c, y $P_C{}^{Y_q}$ al q-ésimo port de salida. Estos ports corresponden con los nombres de ports denotados como X_q o Y_q para cada celda, como fue mencionado en la sección 3.2.1. El número de la celda con la cual debe acoplarse el modelo será generado sumando los números de la lista de vecinos al número de la celda actual. El primer port de salida de una celda estará conectado con el primer port de entrada del vecino, de acuerdo con el orden de la lista de vecinos.

Para la especialización del formalismo en el caso de modelos binarios bidimensionales [WG98], se obtiene:

$$CC_b = < \{f, c\}, \eta, N, C, B, Z, select >$$

donde

$\{f, c\} \in N$;

$\eta \in N, \eta \leq f \cdot c$;

$N = \{ (i_p,j_p) / (i_p, j_p \in Z \wedge i_p, j_p \in [-1, 1]) \ \forall (p \in N, p \in [1,\eta]) \wedge (0 < i_p \leq f-1 \wedge 0 < j_p \leq c-1)\}$.

$C = \{C_{ij} / i \in [1,f] \wedge j \in [1,c]\}$, donde

$$C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij}>$$

es un componente TDC, IDC ó UDC;

B = {∅} si el espacio de celdas es toroidal; o

B = {C_{ij} / (i = 1 ∨ i = f) ∧ (j = 1 ∨ j = c) ∧ C_{ij} ∈ C ∧ C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, δ_{intij}, δ_{extij}, τ_{ij}, λ_{ij}, D_{ij}> es un componente TDC, IDC ó UDC }, si las celdas atómicas del borde tienen distinto comportamiento que el resto del espacio de celdas.

Notación 4

Se denotará P_{ij}^{iq} al port $P_q^i \in I_{ij}$, con $i \in \{X, Y\}$, $q \in N$, $q \in [1, \eta]$.

Z: $I_{ij} \rightarrow I_{kl}$, definida como

$Z(P_{ij}^{Yq}) = P_{kl}^{Xq}$, con $q \in N$, $q \in [1,\eta] \wedge \forall (r,s) \in N$, $k = (i+r) \bmod f$; $l = (j+s) \bmod c$; y
$Z(P_{ij}^{Xq}) = P_{kl}^{Yq}$, con $q \in N$, $q \in [1,\eta] \wedge \forall (r,s) \in N$, $k = (i-r) \bmod f$; $l = (j-s) \bmod c$.

select = { (k,l) / (k,l) ∈ N } es la función de selección ante eventos simultáneos, con la restricción de que select ⊆ f x c → f x c.

Lema 3

Los modelos CC_b son modelos acoplados DEVS.

Prueba:

Un modelo DEVS acoplado está definido como:

$$CM = < X, Y, I, D, \{M_d\}, \{I_d\}, \{Z_{dj}\}, \text{select} >$$

donde

X es el conjunto de eventos externos de entrada;
Y es el conjunto de eventos externos de salida;
I representa la definición de la interfaz modular del modelo;
D ∈ *N* es un índice de los componentes del modelo acoplado, y ∀d ∈ D,
M_d es un modelo básico DEVS,
I_d es el conjunto de modelos influenciados por el modelo d, y ∀ j ∈ I_d,
Z_{dj}: $Y_d \rightarrow X_j$ es la función de traducción de d a j; y
select es el selector ante eventos simultáneos.

CC_b es un modelo acoplado si $CC_b \equiv CM$, con

$$CC_b = < \{f, c\}, \eta, N, C, B, Z, \text{select} >$$

En este caso, podemos considerar la siguiente definición de conjuntos para CM:

I = { ∅ };
X = { ∅ };
Y = { ∅ };
D ∈ *N*, D ∈ [1, f x c].

∀ d ∈ D, sea una función de mapeo de (i,j) a d, definida por

F: D x D → D de forma tal que *F*(i,j) = d. Por ejemplo, podría usarse la función *F*(i,j) = (i-1).c + (j-1).

Ahora, consideremos que $M_d = C_{F(i,j)}$ $\forall$ i $\in$ [1, f], j $\in$ [1, c]. Por definición de la función de mapeo, y por el Lema 1, M_d es un modelo atómico DEVS.

Por la definición de B y la definición de M_d recién presentada, se tiene que $B \subseteq M_d \vee B = \{ \varnothing \}$.

Sea $I_d = \{ x / x = F(i+n_1, j+n_2) \forall (n_1, n_2) \in N\}$, con $d = F(i,j)$. Luego, por definición de N, $I_d \subseteq D$. Comprobemos esta afirmación para uno de los extermos. Sea i = f-1, j = c-1. Luego,

$F(i+n1, j+n2) = (i-1+n1).c + (j-1+n2)$

Ahora, por definición de N, tenemos que

$(i-1+n1).c + (j-1+n2) \leq (f-2+1).c + (c-2+1) = (f-1).c + c = f.c$

Por lo tanto, $F(i+n1, j+n2) \in D$. El mismo razonamiento puede repetirse para todas las celdas del espacio.

Ahora, $\forall d \in I_d$, $Z_{dj} \equiv Z \Leftrightarrow Z: Y_d \rightarrow X_j$. Por definición de los modelos CC_b presentada previamente, se sabe que que Z: $I_{ij} \rightarrow I_{pq}$. Si usamos la misma función de mapeo, (i,j) mapea en I_d por definición de I_d. Sea $F(p,q)$ el mismo mapeo, definido como $F(p,q) = (p-1).c + (q-1)$. Aplicando la definición de la función Z, se obtiene que $F(p,q) = [(p+r) \bmod f - 1].c + [(q+s) \bmod c -1]= A. c + B$, donde $A \in [0, f-1] \wedge B \in [0, c-1] \Rightarrow F(p,q) \subseteq I_d$.

Finalmente, usando la misma función de mapeo, se obtiene que select: $D \rightarrow D$.

Por ende, ambos modelos son equivalentes, y vale la proposición.

■

En la Figura 35 se muestra un ejemplo del procedimiento de acoplamiento en espacios de celdas bidimensionales. La Figura 35(a) muestra la vecindad de la celda (i,j) y su representación usando la lista de vecinos. La Figura 35(b) muestra cómo el primer port de salida de la celda (i,j) se conecta con el primer port de entrada del primer vecino en la lista; el segundo port con el segundo vecino, etc. En cambio, para los ports de entrada la conexión se hace a través de la lista de vecindad inversa. Para cada par (i,j) en la vecindad, el par (-i, -j) debe ser incluido en la lista.

La definición presentada sólo incluye permite vecindades regulares y vecinos en celdas adyacentes. Si fueran necesarios otros tipos de vecindades (incluyendo distintas vecindades para cada celda), la definición puede extenderse. Aquí,

$$N = \{N_c / c \in \mathbf{I} \} \text{ con } \mathbf{I} \text{ definido como en (4)}$$

donde

$$N_c = \{ (v_{k1},...,v_{kn})_c / \forall (k \in N, k \in [1, \eta_c]) \wedge (i \in N, i \in [1, n]), v_{ki} \in Z \wedge (ti - | v_{ki} | \geq 0) \};$$

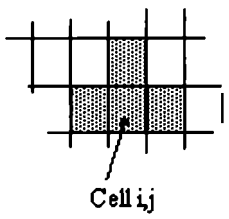
En este caso, si suponemos el caso de espacios bidimensionales, N se define como un arreglo de listas de vecindades, especificado formalmente como:

$$N = \{N_{kl} / k \in [1,f], l \in [1,c] \}$$

donde

$$N_{kl} = \{(i_p,j_p) / \forall(p \in N, p \in [1,\eta_{kl}]), i_p, j_p \in Z; | k - i_p | \geq 0 \wedge | l - j_p | \geq 0 \wedge \eta_{kl} \in N \}$$

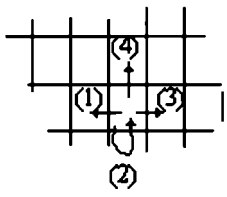
En este caso, cada celda tendrá una lista de vecinos de η_c (η_{kl}) elementos, compuesta por tuplas de índices relativos a la celda de origen. Estos índices no pueden exceder el límite del espacio de celdas.



Lista de vecinos:
{ (0, -1), (0,0), (0,1), (-1,0)}

Nota: -1: izquierda, arriba;
1: derecha, abajo.

(a)



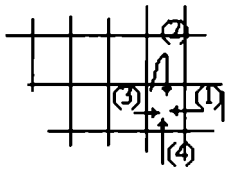
$P_{ij}\ Y_1 \rightarrow P_{i,j-1}\ X_1$ (1)

$P_{ij}\ Y_2 \rightarrow P_{ij}\ X_2$ (2)

$P_{ij}\ Y_3 \rightarrow P_{i,j+1}\ X_3$ (3)

$P_{ij}\ Y_4 \rightarrow P_{i-1,j}\ X_4$ (4)

(b)



Liste de vecindad inversa:
{ (0, 1), (0,0), (0, -1), (1, 0)}

$P_{ij}\ X_1 \leftarrow P_{i,j+1}\ Y_1$ (1)

$P_{ij}\ X_2 \leftarrow P_{ij}\ Y_2$ (2)

$P_{ij}\ X_3 \leftarrow P_{i,j-1}\ Y_3$ (3)

$P_{ij}\ X_4 \leftarrow P_{i+1,j}\ Y_4$ (4)

(c)

Figura 35. (a) Una celda, su vecindad y la lista de vecinos; (b) conexión de los ports de salida de la celda i,j (usando la lista de vecindad); (c) conexión de los ports de entrada de la celda i,j (usando la lista de vecindad inversa).

3.4 Especificación formal de modelos Cell-DEVS acoplados genéricos.

El procedimiento analizado en la sección anterior permitió definir espacios de celdas completos de forma paramétrica. En esta sección, este marco será usado para definir el acoplamiento de otros tipos de espacios de celdas.

La idea es extender la definición de modelos acoplados para permitir la especificación de espacios Cell-DEVS que puedan ser combinados con otros modelos básicos en una jerarquía DEVS. Por ejemplo, un modelo Cell-DEVS podría acoplarse con otro con distinta vecindad o comportamiento de celda, o con modelos que no sean Cell-DEVS. Esta definición permite construir modelos complejos consistentes de diversos submodelos con distinto comportamiento usando distintos paradigmas o niveles de abstracción.

Para permitir la definición múltiple de submodelos y su acoplamiento, un modelo acoplado Cell-DEVS genérico puede representarse como:

$$GCC = < Xlist, Ylist, I, X, Y, n, \{t_1,...,t_n\}, \eta, N, C, B, Z, select >$$

En este caso,

Ylist es la lista de acoplamiento de salida;

Xlist es la lista de acoplamiento de entrada;

I representa la definición de la interfaz modular del modelo;

X es el conjunto de eventos externos de entrada;

Y es el conjunto de eventos externos de salida;

n es la dimensión del espacio de celdas;

$\{t_1,...,t_n\}$ es la cantidad de celdas en cada una de las dimensiones;

η es el tamaño de la vecindad;

N es el conjunto de vecindad;

C es el espacio de celdas;

B es el conjunto de celdas del borde;

Z es la función de traducción; y

select es la función de selección ante eventos simultáneos.

Donde, para $\#T < \infty \wedge T \in \{N, Z, R, \{0,1\}\}$;

Ylist $\subseteq \{ (i_1,...,i_n) / i_k \in [1,t_k] \; \forall \; k \in [1,n], k \in N\}$;

Xlist $\subseteq \{ (i_1,...,i_n) / i_k \in [1,t_k] \; \forall \; k \in [1,n], k \in N\}$;

$\mathbf{I} = < P^X, P^Y >$ representa la definición de la interfaz modular del modelo. En este caso,
$\forall \; c \in I$, con **I** definido como en (4) $i \in \{X, Y\}$, P_c^i es una definición de un port,
$P_c^i = \{ (N_c^i, T_c^i) \; / \; \forall \; c \in i\text{list}, \; N_c^i \in i(c)$ (nombre del port), y $T_c^i \in T$ (Tipo del port)$\}$;

$X \subseteq T$;

$Y \subseteq T$;

$\mathbf{n} \in N$;

$\{t_1,...,t_n\} \in N$, con $t_k < \infty \; \forall \; k \in [1,n]$;

$$\eta \in N, \eta \leq \prod_{i=1}^{n} t_i;$$

$\mathbf{N} = \{ (v_{k1},...,v_{kn}) / \forall (k \in N, k \in [1, \eta]) \wedge (i \in N, i \in [1, n]); v_{ki} \in Z \wedge (ti - | v_{ki} | \geq 0) \}$;

$\mathbf{C} = \{ C_c / c \in \mathbf{I} \wedge C_c = < I_c, X_c, Y_c, S_c, N_c, d_c, \delta_{int_C}, \delta ext_c, \tau_c, \lambda_c, D_c >$ es un componente TDC, IDC ó UDC tal como los definidos anteriormente $\}$, siendo **I** definido como en (4);

$\mathbf{B} = \{\varnothing\}$ si el espacio de celdas es toroidal; o
$\mathbf{B} = \{C_b / C_b \in C$ con $b \in I \}$, con **I** definido como en (4). En este caso, el conjunto B está sujeto a la restricción que $\tau_b \neq \tau_c = \tau \; \forall \; (C_c \notin B) \wedge (C_b \in B)$.

En la mayoría de los casos, suele definirse

$B = \{C_b / C_b \in C \text{ con } b \in L\}$, siendo $L = \{ (i_1,...,i_n) / i_j = 0 \vee i_j = t_j \; \forall j \in [1, n] \}$, y sujeto a que $\tau_b \neq \tau_c = \tau \; \forall c \notin L$.

Z: $I_C \rightarrow I_D$ es la función de traducción, definida (usando (6)) como:

$Z(P_C{}^{Y_q}) = P_D{}^{X_q}$, $\forall (q \in N, q \in [1,\eta])$, donde $\forall V_k \in N$, $D = C +_t V_k$; y
$Z(P_C{}^{X_q}) = P_D{}^{Y_q}$, $\forall (q \in N, q \in [1,\eta])$, donde $\forall V_k \in N$, $D = C -_t V_k$.

select = $\{ (s_1, ..., s_n) / s_i \in N, \forall i \in [0, n] \}$, con la restricción que select $\subseteq t_1xt_2x...xt_n \rightarrow t_1xt_2x...xt_n$.

La presente especificación permite definir automáticamente un espacio Cell-DEVS modular consistente de distintos submodelos. La especificación es completamente independiente de los simuladores, y puede usarse para validar la correctitud del modelo.

Se pueden encontrar algunas diferencias con la especificación de los espacios Cell-DEVS cerrados definidos en la sección anterior. Primero, como en todo modelo acoplado DEVS que admite entradas y salidas, han sido incluidos los conjuntos X e Y. Como estos modelos pueden ser acoplados con otros modelos DEVS, también ha sido definida la interfaz **I**.

La función **Z** definida en la sección anterior se sigue usando para llevar a cabo el acoplamiento interno del espacio de celdas. Finalmente, se han incluido dos nuevos conjuntos: **Xlist**, una lista de las posiciones de las celdas donde se reciben eventos externos al modelo acoplado; e **Ylist**, una lista de posiciones de celdas cuyas salidas serán recolectadas para ser enviadas a otros modelos en la jerarquía. Estas listas se usarán para definir automáticamente otros ports de entrada/salida (distintos de los usados para recibir datos de los vecinos) en las celdas de interfaz delmodelo, com fue definido en la sección 3.2.

En modelos DEVS acoplados, la ocurrencia de eventos internos en un modelo es precedida por la ejecución de la función de salida λ. Esta función transmite el valor a otros modelos a través de ports en la interfaz. Para modelos Cell-DEVS, el comportamiento será distinto. Primero, si ocurre un evento en una celda, sus vecinos serán influenciados automáticamente a través de la ejecución de la función Z. Por otro lado, ciertas celdas en el espacio serán elegidas como celdas de entrada y salida, y serán incluidas en las listas Xlist y la Ylist respectivamente. Los valores de estas celdas serán considerados como las entradas y salidas de todo el espacio de celdas.

La función select se define como una lista de posiciones en la vecindad. La lista se ordena de acuerdo con el criterio de selección a ser utilizado cuando más de una celda está activa simultáneamente.

Para el caso de espacios de celdas bidimensionales, se define:

$$GCC_b = < \text{Xlist, Ylist, I, } X, Y, \eta, \text{ N, } \{f, c\}, \text{ C, B, Z, select} >$$

donde

Ylist = $\{ (k,l) / k \in [0,f], l \in [0,c] \}$;

Xlist = $\{ (k,l) / k \in [0,f], l \in [0,c] \}$;

I = $< P^X, P^Y >$, donde $\forall i \in \{X, Y\}$, $\forall ((k,l) \in i\text{list}, (k, l) \in [1, fxc])$, $P_{kl}{}^i = \{ (N(k,l)^i, T(k,l)^i) / N(k,l)^i = i(k,l)$ (nombre del port), y $T(k,l)^i = \{0, 1\}$ (tipo del port)$\}$;

$X = \{0, 1\}$;

$Y = \{0, 1\}$;

$\{f, c\} \in N$;

$\eta \in N, \eta \leq f.c$;

$\mathbf{N} = \{ (i_p,j_p) / (i_p, j_p \in \mathbf{Z} \wedge i_p, j_p \in [-1, 1]) \forall p \in \mathbf{N}, p \in [1,\eta] \wedge (0 < i_p \leq f\text{-}1 \wedge 0 < j_p \leq c\text{-}1)\}$.

$\mathbf{C} = \{C_{ij} / i \in [1,f] \wedge j \in [1,c]\}$, donde

$$C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij}>$$

es un componente atómico TDC, IDC ó UDC;

$\mathbf{B} = \{\varnothing\}$ si el espacio de celdas es toroidal; o

$B = \{C_{ij} / (i = 1 \vee i = f) \wedge (j = 1 \vee j = c) \wedge C_{ij} \in C \wedge C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij}>$ es un componente atómico TDC, IDC ó UDC }, si las celdas atómicas del borde tienen distinto comportamiento que el resto del espacio de celdas.

Z: $I_{ij} \rightarrow I_{kl}$, definida como

$Z(P_{ij}{}^{Y_q}) = P_{kl}{}^{X_q}$, con $(q \in N, q \in [1,\eta]) \wedge \forall (r,s) \in N$, $k = (i+r) \bmod f$; $l = (j+s) \bmod c$; y
$Z(P_{ij}{}^{X_q}) = P_{kl}{}^{Y_q}$, con $(q \in N, q \in [1,\eta]) \wedge \forall (r,s) \in N$, $k = (i-r) \bmod f$; $l = (j-s) \bmod c$.

select $= \{ (k,l) / (k,l) \in N \}$ es la función de selección ante eventos simultáneos, con la restricción de que select $\subseteq f \times c \rightarrow f \times c$.

Lema 4

Los modelos GCC_b son modelos acoplados DEVS.

Prueba:

Se debe probar que los conjuntos

$$CM = < X, Y, I, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, \text{select} >$$

y

$$GCC_b = < \text{Xlist}, \text{Ylist}, I, X, Y, \eta, N, \{f, c\}, C, B, Z, \text{select} >$$

son equivalentes.

La equivalencia de la mayoría de los conjuntos fue probada en el Lema 3. Retomando la prueba de dicho Lema, se tiene que

$I' \equiv I$;
$X' \equiv X$; y
$Y' \equiv Y$, por definición de cada uno de los conjuntos.

Por otro lado,

Xlist $\subseteq D$
Ylist $\subseteq D$

Por ende, los conjuntos son equivalentes y vale la proposición.

3.5 Modelos acoplados DEVS: redefinición para incluir modelos Cell-DEVS

La definición de modelos acoplados DEVS ha sido modificada para permitir la inclusión de modelos de base que sean espacios de celdas. Para estos, el esquema de acoplamiento deberá utilizar la definición interna y externa de celdas de entrada y salida definidas previamente. Por ende, un modelo acoplado DEVS serán definidos como:

$$CM = < I, X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

donde

$I = < P^X, P^Y >$ representa la definición de la interfaz modular del modelo. En este caso, $\forall\ j \in [1, \eta]$, $i \in \{X, Y\}$, P_j^i es una definición de un port (de entrada o salida respectivamente), donde

$$P_j^i = \{ (N_j^i, T_j^i) \ / \ \forall j \in [1, \mu], (\mu \in N, \mu < \infty), N_j^i \in [i_1, i_m] \text{ (nombre del port), y } T_j^i = \text{Tipo del port}\};$$

X es el conjunto de eventos externos de entrada;
Y es el conjunto de eventos externos de salida;
$D \in N$ es un índice de los componentes del modelo acoplado, y

$\forall d \in D$, M_d es un modelo básico DEVS, donde

$$M_d = GCC_d = < I_d, X_d, Y_d, Xlist_d, Ylist_d, n_d, \{t_1,...,t_n\}_d, \eta_d, N_d, \{f_d, c_d\}, C_d, B_d, Z_d, select_d >$$

si el modelo acoplado es Cell-DEVS

$$M_d = GCC_{bd} = < I_d, X_d, Y_d, Xlist_d, Ylist_d, \eta_i, N_d, \{f_d, c_d\}, C_d, B_d, Z_d, select_d >$$

si el modelo acoplado es Cell-DEVS binario bidimensional, y

$$M_d = < I_d, X_d, S_d, Y_d, \delta_{intd}, \delta_{extd}, D_d >$$

en otro caso.

I_d es el conjunto de modelos influenciados por el modelo d, y $\forall j \in I_d$,

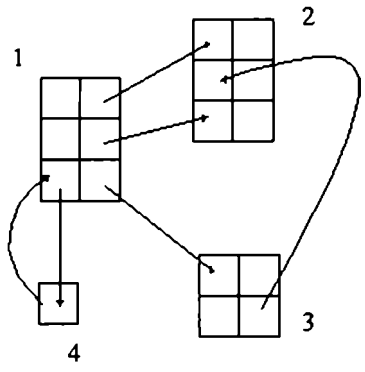
Z_{dj} es la función de traducción de d a j, donde

Z_{dj}: $Y_d \rightarrow X_j$ si ninguno de los modelos implicados son Cell-DEVS,
Z_{dj}: $Y(c_1)_d \rightarrow X(c_2)_j$, con $(c_1) \in Ylist_d$, y $(c_2) \in Xlist_j$ si alguno de los modelos d o j es un GCC, o
Z_{dj}: $Y(f,g)_d \rightarrow X(k,l)_j$, con $(f,g) \in Ylist_d$, y $(k,l) \in Xlist_j$ si alguno de los modelos d o j es un modelo GCC_b.

Finalmente, **select** es el selector ante eventos simultáneos.

Cuando se ejecuta un modelo Cell-DEVS, la función Z_{ij} traduce las salidas en entradas entre una celda y los modelos externos al espacio de celdas usando las dos listas definidas previamente. Los nombres de los ports de entrada/salida también se definirán usando los contenidos de las listas Xlist e Ylist, usando un identificador (X para los de entrada, Y para los de salida) y la posición de la celda, y posteriormente se mapearán estos pares en números mayores que η, para seguir la definición de la sección .3.2

Para ejemplificar esta nueva definición para acoplamientos externos, supongamos que deben acoplarse los modelos que se ven en la Figura 36. El acoplamiento será realizado usando la función Z_{ij}, que usa los contenidos de las listas Xlist e Ylist.



(a)

$Xlist_1 = \{ (3,1) \}$
$Ylist_1 = \{ (1,2), (2,2), (3,2) \}$
$Xlist_2 = \{ (1,1), (2,1), (3,1) \}$
$Ylist_2 = \{\varnothing\}$
$Xlist_3 = \{(1,1)\}$
$Ylist_3 = \{(2,2)\}$

(b)

$Y(1,2)_1 \rightarrow X(1,1)_2$
$Y(2,2)_1 \rightarrow X(3,1)_2$
$Y(3,2)_1 \rightarrow X(1,1)_3$
$Y(3,1)_1 \rightarrow X_4$
$Y_4 \rightarrow X(3,1)_1$
$Y(2,2)_3 \rightarrow X(2,1)_2$

(c)

Figura 36. Ejemplo de acoplamiento usando la función Z_{ij} para espacios Cell-DEVS (a) Modelos básicos; (b) listas Xi e Yi para cada modelo; (c) Acoplamiento Z_{ij}.

La presente definición de modelos Cell-DEVS es equivalente a la de otros modelos acoplados DEVS. Intuitivamente, en este caso, D = f x c, $\{M_i\}$ está definido por f, c, C y B, el conjunto de vecinos inminentes Ii está definido por N, y la función de traducción Z_{ij} está definida por Z y Z_{ij}..

Lema 5

Los modelos CM son modelos DEVS acoplados.

Prueba:

Como fue probado que los modelos GCC_b son modelos básicos DEVS, y todos los conjuntos definidos son equivalmentes a los de los modelos acoplados DEVS, basta con probar que la función Zdj es equivalente a la definida para modelos DEVS. Pero aquí

$Y(f,g)_d \subseteq Y_d$
$X(k,l)_j \subseteq X_j$

por definición de las listas Xlist e Ylist. Luego, ambos conjuntos son equivalentes y la proposición es válida

■

3.6 Clausura bajo acoplamiento de los modelos Cell-DEVS

La idea de esta sección es probar que los modelos Cell-DEVS son cerrados bajo clausura. Esto implica que un modelo Cell-DEVS acoplado puede asociarse con un modelo DEVS básico. Ello permite integrar el modelo en la jerarquía de modelos, y reutilizar submodelos que fueron verificados.

Lema 6

Los modelos GCC_b son cerrados bajo clausura.

Prueba:

Para probar la clausura, es necesario mostrar que el modelo acoplado

$$GCC_b = < I, X, Y, Xlist, Ylist, \eta, N, \{f, c\}, C, B, Z, select >$$

es equivalente al modelo básico resultante definido por

$$M = < I, X, S, Y, \delta_{int}, \delta_{ext}, D >$$

Sea $S = \underset{\substack{i=1,\ldots,f \\ j=1,\ldots,c}}{X} Q_{ij}$

donde $\forall (i,j) \in [1,f] \times [1,c]$, $Q_{ij} = \{ (s_{ij}, e_{ij}) / s_{ij} \in \theta_{ij} \times N_{ij} \times d_{ij};\ e_{ij} \in [0, D_{ij}(s_{ij})] \}$; siendo $C_{ij} \in C$ un modelo TDC, IDC ó UDC.

$D: S \rightarrow \boldsymbol{R}_0^+ \cup \infty$, definido como $D(s) = \min \{ \sigma_{ij} / (i \in [1,f], j \in [1,c]) \ \forall C_{ij} \in C, \sigma_{ij} = D_{ij}(s_{ij}) - e_{ij} \}$.

En base a esta definición de la función de tiempo transcurrido, se puede definir al conjunto de componentes inminentes como:

$IMM(s) = \{ (i,j) / i \in [1,f], j \in [1,c] \wedge \sigma_{ij} = D(s) \}$

Sea $(i,j)^* = select(IMM(s))$. Este es el componente que será ejecutado.

$\delta_{int}: S \rightarrow S$. Sea $s = (\ldots, (s_{ij}, e_{ij}, N_{ij}, d_{ij}), \ldots)$. Entonces, $\delta_{int}(s) = s' = (\ldots, (s_{ij}', {}_{ij}', N_{ij}', d_{ij}'), \ldots)$, con

$$(s_{ij}', {}_{ij}', N_{ij}', d_{ij}') = \begin{cases} (\delta_{int\ ij}(s_{ij}), 0, N_{ij}, d_{ij}) & \text{si } (i,j) = (i,j)^* \\ (\delta_{ext\ kl}((s_{kl}, e_{kl} + D(s)), X_{kl}), 0, N_{kl}, d_{kl}) & \text{si } (k,l) \in N_{ij^*} \\ (s_{ij}, e_{ij} + D(s), N_{ij}, d_{ij}) & \text{sino} \end{cases}$$

donde $X_{kl} = Z_{ij^*,\ kl}(\lambda_{ij^*}(s_{ij^*}))$

Esto significa que la función de transición interna cambia el estado de la celda inminente $(i,j)^*$ de acuerdo con su función de transición interna, y actualiza todos los componentes influenciados de acuerdo con las entradas producidas por la salida de $(i,j)^*$. En los demás componentes sólo se actualiza el tiempo transcurrido.

$\delta_{ext}: Q \times X \rightarrow S$

$\delta_{ext}((s, e), x) = s' = (\ldots, (s_{ij}', {}_{ij}', N_{ij}', d_{ij}'), \ldots)$ con

$$(s_{ij}', {}_{ij}', N_{ij}', d_{ij}') = \begin{cases} (\delta_{ext\ kl}(s_{kl}, e_{kl} + e, X_{kl}), 0, N_{ij}, d_{ij}) & \text{si } (k,l) \in N_{kl} \\ (s_{ij}, e_{ij} + e, N_{ij}, d_{ij}) & \text{sino} \end{cases}$$

con $X_{kl} = Z_{ij,\ kl}(x)$. O esa, todas las celdas influenciadas por la entrada externa x cambian de estado de acuerdo con la entrada x(k,l) transmitida a ellos, y convertida por el mapeo entre interfaces. El resto sólo cambia su tiempo transcurrido.

Finalmente, $\lambda: S \rightarrow Y$.

$$\lambda(s) = \begin{cases} Z_{ij^*,\ kl}(\lambda_{ij^*}(s_{ij^*})) & \text{si } (i,j)^* \in I_{kl} \\ \varnothing & \text{sino} \end{cases}$$

Es decir, si $(i,j)^*$ envía una salida, se la traduce en una entrada al modelo acoplado por el mapeo entre interfaces $Z_{ij^*,\ kl}$.

■

4. SIMULACIÓN DE ESPACIOS CELL-DEVS

En el capítulo anterior se presentaron especificaciones formales para modelos conceptuales basados en el formalismo Cell-DEVS. Esta sección está dedicada a analizar distintas cuestiones relacionadas con el desarrollo de los modelos de implementación que deben ejecutarse en base a las especificaciones realizadas usando el marco conceptual dado por el formalismo.

Se consideraron dos puntos de vista distintos. Primero, se tomó en cuenta el punto de vista del usuario para producir simulaciones seguras y efectivas en cuanto sus costos de desarrollo y mantenimiento. Un entorno de simulación Cell-DEVS debería proveer facilidades para ejecutar modelos de implementación especificados de acuerdo con los formalismos definidos previamente. Esto significa que un usuario debería ser capaz especificar un modelo conceptual usando alguna de las definiciones previas, y que dicha especificación debería ser ejecutable. De esta forma, si la especificación está completa, la correctitud del modelo ejecutable puede verificarse automáticamente.

Como fuera mencionado, el especificar formalmente el modelo del sistema permite su validación contra el comportamiento del sistema real. Si el modelo de ejecución sigue la especificación formal del modelo (y el entorno de simulación está construido correctamente), los simuladores ejecutarán correctamente. Por ende, si se detectara un error durante la simulación, este sólo dependerá de la especificación del sistema real.

Por otro lado, se tomó en cuenta el punto de vista del diseñador del entorno de modelización. Tal entorno debería ejecutar modelos Cell-DEVS especificados por medio de los formalismos, y para ello se han considerado otras definiciones existentes para simulación de modelos. La jerarquía de clases propuesta por Zeigler para entornos de modelado orientados a objeto [Zei90] fue modificada para permitir la ejecución eficiente de modelos de implementación para espacios Cell-DEVS.

En todo el capítulo se considerarán espacios de celdas binarios bidimensionales, ya que el objetivo es analizar la implementación de un entorno de tales características. A pesar de esto, los principales aspectos mencionados son válidos para espacios de celdas n-dimensionales, o pueden adaptarse con modificaciones mínimas.

4.1 Definición de un modelo de implementación Cell-DEVS.

Para cumplir con los objetivos propuestos, todo entorno para simulación Cell-DEVS debería adecuarse al comportamiento definido en las especificaciones estudiadas en las secciones 3.2, 3.3 y 3.4. Aquí se enfocarán los principales conceptos relacionados con la definición de los mecanismos de simulación de estos espacios, y el capítulo 6 describirá con detalle la implementación de una herramienta basada en ellos.

El usuario de un entorno de simulación de un entorno de modelado Cell-DEVS debe poder representar tres aspectos básicos del espacio de celdas: dimensión, influencias y comportamiento. Las cuestiones de dimensión e influencias están relacionadas con la definición del espacio de celdas. En cambio, los aspectos de comportamiento están relacionados con la definición de cada celda. Esta sección está dedicada a presentar las características que debe tener un entorno de simulación que considere estos tres aspectos. Estas definiciones fueron introducidas en [WFG97], y sus conceptos fueron implementados posteriormente en una herramienta de simulación [BBW98b].

Uno de los objetivos es que un modelador pueda caracterizar un modelo de implementación completo sin la necesidad de programarlo en un lenguaje de programación tradicional. En cambio debe poder especificar el espacio de celdas completo por medio de un lenguaje de especificación para el espacio de celdas, y definir las propiedades generales del modelo de forma paramétrica.

Con esta finalidad se han utilizado los enfoques teóricos definidos en [Zei84, Zei90], respetando las definiciones de jerarquías de clases para simular modelos DEVS definidas en estos trabajos. Como base para los desarrollos también se han considerado algunas herramientas de simulación de modelos DEVS [Zei90, Zei97, Cho95, GFE95], tomando como base esos marcos, y modificándolos para permitir la definición de un modelo celular completo basado en el nuevo formalismo. En la jerarquía de clases redefinida (que se presenta a continuación) se han incluido las principales clases del entorno de simulación Cell-DEVS [BBW98b] (cuya implementación se analizará en el Capítulo 6). También se ha extendido con el fin de permitir simulaciones en paralelo.

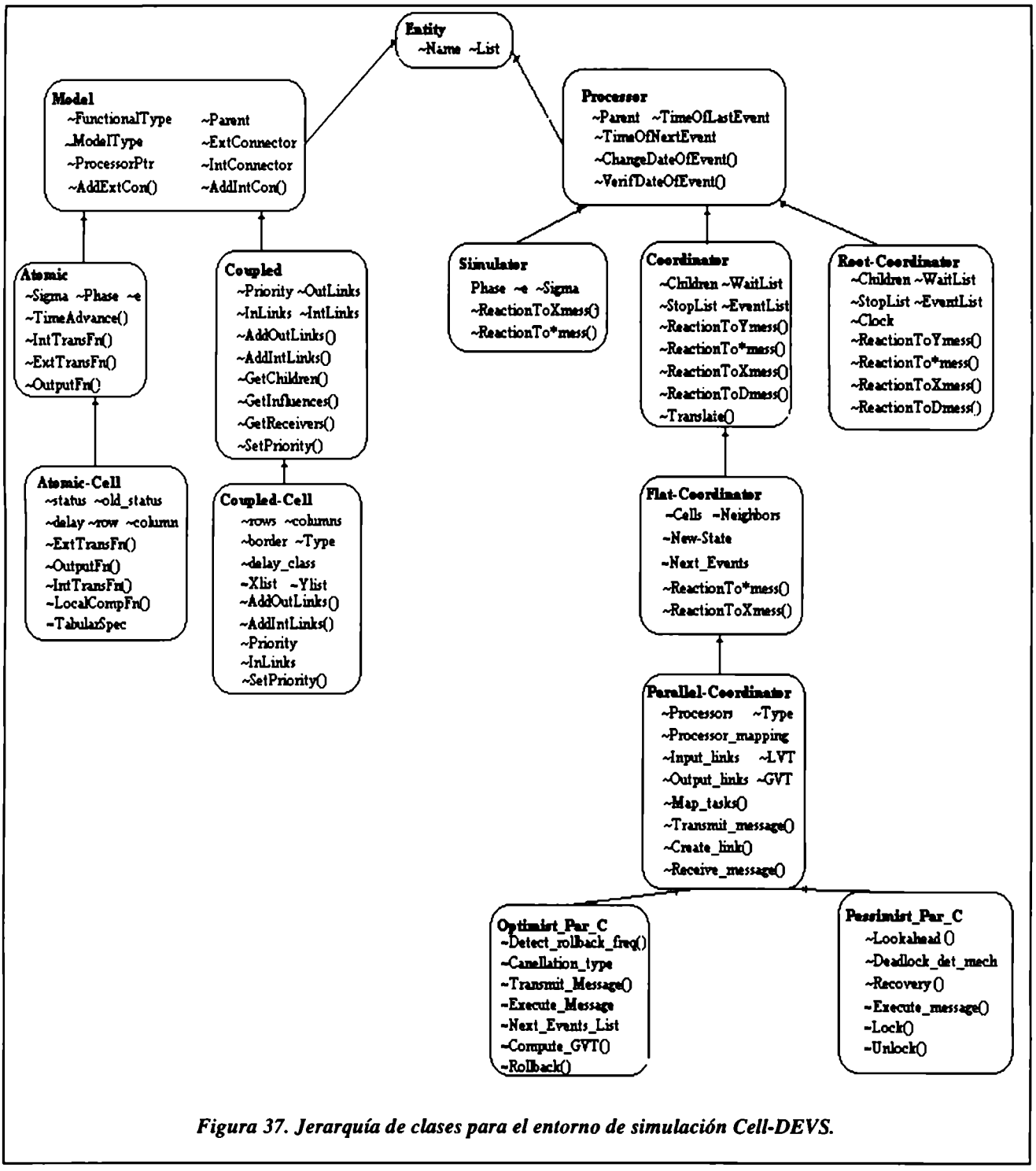


Figura 37. Jerarquía de clases para el entorno de simulación Cell-DEVS.

Analizando la jerarquía de clases, podemos ver dos clases principales derivadas de una clase raíz llamada *Entity* (Entidad): *Models* (Modelos) y *Processors* (Procesadores). La clase *Models* permite definir el comportamiento para los modelos en el sistema. Cada modelo puede ser *Atomic* (Atómico) o *Coupled* (Acoplado). Las clases *Atomic-Cell* (Celda Atómica) y *Coupled-Cell* (Celda Acoplada) actúan como interfaz entre el usuario y el entorno de simulación. La instanciación de estas clases permite definir la especificación de los modelos presentados en las secciones 3.2, 3.3 y 3.4.

Con respecto a los aspectos de **dimensión** del modelo, un usuario del entorno debería definir el tamaño del espacio de celdas, incluyendo el número de dimensiones y la cantidad de celdas en cada una de ellas (en el caso de espacios binarios bidimensionales, los parámetros f y c de la especificación formal). Los parámetros definidos por el modelador se usarán para definir las variables de instancia *~rows* y *~columns*.

Luego deben especificarse las **influencias** de las celdas, definiendo el formato de la vecindad. Con esta información deberán crearse automáticamente los ports de entrada y salida para cada celda, siguiendo los procedimientos mencionados en la definición formal. El acoplamiento de las celdas también se especifica automáticamente, tomando como base la definición de la vecindad usando el procedimiento presentado en la sección 3.4. Cuando el espacio de celdas deba ser acoplado con otros modelos DEVS, también deben especificarse las conexiones externas, por medio de la definición de las listas Ylist y Xlist.

En este sentido, el método *~AddIntLinks()* de la clase *Coupled_Cell*, estará encargado de definir los nombres de los ports (como ya fuera mencionado), y de agregar los vecinos de cada celda en la variable de instancia *~InLinks*. Esta variable será usada de distinta forma que para otros modelos DEVS: sólo contendrá una lista de vecinos, que será usada por el mecanismo de simulación, de la forma que será analizada más adelante. Además, para permitir la integración entre modelos de celdas y otros modelos DEVS, la clase *Coupled_Cell* incluye el mecanismo de acoplamiento de modelos presentado en la sección 3.5 dentro del método *~AddOutLinks()*. Los ports de salida en la interfaz de un modelo modular se deben conectar con los ports de entrada en otra especificada. Para ello se debe ejecutar la siguiente secuencia:

$\forall (p,q) \in Ylist \wedge (i,j) \in Z_{ij}$
crear y conectar un port Y de la celda C_{pq} del modelo M_i con un port de entrada binario en el modelo M_j;

$\forall (p,q) \in Xlist \wedge (i,j) \in Z_{ij}$
crear y conectar un port de salida binario del modelo M_i con un port X de la celda C_{pq} en el modelo M_j.

Esta información se usará para almacenar las variables de instancia *~InLinks* y *~OutLinks* del modelo acoplado. Utilizando estos parámetros, puede generarse un espacio completo de celdas y definirse su acoplamiento. Al finalizar la inicialización del modelo, quedarán definidas las variables de instancia *~Xlist*, *~Ylist*, *~InLinks*, *~IntLinks*, y *~OutLinks*.

Para definir el **comportamiento** de la celda, debe comenzarse por incluir el tipo de demora (de transporte, inercial o ninguna) y su duración. Esta información debe almacenarse en las variables de instancia *~delay_class* (del modelo acoplado) y *~delay* respectivamente. Con esta definición, puede considerarse que la demora de cada celda es diferente de la de las otras.

Luego debe especificarse la función de cálculo local, para lo cual se ha definido un lenguaje de especificación simple (cuya BNF ha sido incluida en el Apéndice). Un compilador debe traducir una especificación hecha en este lenguaje a una especificación simulable del comportamiento. Ello implica realizar una representación interna en los modelos Cell-DEVS.

Usando este lenguaje, el comportamiento de una celda se define como una lista de sentencias, cada una de las cuales tiene el siguiente formato:

{0 | 1 | ϕ} (Expresión lógica)

Los estados indefinidos serán manipulados lógicamente usando las siguientes reglas:

AND	ϕ
0	ϕ
1	ϕ
ϕ	ϕ

OR	ϕ
0	ϕ
1	1
ϕ	ϕ

NOT	
ϕ	ϕ

Figura 38. Extensiones de operaciones booleanas para estados indefinidos [AAD97].

Cada expresión lógica relaciona varias subexpresiones enteras o de tres estados (usando lógica trivalente). El lenguaje sólo acepta estados ternarios para una celda, y las expresiones pueden incluir operaciones relacionales y llamadas a determinadas funciones predefinidas. Los resultados de cada celda se calculan analizando cada sentencia, y considerando el estado actual para la celda y su vecindad. Los operadores y funciones principales que pueden usarse se encuentran en la siguiente tabla:

Operadores	Comportamiento
<, <=, >, >=	Operadores de desigualdad
=, !=	Operadores de igualdad
AND, OR, NOT	Operadores booleanos
( )	Operador de precedencia
+, -, *, /, ^	Operadores aritméticos
Funciones	
one_count()	Devuelve el número de vecinos cuyo estado es uno
zero_count()	Devuelve el número de vecinos cuyo estado es cero
fi_count()	Devuelve el número de vecinos cuyo estado es indefinido
random(tipo, esper.)	Devuelve un número aleatorio calculado según una distribución tipo y con la esperanza utilizada como parámetro.
external(nombre)	Ejecuta una función definida externamente
Delays	
tr_delay(d);	Demora d unidades de tiempo la ejecución de la celda usando demora de transporte
in_delay(d);	Demora d unidades de tiempo la ejecución de la celda usando demora inercial

Tabla 4. Principales operadores y funciones para definir el comportamiento de una celda.

La especificación del modelo será completada definiendo el comportamiento de las celdas de los bordes. El espacio de celdas puede ser toroidal, en cuyo caso el procedimiento de acoplamiento se encargará de la conexión usando el tamaño del espacio de celdas y la definición de la vecindad. Si, en cambio, se quiere proveer distinto comportamiento para los bordes, puede usarse el lenguaje de especificación. La variable de instancia *~border* almacenará las celdas que forman parte del borde. Finalmente, deben especificarse las prioridades para tratar eventos simultáneos, permitiendo clasificar las celdas inminentes en el espacio. Las prioridades pueden usarse para definir la función select de la especificación DEVS. Esta información se almacena en la variable de instancia *~Priority*, que en este caso será una lista de vecinos, usando el método *~SetPriority()*.

Los parámetros mencionados se utilizan para crear un arreglo de celdas atómicas y para definir las influencias de cada celda. Los nombres de estos modelos atómicos se crean en base a los parámetros definidos, utilizando los nombres del modelo de celdas que componen los mismos, y los tamaños deseados en cada una de las dimensiones. Los ports de entrada y salida de cada uno de los modelos de celdas también son creados en base a estos parámetros. Las funciones de transición externa, interna y de salida para cada celda, serán las definidas con anterioridad en la sección 3.2.

Para ejecutar el modelo de celdas, la especificación del comportamiento de cada celda definida usando el lenguaje de especificación debe traducirse a una especificación ejecutable. Esto puede hacerse de distintas formas, pero se ha elegido utilizar una aproximación que permita ejecutar diferentes funciones de cálculo de acuerdo con el espacio de celdas a modelar. Por esto, la función de cálculo local definida en la Figura 37 (*~LocalCompFn()*) tendrá un comportamiento fijo, de forma tal que no deba ser reprogramada para los distintos modelos. Esta función de cálculo tomará como base la descripción del comportamiento definida usando el lenguaje de especificación, y ejecutará, para cada caso, una implementación tabular de la especificación.

La idea será usar una tabla en la que se ingresarán las distintas expresiones de la especificación usando un formato fijo (*~TabularSpec*). Como fuera mencionado, cada expresión estará compuesta por una o varias funciones relacionadas por conectores lógicos. El traductor de la especificación tomará cada expresión, la analizará, y pondrá su resultado en una tabla con el siguiente formato:

Result.	Variables						Operadores						Valor	And
	i_1	...	i_9	$\Sigma 0$	$\Sigma 1$	$\Sigma \phi$	<	<=	>	>=	=	!=		
0	*								*				7	
			*				*						5	
1				*							*		2	*
						*		*					5	
...														

Tabla 5. Formato de las tablas de cálculo local.

Cada una de las columnas de la izquierda representa una de las posibles variables implicadas en el cálculo local. Las columnas marcadas i_1-i_9 se usan para representar los valores de los contenidos de las entradas memorizadas en cada celda, que reciben valores de alguna de las nueve celdas vecinas. Las demás columnas representan el resultado obtenido si se ejecuta una de las funciones predefinidas. Las componentes de la expresión se relacionan con el valor mencionado en la columna de la derecha, por medio de alguno de los operadores existentes. Si el resultado de la operación lógica es verdadero, la función de cálculo local devuelve el resultado correspondiente a la fila.

Cada expresión se insertará en una línea separada que expresa la operación deseada, sus operandos y los resultados que se obtienen al ejecutarla. Las expresiones conectadas por operadores lógicos OR serán divididas en dos líneas. Por ende, distintas sentencias también serán consideradas como conectadas por un operador relacional OR, y puestas en líneas separadas. Las expresiones conectadas por operadores booleanos AND serán ingresadas en dos líneas seguidas, pero distinguiéndolas con una marca especial, de forma tal que el procedimiento asociado con la función de transición local haga un AND de las expresiones. Cuando varias subexpresiones están agrupadas por medio del operador de precedencia (paréntesis), el resultado correspondiente a la expresión se ubicará en la línea correspondiente a la primera expresión.

La función de cálculo local recorrerá la tabla de a una línea a la vez, verificando las expresiones lógicas incluidas, y obteniendo sus resultados. De esta forma se obtiene y actualiza el valor del nuevo estado para la celda.

En tiempo de ejecución pueden encontrarse diversos errores en la especificación. Por ejemplo, si existe más de un resultado para una celda tomando como precondición los mismos valores de entrada y estado originales, esto es una señal de que la especificación es ambigua. En cambio, si no existen resultados para un estado dado, la especificación de la celda está incompleta. Si distintas expresiones producen el mismo resultado, la simulación puede seguir, pero el usuario debe ser notificado del posible error. En cambio, si se encuentran distintos resultados para dos precondiciones diferentes, la simulación debe ser detenida, ya que no puede determinarse el estado futuro de una celda.

Estos chequeos permiten detectar inconsistencias en la definición del modelo, las cuales pueden ser reportadas al usuario, que puede cambiar la especificación para asegurar su correctitud. De esta forma la validez del modelo puede ser verificada.

Cuando todos los parámetros para el espacio de celdas han sido completados, se inicializa el estado del espacio de celdas, luego de lo cual la simulación puede comenzar. Para ello deben definirse condiciones iniciales para cada celda, y también condiciones de terminación para el ciclo de simulación (en forma de un tiempo máximo de evolución).

4.2 Simulación abstracta de espacios Cell-DEVS

Una vez que el espacio de celdas haya sido definido, el entorno debe simular el modelo acoplado. En esta sección se analizará el mecanismo de simulación propuesto para estos espacios de celdas, que se basa en la existencia de diversos coordinadores con distinta funcionalidad que permiten simular el comportamiento de estos modelos. Los Coordinadores (*Coordinator*) y Simuladores (*Simulator*) están asociados con los modelos atómicos y acoplados jerárquicos, tal como en otros entornos DEVS, y estas definiciones están tomadas de las originalmente definidas en [Zei90]. Las principales diferencias residen en los métodos usados para crear los procesadores.

Cuando se define un modelo acoplado de celdas, también se debe crear automáticamente el conjunto de procesadores asociados con el mismo. Como el modelo acoplado consiste de un espacio de distintos modelos (celdas atómicas), se deberá asociar un simulador con cada una de ellas. Los nombres de los simuladores también se definen en base a los parámetros del modelo de celdas, como se hizo para generar los nombres de los modelos. Por otro lado, al establecerse los acoplamientos internos de la forma que fuera presentado en la sección 3.3, también se asocia un coordinador con el modelo acoplado. Los hijos de dicho coordinador (variables de instancia *~Children* y *~Parent*) son los simuladores recién mencionados. Para el coordinador modelo de celdas, los hijos serán todos los componentes del espacio. Para cada uno de los simuladores, el procesador padre (*~Parent*) será instanciado al coordinador actual.

Inicialmente, todas las listas de los procesadores son inicializadas como vacías, y las variables de estado puestas a cero, incluyendo las variables de manejo de tiempo. Estas variables se usarán con el fin de controlar la función de duración de vida D.

Los modelos se inicializarán por medio de mensajes especiales (mensajes I) en base a archivos de configuración de estados iniciales para cada una de las celdas. De ahí en más, la simulación avanzará como en otros entornos DEVS, lo

que fue analizado en la sección 2.2.3, por medio del uso de los métodos *ReactionToXMessage()*, *ReactionToYMessage()*, *ReactionTo*Message()* y *ReactionToDoneMessage()*. Como estas funciones de reacción a los mensajes no han sido redefinidas, a continuación se analizará la interacción de mensajes al simular espacios de celdas.

Cuando el un modelo atómico inminente es seleccionado, se debe enviar un mensaje * a su simulador, pasando por los coordinadores de nivel intermedio. Los coordinadores, en este caso, tendrán un comportamiento distinto al presentado en la sección 2.2.3 para detectar el destino del mensaje, por lo que se han redefinido los métodos de manipulación de mensajes. Aquí se deberá considerar que el campo de destino de los mensajes está compuesto por el nombre del modelo acoplado de celdas y la posición de la celda dentro del mismo. El nombre del espacio de celdas será utilizado por los coordinadores de mayor nivel para enviarle el mensaje al modelo acoplado asociado, y la posición de la celda será empleada por el coordinador del espacio de celdas para enviar el mensaje al simulador correspondiente. Los coordinadores intermedios consultarán sus listas de hijos inminentes, hasta llegar al coordinador de menor nivel, que activará al simulador asociado con la celda inminente, ejecutando la función de salida y la función de transición interna del modelo.

Cuando un mensaje externo de entrada llega a una celda, se consume un mensaje **X** y se ejecuta la función de transición externa. El simulador del modelo devolverá mensajes **listo** e **Y**, que se convierten en nuevos mensajes * y **X**, respectivamente, usando para ello el acoplamiento de celdas definido previamente, y la función de traducción asociada con dicho acoplamiento (*~Translate()*).

El comportamiento de esta función es dependiente de cómo hayan sido definidos los vecinos de cada celda, los bordes y la función select. En general, los mensajes **Y** se traducirán en mensajes **X** para cada uno de los vecinos, usando la conexión definida por la lista de vecindad. Simultáneamente se transmitirán mensajes **listo** a los coordinadores de nivel superior, para indicar la actualización de las horas del modelo atómico correspondiente a una celda. Un mensaje **listo** de un modelo atómico también es transmitido al coordinador de nivel superior, para actualizar la hora de todo el modelo. Si en un momento determinado existen dos eventos para una misma celda, se encolarán usando el criterio especificado por la lista de prioridades, que, como fuera analizado, es una lista de posiciones celdas ordenada de acuerdo a su prioridad de atención.

En el caso que el autómata sea toroidal, el comportamiento en los bordes será similar al recién mencionado, pero al ejecutarse en los extremos del espacio de celdas, los mensajes se difundirán a los extremos opuestos. En cambio, si el comportamiento de los bordes fue especificado por separado, se ejecutará una función de transición local alternativa para cada celda del borde.

4.3 Simulación achatada de los espacios de celda

Como fuera mencionado, la definición jerárquica de los módulos en el paradigma DEVS tiene asociado un mecanismo de simulación cuya interacción también es jerárquica. Las entidades especializadas en manejar la simulación de los modelos (Coordinadores y Simuladores), comparten información por medio del pasaje de mensajes. Esta interacción jerárquica entre los módulos produce un alto grado de overhead que puede reducir la performance en tiempo de ejecución. Este overhead es especialmente alto en el caso de simulaciones Cell-DEVS, y mucho mayor en el caso en que los modelos estén compuestos por un gran número de celdas.

Una forma de reducir el overhead es evitar la interacción entre los módulos y el alto grado de intercambio de mensajes, para lo cual se debe achatar la jerarquía de modelos. Este es el objetivo de la clase *Flat_Coordinador* incluida en la Figura 37.

Un coordinador chato se implementará como un arreglo bidimensional de registros asociado con el espacio de celdas (*~Cells*). En él, cada registro incluirá información sobre el estado y la demora de cada celda, y usará una lista de vecindad única (*~Neighbors*), en la que se almacenará el formato de los influenciados de cada celda en el espacio, como puede verse en la Figura 39.

La lista de vecinos almacenará el formato de la vecindad como una lista de posiciones relativas a la celda de origen. También se hace necesario utilizar una lista de próximos eventos (ordenada por la hora de los eventos) con el fin de registrar las celdas que han sido planificadas para ejecutar sus funciones de transición. Finalmente, el arreglo bidimensional de celdas se usa para registrar el estado actual del espacio. Este también se utiliza para detectar cambios en el modelo, y cuando hay un cambio, actualizar la lista de próximos eventos.

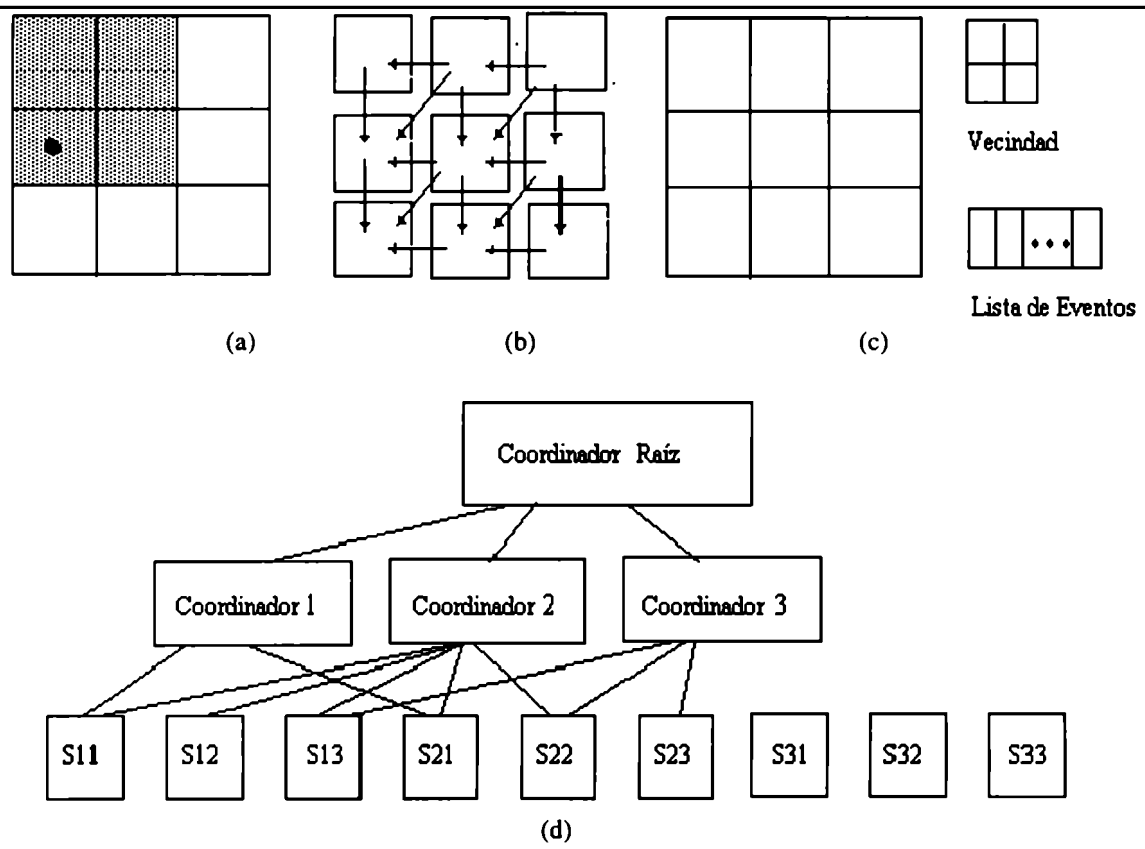


Figura 39. Estructuras del espacio de celdas. (a) Espacio de celdas básico y su vecindad. (b) Acoplamiento usando coordinadores jerárquicos. (c) Definición usando modelos achatados. (d) Jerarquía de procesadores para (b).

Como los eventos generados con el mismo tiempo simulado para una celda dada (ya sean transiciones internas o externas) deberían tratarse juntos, deben recopilarse los valores para un tiempo simulado determinado, y los resultados obtenidos en los cálculos locales ejecutarse simultáneamente. Los valores almacenados en la lista de Next_Events se usan para actualizar el arreglo de celdas cuando hayan terminado todos los cálculos para el mismo tiempo simulado, permitiendo el avance correcto de la función de transición global para el espacio de celdas.

Con el uso del coordinador chato, la interacción jerárquica de mensajes entre procesadores no será necesaria, ya que los múltiples procesadores intermedios son eliminados. Las diferencias entre ambas aproximaciones pueden verse en la Figura 40.

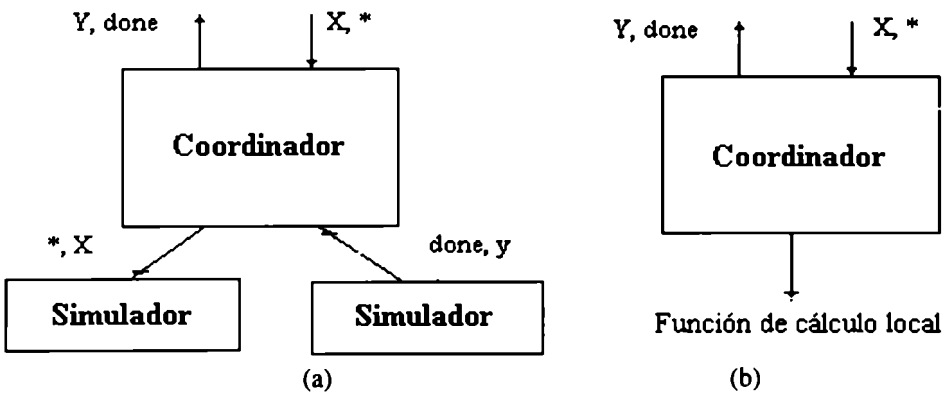


Figura 40. Reacción de los coordinadores ante mensajes (a) Jerárquico (b) Achatado.

El coordinador para modelos achatados comenzará la simulación detectando los estados latentes iniciales en el espacio de celdas. Los estados no latentes (es decir, aquellos cuyo estado puede cambiar en el futuro inmediato), serán agregados en la lista de próximos eventos ordenados de acuerdo con la hora del próximo evento.

Un coordinador chato se activará al recibir un mensaje *. Al ser activado, el coordinador tomará el primer elemento de la lista de próximos eventos, y si el evento considerado es interno, será activada la función de cálculo local de la celda. Si el estado de la celda ha cambiado, deberá retornarse el nuevo valor, y se determinará la hora del próximo evento. El coordinador creará un mensaje **Y** conteniendo el valor del estado de la celda y la hora actual, y lo transmitirá al coordinador de mayor nivel. Finalmente, se creará un nuevo mensaje y se agregará en la lista de próximos eventos para cada celda de la vecindad.

Otra tarea del simulador es convertir los mensajes de entrada al formato de la celda. Para ello usa la función de acoplamiento externo Z_{ij}. Los mensajes **X** de otros modelos externos simplemente se envían al coordinador, que los inserta en la lista de próximos eventos (como a los eventos externos generados por las celdas vecinas). Los mensajes **Y** generados por el simulador chato también son traducidos (usando la inversa de la función de traducción), y se transmiten a los coordinadores de nivel superior. Cuando cada ciclo de simulación termina, se crea un mensaje **listo**, que es enviado al coordinador de nivel superior, con el fin de actualizar la hora de simulación.

El comportamiento detallado del simulador chato para espacios Cell-DEVS binarios (con demoras de transporte o inerciales) puede representarse por medio de los siguientes algoritmos:

Estructuras de datos:

```
Cells = { binary state; float delay; } [f x c];
Neighbors = { (int i, int j), i = -1..1; j = -1..1 };          /* Posiciones relativas de las celdas */
Lista Next-Event = { Cell_position (int i, int j), float time};
binary New-State;
float tn, tl, e; /* hora del siguiente evento, último evento y tiempo transcurrido */


Init() {
  Cargar el estado inicial y las demoras para cada celda en el arreglo;

  Detectar celdas no latentes;
  For cada celda no latente do
        Agregar {Cell_position(i,j), delay} a la lista Next-Event;
  endfor
}

Reaction_to_X-message() {
        tl = tn;
        Tomar el primer ítem de la lista Next-Event;
        tn = hora del primer ítem de la lista Next-Event;
        e = 0;

        New-State = τ(N[i,j]);       /* Calcular la función de cómputo local para
                                        la celda que recibe el mensaje de entrada */
        If (New-State != State) then
            almacenar {Cell_position(i,j), Cells[Cell_position].state} en la lista Next-Event;
            /* No insertar elementos repetidos */
        endif

          If (INERTIAL-DELAY) then
                If (hay un elemento para esa celda con Sigma  mayor que el nuevo) then
                        descartar el elemento de la lista;     /* Remoción */
                endif
          endif
        Enviar un mensaje listo al coordinador de mayor nivel (incluyendo la celda y la hora);
}
```

```
Reaction_to_*-message () {
        tl = tn;
        Tomar el primer ítem de la lista Next-Event;
        tn = hora del primer ítem de la lista Next-Event;
        e = 0;

        /* Tomar las celdas inminentes. Ejecutar sus funciones de transición internas */
        For cada celda (i,j) cuyo mensaje en la lista Next-Event tenga hora = tn do
          actualizar el estado de la celda (si fuera necesario);

          If (la celda está en Ylist) then          /* Acoplamiento externo */
                crear un mensaje Y (incluyendo la hora de próximo evento) y enviarlo al coordinador padre;
          endif

          /* Tareas en común para demoras de transporte o inerciales */
          For cada par (k, l) en la lista de Vecinos do
                Next_event_time = tn + Cells(i+k, j+l).delay;
                /* Acoplamiento interno */
                Agregar { (i+k, j+l), Next_event_time} en la lista Next-Event;
                If existe una posición para ese tiempo de evento then
                        Insertarlo usando la función Select;
                endif
          endfor

          e = tn - tl;
        endfor
}
```

Figura 41. Simulación achatada para espacios Cell-DEVS.

El uso de un coordinador achatado es más eficiente que el de uno jerárquico. Por ende, pueden usarse coordinadores jerárquicos durante la fase de desarrollo y chequeo (lo cual es más eficiente desde el punto de vista de debugging), y ejecutar los modelos eficientemente para poder generar resultados con alta velocidad.

4.4 Simulación paralela de un espacio Cell-DEVS.

Para simular sistemas complejos tales como los espacios Cell-DEVS, suele ser necesario ejecutar un alto número de cálculos. Si los modelos se ejecutan secuencialmente, la obtención de una muestra estadísticamente significativa que permita obtener resultados confiables es un proceso que puede consumir demasiado tiempo. Otro problema de una implementación secuencial es que un modelo inherentemente paralelo, tal como el resultante de construir un espacio de celdas, debe ejecutarse de forma serial, provocando no sólo problemas en la eficiencia sino el diseño y el desarrollo de los simuladores. Esto se refleja en la existencia de un coordinador raíz por el que deben pasar todos los mensajes que reflejan eventos en el sistema.

La solución más natural para implementar espacios de celdas, es poner cada aparato de cálculo en un procesador distinto. Desafortunadamente, esta solución, aunque teóricamente factible, en la gran mayoría de los casos no es viable desde el punto de vista de implementación, debido a la alta granularidad del problema a atacar. Por ende, es más adecuado adoptar una solución intermedia, y ejecutar el espacio de celdas dividiéndolo en distintos subprocesos que se ejecutan en múltiples procesadores.

Como fuera mencionado en la sección 2.4, existen diversas soluciones para mejorar el desempeño de las simulaciones por medio de aproximaciones paralelas o distribuídas. Estas soluciones permiten explotar el paralelismo existente en los sistemas reales a ser simulados, asociando más de un procesador a la simulación. Por otro lado, los formalismos presentados en las secciones anteriores subyacen en aproximaciones asincrónicas; lo que hace que su implementación usando algoritmos paralelos también sea directa.

Para mejorar los tiempos de ejecución en simulaciones Cell-DEVS, se propone implementar coordinadores paralelos. En este caso, cada procesador dentro de un conjunto disponible, tendrá asociado un coordinador chato (por cuestiones

de eficiencia). Aquí, nuevamente se puede aprovechar para implementar las soluciones usando una aproximación centralizada y jerárquica durante el desarrollo y chequeo, y una aproximación paralela y achatada durante la colección de resultados, ya que la ejecución de uno u otro mecanismo dependerá de la sobrecarga de las funciones correspondientes en tiempo de ejecución.

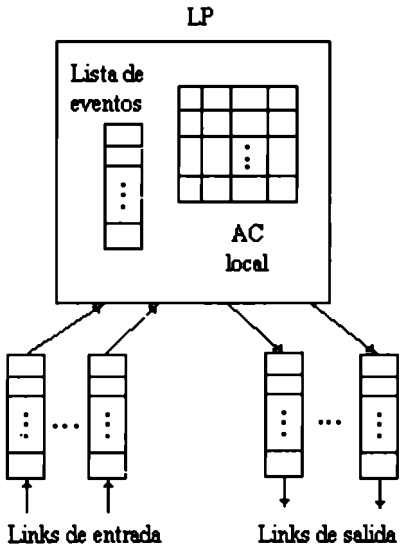


Figura 42. Estructura de un proceso lógico para espacios de celdas.

Se usarán mecanismos de simulación asincrónica, y los procesos lógicos asociados con cada procesador se sincronizarán por medio de aproximaciones optimistas o pesimistas, de tal forma de poder disponer de un entorno configurable, que tenga mayor grado de aceleración según la aplicación a ejecutar. Cada coordinador estará codificado como un proceso lógico incluyendo tres listas de eventos: una lista de eventos local, enlaces de entrada y de salida.

Los principales mecanismos de sincronización serán ejecutados por tres métodos básicos: *Receive_message()*, *Transmit_Message()* y *Execute_Message()*.

Básicamente, cada coordinador elegirá las celdas inminentes para simular, siguiendo el procedimiento achatado analizado en la sección 4.3. Los mensajes **X** que llegan al coordinador pueden ser locales o remotos. Como un resultado de procesar estos mensajes, se generan mensajes **Y** y **listo**, y se transforman de la forma en que fue explicada en la sección 3.3 Si un nuevo evento pertenece a un procesador local, se agrega en la lista de próximos eventos. En cambio, si la celda destino no pertenece al procesador local, el mensaje se agregará en la cola de salida y será retransmitido a otro procesador, donde se agregará como un mensaje de entrada.

La simulación podrá dividirse de acuerdo con la potencia de cada procesador, permitiendo balancear la carga de la simulación. Para ello se usará la función *Map_Tasks()*, que ejecutará algoritmos tradicionales de distribución de la carga. Para este caso, la función *Map_tasks()* almacena en cada coordinador un submapa de celdas indicando a qué procesador pertenece cada porción del espacio. Esta información se almacenará en la variable de instancia *~Processor-mapping*, y será usada para armar los enlaces de entrada y salida.

Como puede verse en la jerarquía de clases de la Figura 37, se considerarán aproximaciones tanto optimistas como pesimistas. Ambas usarán la estrategia presentada recientemente para ejecutar modelos en paralelo. De esta forma, la aproximación elegida puede adaptarse a la aplicación, obteniendo resultados mejorados para cada caso. En todos los casos, los coordinadores locales serán chatos, ya que su ejecución mejora el desempeño. En las siguientes secciones estos métodos se analizan con detalle.

4.4.1 Coordinadores pesimistas

En esta sección se analiza el comportamiento de cada uno de los métodos asociados con los coordinadores conservadores para un entorno Cell-DEVS, cuyo comportamiento se ve en la siguiente figura:

```
Receive_Message() {
        Leer mensaje llegado por un enlace de entrada;
        Almacenarlo en la Lista de Eventos;
        si el mensaje corresponde a un enlace con el menor timestamp entonces
                Unlock();
}

Execute_Message() {
        si el enlace con el menor timestamp no tiene un mensaje en la lista de eventos entonces
            si se usa la estrategia de Mensajes Nulos entonces
                Lookahead = LVT + d;
                Armar un mensaje nulo;
                Enviarlo a los vecinos
            finsi
            Lock(); /* El proceso lógico debe bloquearse en espera del mensaje con menor timestamp */
        finsi

        LVT = hora del primer mensaje en la lista de eventos;
        Para todo mensaje en la cola con timestamp = LVT hacer
                según tipo del mensaje hacer
                    *: ReactionTo*Message();
                       Enviar un mensaje listo al coordinador de nivel superior. Si éste corresponde a otro
                         procesador, encolarlo en el enlace de salida correspondiente.
                             X: ReactionToXMessage();
                       Enviar un mensaje Y al coordinador de nivel superior. Si éste corresponde a otro
                         procesador, encolarlo en el enlace de salida correspondiente;
                       Translate(Y, X);
                       Insertar X en la cola de eventos locales;
                finsegún
        finpara
}

Transmit_Message() {
        Para todo elemento en un enlace de salida hacer
                Enviar el mensaje por el enlace correspondiente;
}
```

Figura 43. Funciones de coordinación pesimista para entornos Cell-DEVS.

En este caso, el método *Receive_Message()* debe guardar un mensaje en la cola de entrada, almacenando su tipo, su timestamp, su origen/destino y su valor, de acuerdo con la clase de mensaje. Los mensajes, cuando llegan, terminan encolados en la lista de eventos.

El método *Execute_Message()* debe procesar todos los eventos en la cola de eventos que tengan timestamps menores a la hora local (LVT), siempre y cuando existan mensajes en las listas de entrada con timestamps mayores al del LVT. En este caso, el proceso lógico se bloquea. Si el proceso lógico está activo, debe hacer un ciclo de activación de las rutinas *ReactionToXMessage()*, y *ReactionTo*Message()*, que activarán las rutinas del coordinador achatado, tal como fuera analizado en la sección 4.3.

En este caso, de acuerdo con la estrategia de manejo de deadlock elegida, deberá activarse el método de *Recovery()* (que detecta y recupera deadlocks), o enviar mensajes nulos. En este caso es simple calcular el lookahead. Suponiendo que todas las celdas tienen una demora fija de **d**, el lookahead será LVT+d.

La función *Transmit_Message()* es activada cuando el coordinador, analizando su mapeo (usando la función *Translate()* que implica activar la función de traducción Z, y usar las listas Xlist e Ylist), decida que el mensaje corresponde a un coordinador que se encuentra en otro procesador. Para ello, debe utilizar nuevamente el *Processor_Mapping*.

4.4.2 Coordinadores optimistas

Aquí, los métodos *Receive_Message()* y *Transmit_Message()* tienen la misma funcionalidad que en el caso anterior, y la diferencia radica en la función *Execute_Message()*.

```
Receive_Message() {
        Leer mensaje llegado por un enlace de entrada;
        Almacenarlo en la Lista de Eventos;
}


Execute_Message() {
  si el primer mensaje en la lista de eventos tiene timestamp menor al LVT entonces
        Rollback();
  finsi

  LVT = hora del primer mensaje.
  si LVT > GVT entonces  /* Colección de fósiles */
        Borrar todos los estados previos al LVT actual de la Lista-de-Espacios y de la Lista-de-Next-Events;
  finsi

  LVT = hora del primer mensaje en la lista de eventos;

  Para todo mensaje en la cola con timestamp = LVT hacer
        según tipo del mensaje hacer
                *:   ReactionTo*Message();
                     Enviar un mensaje listo al coordinador de nivel superior. Si éste corresponde a otro
                       procesador, encolarlo en el enlace de salida correspondiente.

                X:   ReactionToXMessage();
                     Enviar un mensaje Y al coordinador de nivel superior. Si éste corresponde a otro
                       procesador, encolarlo en el enlace de salida correspondiente;
                     Translate(Y, X);
                     Insertar X en la cola de eventos locales;
        finsegún

        si existe un mensaje idéntico al mensaje que se está procesando y se usa CANCELACION_LAZY entonces
           si el mensaje es igual en todo menos en el valor entonces
                     Generar un antimensaje en base al anterior;
                     Encolar el actual;
           finsi
        finsi
  finpara

  Actualizar la Lista-de-Next-Events y la Lista-de-Espacios de celdas con el nuevo estado;
}


Rollback() {

        LVT = hora del dispersor;

        Buscar en la lista de próximos eventos y de espacios de celdas el primer elemento
                cuyo timestamp sea menor al LVT;

        Cells = Lista-de-espacios.Cells;

        Next_Events = Lista-de-Next-Events.lista;
        según TIPO DE CANCELACION hacer
```

```
            PEREZOSA: nada;
            AGRESIVA: generar un antimensaje en la cola de salida;
        finsegún

        Eliminar de la cola de salida los elementos cuyo timestamp es anterior al LVT;
}
```

Figura 44. Funciones de coordinación optimista para entornos Cell-DEVS.

En este caso, se toman los primeros mensajes de la lista de eventos, analizando si el evento arribado es o no un dispersor. Si así fuera, hay que restaurar la lista de eventos al momento del dispersor, así como el estado del espacio de celdas. Para ello, se almacena una lista que incluye todas las listas de Next_Events, y otra lista con todos los espacios de celdas. El dispersor genera antimensajes, y restaura el estado hasta ese momento.

Si se usa cancelación agresiva, la función *Transmit_Message()* debe ser activada cuando el coordinador, analizando su mapeo, decida que el mensaje corresponde a un coordinador que se encuentra en otro procesador. Si se usa cancelación perezosa, la rutina Transmit_Message() elimina elementos seguros de la cola de salida, y no se transmiten antimensajes para los casos de generación de mensajes con igual contenido.

4.4.3 Cuestiones de Desempeño

Como fuera mencionado con anterioridad, el desempeño de los algoritmos de sincronización de PDES depende del grado de lookahead que pueda obtenerse (para las pesimistas), y de las diferencias de avance de la simulación en los distintos procesadores (que puede provocar alto grado de overhead debido a los rollbacks en el caso de las optimistas). En este caso estos parámetros sólo dependerán de la demora de cada celda. Una forma de mejorar el desempeño de estos factores en modelos Cell-DEVS se relaciona con la forma de procesar las listas de eventos de los coordinadores locales. En general, las listas se procesan por medio de distintas tareas que ejecutan concurrentemente. El orden de selección de procesamiento de dichas listas puede determinar influencias importantes en los resultados. Pueden distinguirse varios casos:

- El tiempo simulado en el submodelo local avanza más lentamente en el procesador local que en los procesadores vecinos. Esto , en general, provocará que cuando llegue un mensaje, el proceso lógico tenga un tiempo virtual local inferior al de los timestamps de los mensajes recibidos desde los vecinos. Por ende, la probabilidad de errores de causalidad será baja, y será preferible dar mayor prioridad al procesamiento de la lista de eventos locales. De esta forma, la lista siempre dispondrá de mensajes con orden de procesamiento correcto y no habrá demoras en su tratamiento.

- Si el subespacio de celdas local procesa eventos a mayor velocidad que sus vecinos, los errores de causalidad ocurrirán frecuentemente. En este caso será conveniente dar mayor prioridad al proceso de entrada de nuevos mensajes.

Para detectar estas diferencias de velocidad se puede utilizar información implícita provista por los algoritmos de sincronización. En algoritmos optimistas, la existencia de un alto grado de rollback indica que el proceso local avanza a mayor velocidad que sus vecinos. En algoritmos pesimistas, se dispone de los peores timestamps en todos los enlaces de entrada, información que puede utilizarse para comparar con el reloj local y mejorar dinámicamente el desempeño.

Como fuera mencionado, una cuestión crítica enfrentada por las aproximaciones optimistas está relacionada con el overhead incurrido en el procesamiento de rollbacks. La ocurrencia de rollbacks provoca, por un lado, la explosión de antimensajes, que deben ser tratados. Por otro lado, un mal manejo del GVT trae problemas en cuanto al uso de memoria.

En su primera definición, el comportamiento de los algoritmos de sincronización optimista fue considerado similar al de los usados para administración de memoria virtual en sistemas operativos. La observación de un alto grado de overhead en el sistema es equivalente a un comportamiento de thrashing en sistemas con memoria virtual, pero aquí la mayoría del tiempo de procesamiento de la simulación se pierde en calcular resultados incorrectos, que provocan nuevos rollback.

Considerando estos hechos, se propone emplear una técnica que ha dado buenos resultados en cuanto a la relación overhead/uso de recursos para sistemas de memoria virtual. Esta técnica, llamada "Modelo de Frecuencias de Faltas de Página" [Wul69], permite evitar el thrashing con bajo overhead.

La técnica modificada será llamada Modelo de Frecuencia de Rollbacks, y usará la misma estrategia para estudiar el número de rollbacks para evitar su ocurrencia excesiva. La idea se basa en que cuando el número de Procesos Lógicos activos aumenta, el número de rollbacks también crecerá (exponencialmente), debido a que el número de antimensajes se multiplicará. En cambio, si el número de rollbacks es bajo, puede interpretarse como una señal de que el grado de paralelismo también será bajo, ya que la falta de rollbacks puede indicar un bajo grado de intercambio de mensajes.

Como para una simulación de modelos Cell-DEVS el número de procesos lógicos es fijo, la detección de una frecuencia de rollback alta para un proceso lógico es una indicación de que la simulación ejecutada por el coordinador local debería demorarse. En este caso sólo se aceptarán nuevos mensajes de entrada que se agregarán en la lista de eventos locales (de la misma forma que fuera explicado previamente), pero la lista de eventos locales no será procesada. De esta forma se reduce el número total de procesos lógicos activos y, por ende, en determinado momento el número de rollbacks se reducirá.

Por otro lado, si el número de rollbacks es muy bajo, esto es una indicación de que el procesamiento de la lista de eventos local está avanzando lentamente. Por ende, si existe algún proceso lógico que haya sido demorado, deberá ser reactivado.

El fijar una frecuencia mínima y máxima de rollbacks (como se ve en la Figura 45), permite mantener el paralelismo en un nivel alto, reduciendo el overhead de la simulación. Si el número de rollbacks está por debajo de la cota inferior para cada proceso lógico y ninguno es demorado, el grado de paralelismo no puede aumentar. Por ende, la distribución de la carga debería modificarse. Para ello es necesario contar con algoritmos dinámicos para el balanceo de la carga.

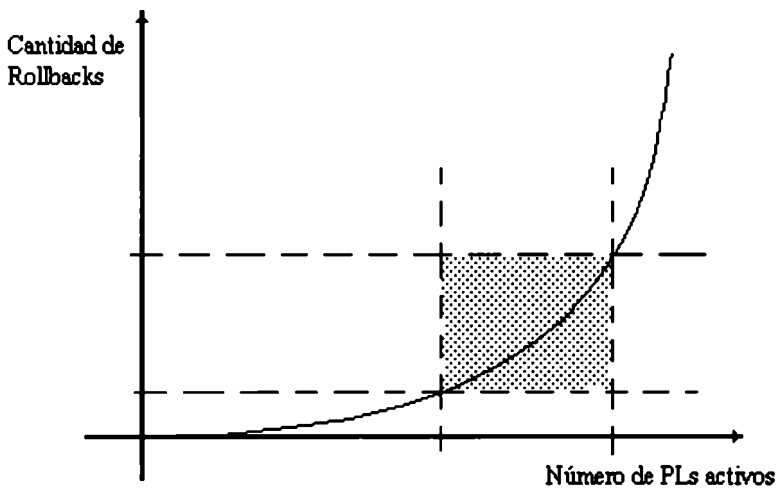


Figura 45 - Modelo de Frecuencia de Rollback.

5. UN EJEMPLO DE MODELIZACIÓN Y SIMULACIÓN CELL-DEVS

En esta sección se muestra un ejemplo detallado de aplicación de los formalismos presentados previamente. La Figura 46 muestra un modelo cuya finalidad es simular el flujo del tráfico automotor en una sección de población urbana. El modelo se compone de varios submodelos, cada uno construido usando distintas aproximaciones. El comportamiento de cada modelo ha sido simplificado imponiéndole diversas restricciones (aunque las reglas presentadas pueden extenderse para representar fenómenos más complejos).

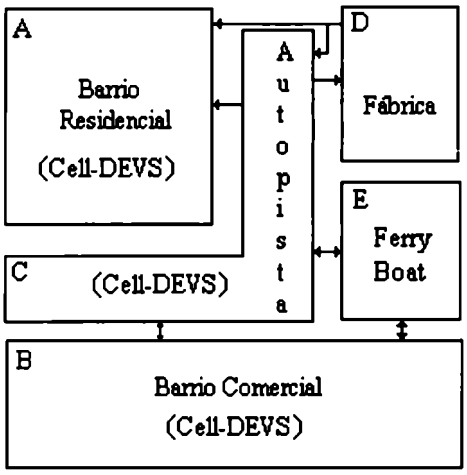


Figura 46. Acoplamiento de modelos Cell-DEVS y DEVS [WG98].

En este caso, A es un modelo Cell-DEVS que representa un barrio residencial. El modelo tiene la finalidad de estudiar valores de polución por medio de un modelo celular que representa partículas de humo en el aire. La influencia del tráfico en la autopista y el humo de los camiones de la fábrica son considerados como generadores de un flujo de polución. La función de cálculo local también debe modelar la influencia del viento sobre el smog. Si una partícula se queda en una celda durante un tiempo determinado, se difunde a sus vecinos; sino, el viento la remueve. Para simplificar el ejemplo, se considerará que un automóvil o un camión generan una cantidad de smog equivalente a la cantidad de partículas necesarias para activar una celda del modelo A. Si la relación entre automotores y polución debiera ser distinta, podría utilizarse un modelo traductor intermedio.

El modelo B representa el flujo de tráfico en un barrio comercial. En este, las calles son de una sola mano (con direcciones Norte o Este), y no se modelan semáforos. Este modelo Cell-DEVS podría ser usado para estudiar el flujo de tráfico y permitir la instalación de semáforos.

El otro modelo Cell-DEVS, llamado C, representa una autopista de mano única que pasa a través de los barrios A y B. El flujo de tráfico en la autopista se representa usando un modelo celular que servirá para estudiar el flujo del tráfico en la autopista y su influencia en el resto de la ciudad. También se considera su influencia en el modelo de análisis de smog, como fuera explicado con anterioridad.

El modelo atómico D representa una fábrica, en la que llegan camiones desde la autopista, y otros camiones salen de la fábrica hacia la autopista. Los resultados de esta simulación podrían utilizarse para planificar la entrada y salida de camiones a la fábrica, mejorando la frecuencia en el despacho de productos. También se considera a los camiones como generadores de polución para el modelo A, como ya fuera analizado.

Finalmente, el modelo atómico E representa la entrada a un Ferry-Boat que conecta la ciudad con una isla. Este modelo podría usarse para estudiar el flujo de tráfico a los ferrys (determinando el número óptimo de botes que deben ser usados dependiendo de la hora).

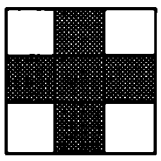
En la siguiente sección se estudia el punto de vista del modelador, analizando la definición de los espacios de celda de este ejemplo usando el lenguaje de especificación presentado en la sección 4.1. Luego se considera le especificación formal de cada uno de los modelos utilizando el formalismo Cell-DEVS. A continuación se incluye una breve

explicación relacionada con el punto de vista del implementador del entorno, estudiando los mecanismos de simulación (jerárquico y achatado) de los modelos. Finalmente, se presenta un análisis de la interacción entre los modelos y los mecanismos de simulación en la herramienta.

5.1 Especificación del comportamiento de las celdas.

A continuación se analizará el comportamiento de los espacios de celdas de la Figura 46. Para ello se describirán las funciones de cálculo locales usando el lenguaje de especificación presentado en la sección 4.1. En las especificaciones que se mencionan a continuación, las variables a_{ij} definen una matriz de 9x9 que representa la vecindad de la celda. Estos valores se toman como precondición de la función de cálculo local, cuyo resultado es el nuevo estado de cada celda.

5.1.1 Modelo A (Smog en el barrio Residencial)



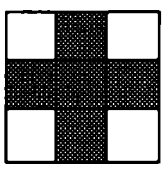
Vecindad: { (a_{12}, a_{21}, a_{22}, a_{23}, a_{32}) };
Select: { (a_{12}, a_{23}, a_{22}, a_{21}, a_{32}) };
viento = 3;
f = 9; c = 10; in_delay(viento).

Result.	Estado de la vecindad
1	(a_{22} = 1 AND NOT (ALL = 0)) OR ((a_{22} = 0) AND (a_{12} = 1 OR a_{21} = 1 OR a_{23} = 1 OR a_{32} = 1))
0	(ALL = 0) AND (a_{22} = 1)

Figura 47. Especificación para el modelo A

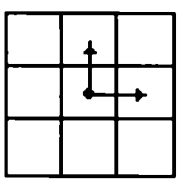
En este caso se considera la difusión del smog en el espacio de celdas. La demora inercial se ha utilizado para modelar la difusión de la polución, de forma tal que si el viento remueve el smog, la polución no se expande. Si hay una entrada con una demora inercial de 3 unidades de tiempo, las partículas se quedan en la celda. Si hay una partícula de smog en una celda vecina, y la partícula se queda allí durante 3 unidades de tiempo, la partícula se expande a la celda actual. De otra forma, se toma en cuenta la influencia del viento, que remueve a los vecinos y deja a la celda en el mismo estado anterior. Ante eventos simultáneos, las celdas se eligen usando la función select, que considera que la dirección del viento es SW.

5.1.2 Modelo B (Tráfico en el barrio comercial)



Vecindad: { (a_{12}, a_{21}, a_{22} ,a_{23}, a_{32}) };
Select: { (a_{32}, a_{21}, a_{22}, a_{12}, a_{23}) };
velocidad = 5;
f = 50; c = 10;
tr_delay(velocidad).

(a)



(b)

Result.	Estado de la vecindad
1	(a_{22} = 0 AND a_{32} = 1) /* Avanzo en dirección Norte*/ OR (a_{22} = 0 AND a_{21} = 1)) /* Avanzo en dirección Este */ OR (a_{22} = 1 AND a_{23} = 1) /* Bloqueado en dirección Este */ OR (a_{22} = 1 AND a_{32} = 1) /* Bloqueado en dirección Norte*/
0	(a_{22} = 1 AND a_{12} = 0) /* Avanzo en dirección Norte */ OR (a_{22} = 1 AND a_{23} = 0) /* Avanzo en dirección Este */ OR (a_{22} = 0 AND a_{32} = 0 AND a_{21} = 0)

Figura 48. Especificación para el modelo B. (a) Parámetros del espacio de celdas. (b) Descripción del movimiento de los automotores.

La especificación del modelo B representa el movimiento de los automotores en calles de mano única en el barrio comercial. La primera regla de la especificación representa la entrada de un nuevo automotor en una celda vacía, o un automotor frenado que no se puede mover de esa celda. La segunda regla representa el desplazamiento de un automotor (partida del automotor de la celda de origen), como puede verse en la
Figura *48* (b). Una celda también queda sin automotor si hay un espacio rodeado por celdas vacías.

Puede verse que el desplazamiento de un automotor no se registra explícitamente en una regla única, sino que el cambio de estado se divide en dos movimientos diferentes. El mecanismo de simulación expandirá el primer movimiento a las celdas influenciadas, que registrarán el cambio. El cambio de estado se expandirá a todos los vecinos, pero sólo tendrá influencia en aquellos donde el movimiento sea válido.

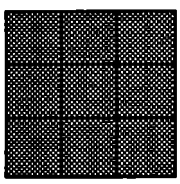
La primera fase de un movimiento se realiza en la celda de origen: si en ella hay un automotor que pueda moverse, el estado cambiará. Como el automotor será removido de la celda, el cambio de estado será detectado por el mecanismo de simulación, y las celdas inminentes serán activadas.

Cuando la celda destino sea activada, analizará los valores de los vecinos. Como para la celda destino el movimiento es válido, cambiará su estado registrando la llegada del nuevo automotor. Si el estado cambia de 1 a 0, los influenciados se activarán y la función de cálculo local será ejecutada, provocando un nuevo movimiento del automotor.

Si hay un cruce, el automotor en la derecha pasa primero porque el flujo hacia el norte es el primero en ser evaluado. También se evita la colisión en la simulación por medio del uso de la función Select.

Las demoras de transporte se usan para modelar las demoras provocadas por la aceleración de los automotores. El movimiento de los mismos se demora antes del siguiente movimiento a la próxima celda. Si este valor se genera con una variable aleatoria, se podrán modelar distintas velocidades para cada uno de los automotores en distinto momento.

5.1.3 Modelo C (Autopista)



Vecindad: { (a_{11}, a_{12}, ..., a_{33}) };
Select = { a_{23}, a_{13}, a_{33}, a_{22}, a_{12}, a_{32}, a_{11}, a_{21}, a_{31} };
velocidad = 2;
f = 15; c = 4;
tr_delay(velocidad).

Result.	Estado de la vecindad
1	/* Flujo normal */ (a_{22} = 0 AND a_{21} = 1) OR /* Sobrepasando por la izquierda */ (a_{22} = 0 AND a_{21} = 0 AND a_{31} = 1 AND a_{32} = 1) OR /* Sobrepasando por la derecha */ (a_{22} = 0 AND a_{21} = 0 AND a_{11} = 1 AND a_{12} = 1) OR /* Embotellamiento */ (a_{22} = 1 AND Column$_3$ = 1)
0	/* Flujo normal */ (a_{22} = 1 AND a_{23} = 0) /* Sobrepasando por la izquierda */ OR (a_{22} = 1 AND a_{23} = 1 AND a_{13} = 0 AND a_{12} = 0) /* Sobrepasando por la derecha */ OR (a_{22} = 1 AND a_{23} = 1 AND a_{33} = 0 AND a_{13} = 1 AND a_{32} = 0) /* Celda vacía no considerada en la regla 1*/ OR (a_{22} = 0)

Figura 49. Especificación del modelo C.

Esta especificación se utiliza para representar una autopista de una sola mano con cuatro carriles. En este caso el movimiento de los automotores también se modela dividiéndolo en dos fases distintas. La primer regla incluida

representa la llegada de un nuevo automotor a la celda actual, o un automotor demorado en una celda. Cada una de las subexpresiones de la regla representa (en orden) los siguiente movimientos:

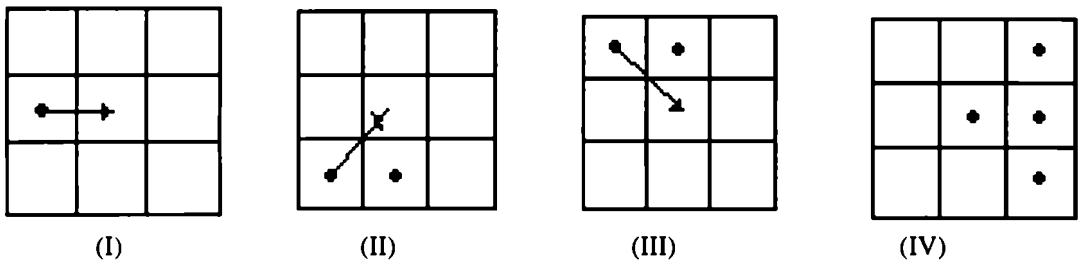


Figura 50. Movimientos válidos para la regla 1 (distintas expresiones).

La segunda regla refleja los siguientes movimientos:

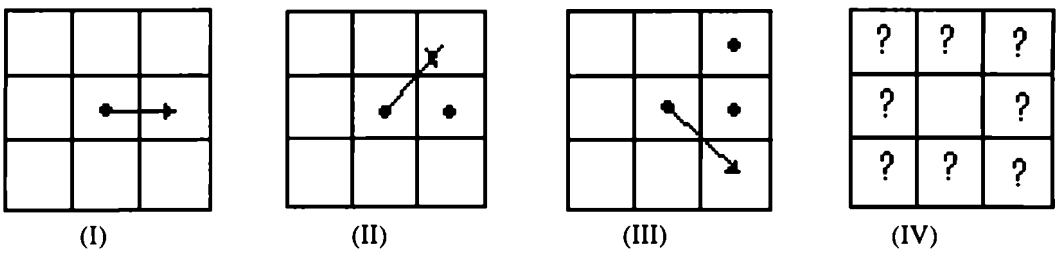


Figura 51. Movimientos válidos para la regla 2 (distintas expresiones).

5.2 Especificación formal de los modelos

En esta sección se considerarán las definiciones de las especificaciones formales de cada uno de los modelos presentados. Inicialmente se considerarán los dos submodelos DEVS tradicionales. En ambos casos, serán construidos como servidores de colas simulando la llegada y partida de automóviles. Los valores generarán usando números aleatorios basados en una distribución de Poisson.

El modelo D, que representa el flujo de los camiones en la fábrica puede especificarse como:

$$M_D = \langle I_D, X_D, S_D, Y_D, \delta_{intD}, \delta_{extD}, D_D \rangle$$

I_D = { (IN_D, binario), (OUT_D, binario), ($SMOG_D$, binario)};
$X_D = Y_D$ = {0, 1};

S_D:
Variables descriptivas:

Truck_q (entero positivo): Número de camiones esperando en la fábrica;
Camiones_saliendo (entero positivo): número de camiones que salen de la fábrica.

Parámetros:

Load_time (entero positivo): tiempo dedicado a llenar un camión. Tiempo promedio: 100 minutos.

```
δext_D(s,e,x) {

cuando x se recibe en el port IN_D

  case phase

        pasiva:
                Load_time = random(100);
                phase = activa;
                hold_in(Load_time);        /* Activar el modelo. Simular la carga del camión */

        activa:
                Truck_q = Truck_q + 1;     /* Aumentar el número de camiones en la fábrica */
                continue;                  /* y continuar con el avance del tiempo */
endcase

}


d_intD(s,e) {

  case phase

        activa:
                Truck_q = Truck_q - 1;     /* Parte el camión */

                if (Truck_q = 0)
                        passivate;
                else
                        Load_time = random(100);
                        hold_in(Load_time);
                endif

        pasiva:
                /* Nunca ocurre */

endcase

}


λ (s) {
        send Truck_q to OUT_D
        send 1 to SMOG_D

}
```

Figura 52. Definición del modelo D.

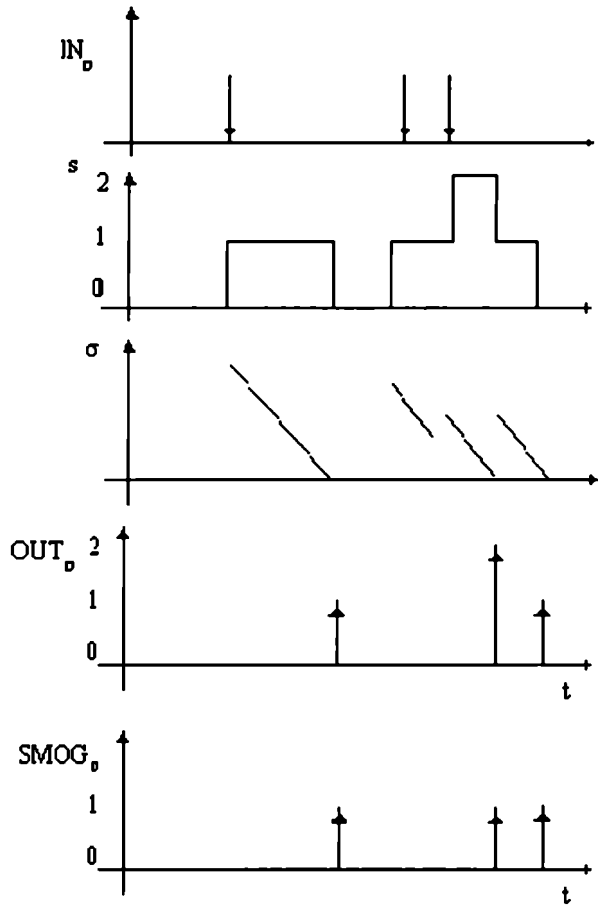


Figura 53. Ejemplo de ejecución del modelo D.

La descripción del ferry boat puede especificarse como:

$$M_E = \langle I_E, X_E, S_E, Y_E, \delta_{intE}, \delta_{extE}, D_E \rangle$$

I_E = { (IN_E, binario), (OUT_E, binario) };
$X_E = Y_E = \{0, 1\}$;

S_E:
Variables descriptivas:

Cola_automotores (entero positivo): Número de automotores esperando entrar al barco.

Parámetros:

Load_time (entero positivo): tiempo dedicado en cargar el Ferry. Tiempo promedio: 30 minutos.

```
δext_E(s,e,x) {

cuando x se recibe en el port IN_E
  case phase
        pasiva:
                Load_time = random(30);
                phase = activa;
                hold_in(Load_time);    /* Activar el modelo. Simular la carga de un automotor */
        activa:
                Cola_automotores = Cola_automotores + 1;  /* Aumentar el número de automotores en el ferry */
                continue;                /* y continuar con el avance del tiempo */
endcase
}

d_intE(s,e) {
  case phase
        activa:
                Cola_automotores = Cola_automotores + 1;;  /* El automotor está cargado */

                if (Cola_automotores = 0)
                        passivate;
                else
                        Load_time = random(30);
                        hold_in(Load_time);
                endif
        pasiva:
                /* Nunca ocurre */
endcase
}

λ (s) {
        send Cola_automotores to OUT_E
}
```

Figura 54. Definición del modelo E.

En este caso, las variables aleatorias podrían ser generadas con un tiempo de servicio promedio de 30 minutos.

Consideremos ahora la definición del espacio de celdas A completo. En la Figura 55 podemos ver un esbozo detallado de este modelo.

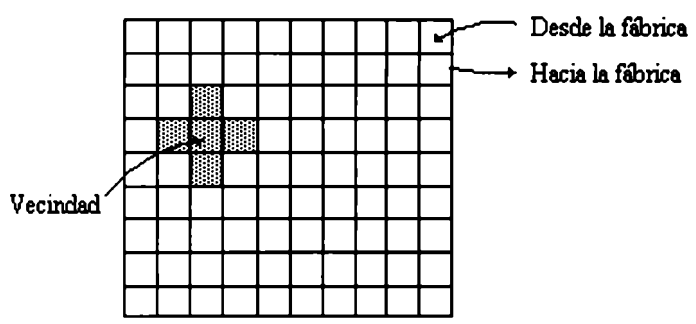


Figura 55. Espacio de celdas A; celdas de entrada/salida del modelo.

La definición formal para este espacio de celdas es:

$$M_A = < I_A, X_A, Y_A, Xlist_A, Ylist_A, \eta_A, N_A, \{m_A, n_A\}, C_A, B_A, Z_A, select_A>$$

De acuerdo con la definición previa de interfaces y nombres de ports,

$\mathbf{Xlist_A}$ = { (1,10); (2,10) };
$\mathbf{Ylist_A}$ = { ∅ }.

$\eta_A = 5$;

$\mathbf{I_A} = < P^X, P^Y>$, donde

P^X = { <X(1,10), binario>, <X(2,10), binario> };
P^Y = { ∅ };

$\mathbf{N_A}$ = { (0,0), (-1,0), (1, 0), (0,1), (0,-1) };

$X_A = Y_A = \{0, 1\}$;

$\mathbf{m_A} = 9$; $\mathbf{n_A} = 10$;

$\mathbf{B_A} = \{\emptyset\}$;

$\mathbf{C_A}$ es el espacio de celdas definido como $C_A = \{C_{Aij} / i \in [1,9], j \in [1,10]\}$, donde cada C_{Aij} es un componente Cell-DEVS tal como los definidos en la sección 3.2. La especificación para cada celda será descripta más adelante.

$\mathbf{Z_A}$ se define como en la sección 3.3. En este caso, si se consideran la lista de vecindad N y la vecindad inversa, la interconexión de los ports para la función $\mathbf{Z_A}$ se definirá como:

$P_{ij}\ Y_1 \rightarrow P_{i,j-1}\ X_1$
$P_{ij}\ Y_2 \rightarrow P_{i+1,j}\ X_2$
$P_{ij}\ Y_3 \rightarrow P_{i,j+1}\ X_3$
$P_{ij}\ Y_4 \rightarrow P_{i-1,j}\ X_4$
$P_{ij}\ Y_5 \rightarrow P_{ij}\ X_5$

$P_{i,j+1}\ Y_1 \rightarrow P_{ij}\ X_1$
$P_{i-1,j}\ Y_2 \rightarrow P_{ij}\ X_2$
$P_{i,j-1}\ Y_3 \rightarrow P_{ij}\ X_3$
$P_{i+1,j}\ Y_4 \rightarrow P_{ij}\ X_4$
$P_{ij}\ Y_5 \rightarrow P_{ij}\ X_5$

Figura 56. Interconexión de ports para la función Z_A.

$\mathbf{select_A}$ = { (-1,0), (0,1), (0,0), (0,-1), (1, 0) };

Cada celda en el espacio puede describirse formalmente como:

$$C_{Aij} = < I_A, X_A, S_A, Y_A, N_A, d_{intA}, \delta_{extA}, d_A, \tau_A, \lambda_A, D_A >$$

con

$\mathbf{I_A} = < \eta_A, P^X, P^Y>$, donde

$\eta_A = 5$;
P^X = { (X_1, binario), (X_2, binario), (X_3, binario), (X_4, binario), (X_5, binario) };
P^Y = { (Y_1, binario), (Y_2, binario), (Y_3, binario), (Y_4, binario), (Y_5, binario) }.

$X_A = Y_A = \{0, 1\}$;

$\mathbf{S_A}$:
Variables descriptivas:

$$s = \begin{cases} 1 \text{ si hay smog en la celda;} \\ 0 \text{ sino.} \end{cases}$$

Parámetros:

viento: demora relacionada con la influencia del viento.

$\mathbf{N_A}$ = { (0,0), (-1,0), (1, 0), (0,1), (0,-1) };

$\mathbf{d_A}$ = viento;

λ_A, δ_{intA} y δ_{extA} se comportan como las funciones definidas en la sección 5.1.1.

$\mathbf{t_A}$: $S_A \times N_A \rightarrow S_A$ se ha definido previamente usando el lenguaje de especificación.

Ahora, supongamos que el autómata celular de la autopista tiene la siguiente forma:

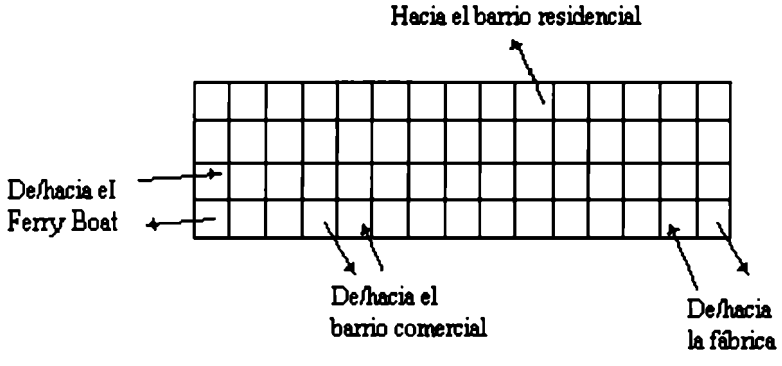


Figura 57. Espacio de celdas C; celdas de entrada/salida del modelo.

La definición formal del espacio de celdas es:

$$C = < I_C, X_C, Y_C, \text{Xlist}_C, \text{Ylist}_C, \eta_C, N_C, \{m_C, n_C\}, C_C, B_C, Z_C, \text{select}_C >$$

En este caso,

$\mathbf{Xlist_C}$ = { (4,14), (4,5), (3, 4) };
$\mathbf{Ylist_C}$ = { (1,10), (4,4), (4,15), (4,1) }.

$\mathbf{I_C} = < P^X, P^Y >$, donde

P^X = {<X(1,3), binario>,<X(4,14), binario>,<X(4,5), binario>,<X(4,3), binario> };
P^Y = {<Y(1,10), binario>,<Y(4,4), binario>,<Y(4,15), binario>,<Y(4,1), binario> }.

$\eta_C = 9$;

$\mathbf{N_C}$ = { (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1, 0), (1, 1) };
$X_C = Y_C = \{0, 1\}$;

$\mathbf{m_C} = 4$; $\mathbf{n_C} = 15$;
$\mathbf{B_C} = \{\varnothing\}$;

C_C es el conjunto del espacio de celdas, definido como en el ejemplo anterior.

S_C:
Variables descriptivas:

{ 1 si hay un automotor en la celda;
{ 0 sino.

Valor inicial: s = 0.

Parámetros:

VelocidadH: demora relacionada con la velocidad de los automotores.

Z_C también se define como en el ejemplo anterior.

select$_C$ = { (0,1), (-1,1), (1, 1), (0,0), (-1,0), (1, 0), (-1,-1), (0,-1), (1,-1) };

La definición para el modelo B es similar que para los dos casos anteriores. La definición de la interfaz del modelo se incluye a continuación con el objetivo de analizar el acoplamiento entre los distintos modelos.

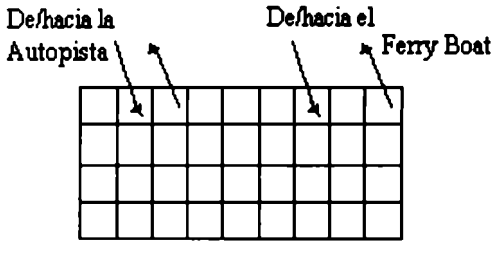


Figura 58. Espacio de celdas B; celdas de entrada/salida del modelo.

En este caso,

$$B = < I_B, X_B, Y_B, Xlist_B, Ylist_B, \eta_B, N_B, \{m_B, n_B\}, C_B, B_B, Z_B, select_B>$$

con

Xlist$_B$ = { (1,2), (1, 7) };
Ylist$_B$ = { (1,3), (1, 9) }.

I$_B$ = < P^X, P^Y>, donde

P^X = { <X(1,2), binario>, <X(1,7), binario>};
P^Y = { <Y(1,3), binario>, <Y(1, 9), binario> }.

$\eta_B = 9$;

N$_B$ = { (0,0), (-1,0), (1, 0), (0,1), (0,-1) };

m$_B$ = 4; **n$_B$** = 9;

select$_B$ = { (1, 0), (0,-1), (0,0), (-1,0), (0,1) };

Finalmente, consideremos la especificación formal para el modelo completo. En este caso,

$$M = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, \text{select} >$$

donde

$X = Y = \{\varnothing\}$;

D = { A, B, C, D, E }, y $\forall i \in D$, M_i es uno de los modelos DEVS básicos definidos previamente.

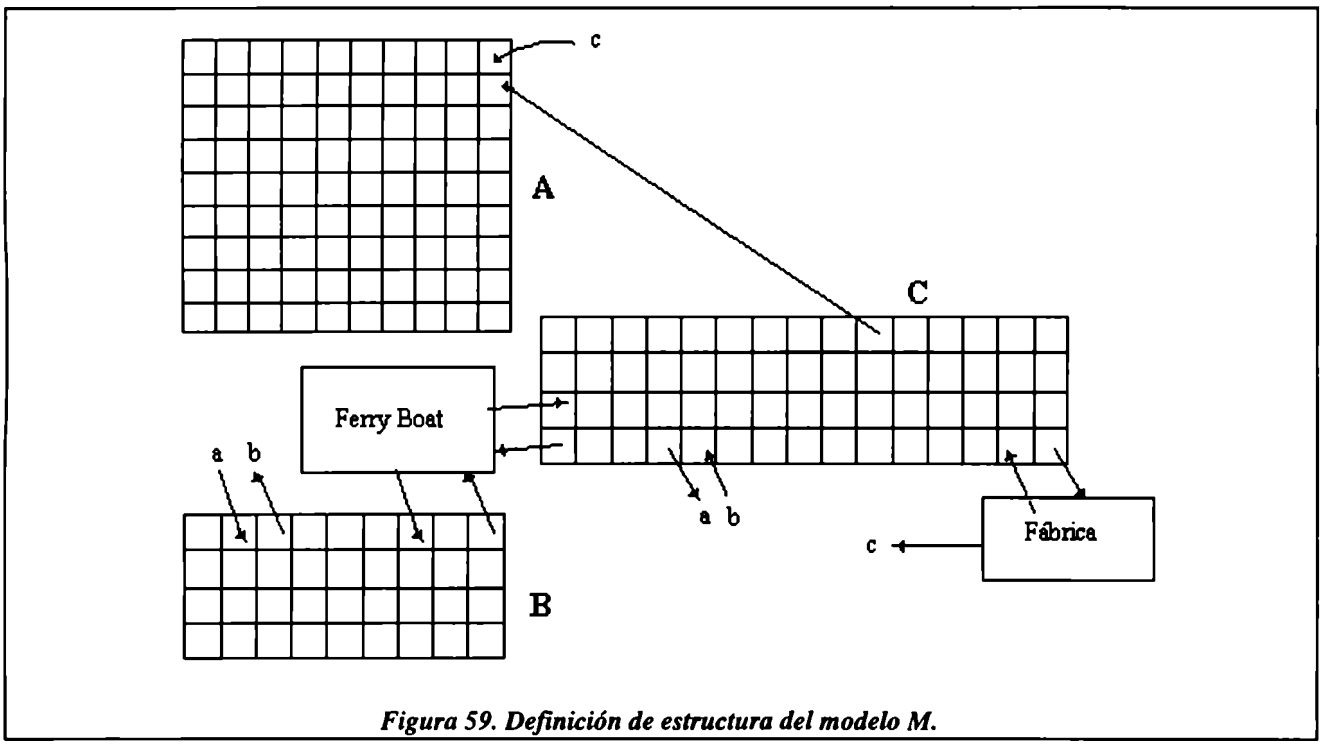


Figura 59. Definición de estructura del modelo M.

$\mathbf{I_i}$ es el conjunto de influencias del modelo i. En este caso,

$I_A = \{\varnothing\}$;
$I_B = \{ C, E \}$;
$I_C = \{ A, B, D, E \}$;
$I_D = \{ A, C \}$; y
$I_E = \{ B, C \}$.

$\forall j \in I_i$, $\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ if i y j son modelos DEVS, o
$\mathbf{Z_{ij}}$: $Y(f,g)_i \rightarrow X(k,l)_j$, con $(f,g) \in Ylist_i$, y $(k,l) \in Xlist_j$ si alguno de los modelos es Cell-DEVS.

En este caso, la función $\mathbf{Z_{ij}}$ se define como:
Z_{BC}: $Y(1,3)_B \rightarrow X(4,5)_C$
Z_{BE}: $Y(1,9)_B \rightarrow IN_f$
Z_{CA}: $Y(1,10)_C \rightarrow X(2,10)_A$
Z_{CB}: $Y(4,4)C \rightarrow X(1,2)_B$
Z_{CD}: $Y(4,15)_C \rightarrow IN_f$
Z_{CE}: $Y(4,1)_C \rightarrow IN_b$
Z_{DA}: $OUT_f \rightarrow X(1,10)_A$
Z_{DC}: $OUT_f \rightarrow X(4,14)_C$
Z_{EB}: $OUT_b \rightarrow X(1,7)_B$
Z_{EC}: $OUT_b \rightarrow X(3,4)_C$

Finalmente, **select** = { C, A, B, D, E}.

5.3 Mecanismo de simulación

Esta sección está dedicada a analizar el comportamiento de los mecanismos empleados en simulación de espacios de celdas usando el ejemplo definido en las secciones anteriores. Primero se ejemplifican los mecanismos de simulación jerárquica, y luego se atacan los procedimientos de simulación achatada.

5.3.1 Simulación jerárquica de los espacios de celdas.

La Figura 60 muestra una sección de la jerarquía de procesadores de simulación para el ejemplo presentado. Como se muestra en la sección 4.2, los simuladores para los espacios de celdas A, B y C se crean automáticamente cuando se define el espacio de celdas. Para ello, el modelador debería definir todos los parámetros presentados en las secciones anteriores.

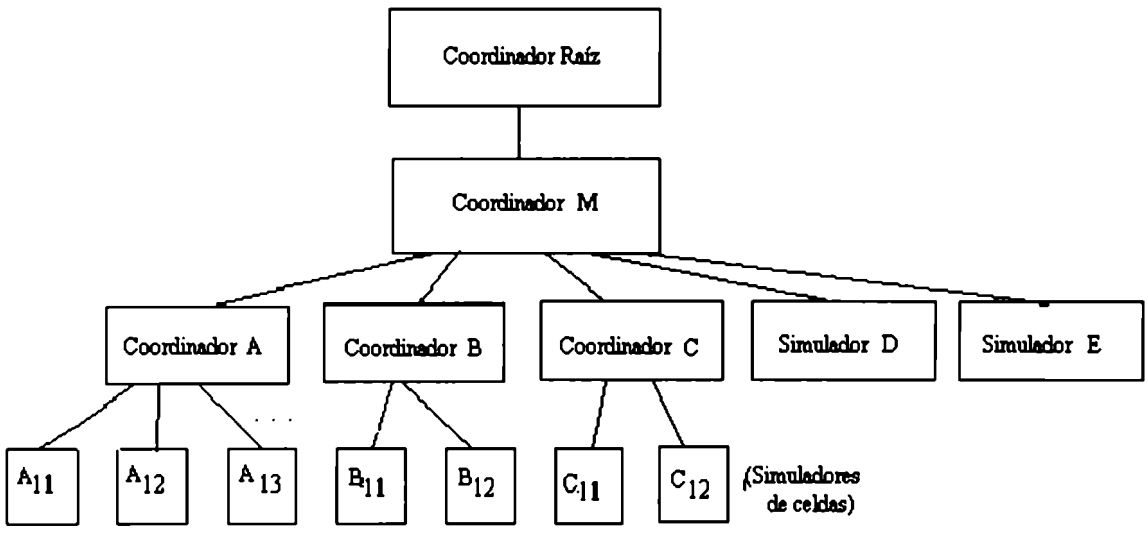


Figura 60. Jerarquía de procesadores de simulación para el modelo M.

En esta sección se estudia el flujo de mensajes entre los distintos procesadores que se muestran en la figura. Para comenzar, se supone que en el instante simulado 10, los contenidos del modelo son los de la Figura 61. Sólo existe una celda activa en el modelo: la celda (1, 9) del modelo C.

Notación: de aquí en más, X(i,j) se refiere a la ij-ésima celda del modelo X o su simulador correspondiente.

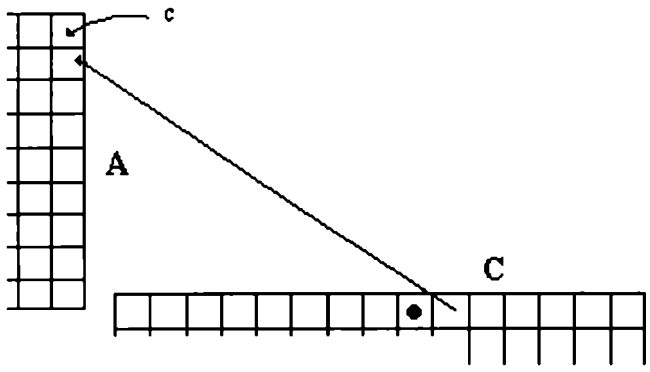


Figura 61. Estado inicial del modelo M.

Los contenidos de las estructuras de datos de los simuladores y coordinadores son los siguientes:

Coordinador Raíz: **Clock** = 10; **Coordinador asociado**: Coordinador M.	Coordinador M **Padre**: Coordinador Raíz **Hijos**: { A, B, C, D, E } **Modelo asociado:** {M} **Lista de Espera**: { } **Hijos inminentes:** { C } **tn:** 10; **tl:** 0.
Coordinador A **Padre**: Coordinador M **Hijos**: { A_{11}, A_{12}, ..., $A_{10,10}$ } **Modelo asociado:** {A} **Lista de Espera**: { } **Hijos inminentes:** { } **tn:** 0; **tl:** 0.	Coordinador B **Padre**: Coordinador M **Hijos**: { B_{11}, B_{12}, ..., B_{49} } **Modelo asociado:** {B} **Lista de Espera**: { } **Hijos inminentes:** { } **tn:** 0; **tl:** 0.
Coordinador C **Padre**: Coordinador M **Hijos**: { C_{11}, C_{12}, ... , $C_{4,15}$} **Modelo asociado:** {C} **Lista de Espera**: { } **Hijos inminentes:** {(1,9) } **tn:** 10; **tl:** 0.	Simulador D **Padre**: Coordinador M **Modelo asociado:** {D} **tn:** 0; **tl:** 0.
Simulador E **Padre**: Coordinador M **Modelo asociado:** {E } **tn:** 0; **tl:** 0.	Simulador A_{ij} **Padre**: Coordinador A **Modelo asociado:** { A_{ij} } **tn:** ∞; **tl:** 0.
Simulador B_{ij} **Padre**: Coordinador B **Modelo asociado:** { B_{ij} } **tn:** ∞; **tl:** 0.	Simulador C_{ij} (Excepto C_{19}) **Padre**: Coordinador C **Modelo asociado:** { C_{ij} } **tn:** ∞; **tl:** 0.
Simulador C_{19} **Padre**: Coordinador C **Modelo asociado:** { C_{19} } **tn:** 10; **tl:** 0.	

Figura 62. Contenidos iniciales de las estructuras de datos de los Simuladores/Coordinadores.

Volviendo a considerar los contenidos de las estructuras de datos de los procesadores, que fueran previamente analizados en la sección 4.2, cada procesador almacenará la siguiente información:

- Clock: pertenece al coordinador raíz, y se utiliza para almacenar la hora global de la simulación;
- Modelo asociado: se utiliza para realizar el apareo entre un modelo y su procesador;
- Padre: almacena un puntero al procesador padre, con el fin de poder intercambiar mensajes con el mismo;
- Hijos: contiene una lista con los procesos hijos, con el fin de intercambiar mensajes;
- Lista de espera: almacena información de los hijos inminentes que aún no han sido activados;
- tl: hora del último evento tratado por el procesador;
- tn: hora del próximo evento a ser tratado por un procesador.

En el estado inicial que será considerado en este ejemplo sólo existe un mensaje esperando para ser procesado en la Event-List del coordinador raíz.

Como fuera analizado en la sección 4.2, los contenidos de un mensaje son los siguientes:

< Tipo, Fuente I Destino, Hora, [Valor] >

El primer paso de simulación a ser considerado a partir del instante actual, consiste en que el coordinador Raíz cree el siguiente mensaje:

< *, Root, 10 >

Este mensaje se envía al coordinador M, que consulta su lista de hijos inminentes. Allí el coordinador selecciona al modelo C (el primero de su lista de hijos inminentes), y envía el mensaje

<*, M, 10>

a su coordinador.

El coordinador C recibe este mensaje, y selecciona su hijo inminente consultando su lista de hijos inminentes (en este caso, la celda [1,9]). También verifica que la hora del próximo evento sea igual a la hora simulada incluida en el mensaje. El mensaje

<*, C, 10>

se envía y llega al Simulador C_{19}, que debe ejecutar la función de transición interna. Como primer medida, todos los procesadores chequean la hora de próximo evento (tn). Como para todos los casos esta hora fue la misma que la recibida en el mensaje *, puede determinarse que el mensaje llegado es correcto. En el simulador debe ejecutarse la función de salida, activando a la función de cálculo local.

Analizando la especificación descripta en la sección 5.1.3, puede verse que la segunda regla de la segunda expresión es verdadera. Por ende, el resultado del cálculo local será s'=0. Como s=1, el estado de la celda ha cambiado, y deberá transmitirse su valor actual. Con este fin, a continuación el simulador genera un mensaje Y conteniendo los siguientes valores:

< Y, C(1,9), 10, 0 >

seguido por un mensaje

< **listo**, C(1,9), 10 >

Este mensaje indica que la transición interna ha sido ejecutada. El mensaje se transmite al coordinador C. Al recibir este mensaje, el coordinador actualiza los tiempos de los eventos:

tl = tn = 10;
tn = tl + D(s) = 10 + 0 = 10.

Lo mismo sucederá para el simulador, que además debe actualizar los valores locales de la vecindad haciendo $a_{22} = 0$ en los valores de los vecinos de la celda C_{19}.

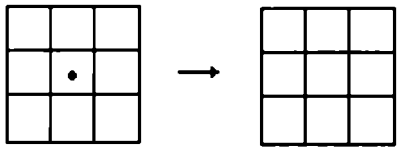


Figura 63. Actualización de los valores de la vecindad local.

El mensaje Y es recibido por el coordinador C, será traducido y retransmitido. Para ello, el coordinador consulta su esquema de acoplamiento (definido en la sección 4.2. Como este mensaje es local al modelo C, debe traducirse a un mensaje X, que debe ser enviado a las siguientes celdas: (1,10), (1,8), (2,8), (2,9), (2,10), (4,8), (4,9), (4,10). El mensaje

$$<X, C(1,9), 10, 0>$$

será enviado a cada uno de los simuladores de celdas, y los números de celda se agregan en la lista de espera del coordinador.

En cada Simulador se chequea la hora del mensaje para ver si está entre los tiempos del último evento y del siguiente evento. Como esto es verdad, se actualizan el tiempo transcurrido y las horas de los eventos, de la siguiente forma:

e = t - tl, donde t es la hora actual acarreada en el mensaje, por ende, e = 10 - 0 = 10;
σ = tn - e = 10 - 10 = 0;
tl = tn = 10; y
tn = tl + D(s) = 10 + 2 = 12.

D(s) = 2, porque la demora de transporte es de 2 unidades de tiempo. Por ende, se planifica el siguiente evento interno a la hora 12.

A continuación se deberán ejecutar cada una de las funciones de entrada. En este caso se encolarán los valores de entrada y sus correspondientes σ en la cola σqueue, como fuera analizado en la sección 3.2. En este caso, D(s) incluirá el valor de las demoras de transporte o inerciales.

Luego, cada Simulador creará un mensaje

$$<\textbf{listo}, C(i,j), 12>$$

Cuando cada uno de estos mensajes sean recibidos por el coordinador C, se eliminarán los hijos de la lista de espera, y se agregarán en la lista de hijos inminentes.

Cuando la lista de espera está vacía, todos los influenciados han finalizado la ejecución de la función de transición interna. Luego, el menor de los hijos inminentes está listo para ser ejecutado. En este caso, todos los mensajes tienen la misma hora, por ende se utiliza la función select para elegir al inminente (como fuera mencionado, los mensajes se encolan usando este criterio). Por ende, para el coordinador C, la lista de hijos inminentes pasará a contener los siguientes valores:

Hijos inminentes: { (1,10), (1,8), (2,8), (2,9), (2,10), (4,8), (4,9), (4,10) }.

A continuación se elige el primer elemento de la lista de hijos inminentes, y se envía el mensaje

$$<\textbf{listo}, C(1,10), 12>$$

al coordinador M. De esta forma, si existiera más de un modelo inminente en la jerarquía, el hijo inminente de todo el modelo puede ser elegido para ser simulado. Al recibir el mensaje, el coordinador M agrega al modelo C en su cola de hijos inminentes. En este caso, como es el único mensaje en la cola, será el hijo inminente. Las horas de los último y próximo eventos pueden ser actualizadas, haciendo tn = 12, y tl = 10. Luego se envía un mensaje **listo** al coordinador Raíz, indicando que el coordinador no está esperando otros eventos (la lista de espera está vacía).

El Coordinador Raíz actualiza el reloj a 12, y crea un nuevo mensaje:

$$<*, Root, 12>$$

Cuando el coordinador M recibe este mensaje, verifica que la hora sea correcta, toma el hijo inminente (C) de su cola de hijos inminentes, y le envía el mensaje

$$<*, M, 12>$$

a su Coordinador. Este Coordinador selecciona su hijo inminente, que en este caso es el (1,10), y activa su Simulador.

Para esta celda, la primer regla de la primer sentencia en la especificación es válida. Por ende, el estado de la celda cambia de cero a uno, se ejecuta λ, y se generan dos mensajes:

$$< Y, C(1,10), 12, 1 >$$

$$< \textbf{listo}, C(1,10), 12 >$$

que se envían al coordinador C.

El mensaje Y se traduce y se expande a las celdas vecinas repitiendo los procedimientos presentados recientemente. Esto provocará actualización en las copias locales de las vecindades de cada simulador. El mensaje **listo**, en cambio, provoca que se elimine el hijo (1,10) de la lista de hijos inminentes.

En este caso, el mensaje Y también influencia los enlaces que son externos al modelo C. Luego, debería ser retransmitido al coordinador padre M para ejecutar la función Z_{ij}, permitiendo enviar el cambio a los otros modelos.

Cuando el mensaje es recibido por el coordinador M, consulta la función de acoplamiento externa y determina que debería utilizarse el acoplamiento externo

$$Y(1,10)_C \rightarrow X(2,10)_A$$

Por ende, genera el siguiente mensaje:

$$< X, A(2,10), 12, 1 >$$

El mensaje se envía al coordinador A, que lo expande hasta que llega a la celda A(2,10), donde se ejecuta la función de transición externa y el mensaje se encola, generando un mensaje

$$<\textbf{listo}, A(2,10), 15>$$

A continuación, se repiten estos procedimientos para cada uno de los hijos inminentes del coordinador C. Para cada una de las celdas inminentes, al activarse los simuladores, se ejecutarán las funciones de cálculo local. Estas celdas, que fueron activadas por el evento en (1,9), quedarán en un estado latente. Esto ocurre porque, al analizar la especificación de la ***Figura 57***, la única regla válida para estas celdas es la cuarta de la segunda sentencia. Luego, el valor del estado quedará en 0, y no se generarán nuevos eventos externos.

Luego, para cada una de estas celdas se genera un mensaje

$$< \textbf{listo}, C(i,j), \infty>$$

que se transmite al coordinador C.

5.3.2 Simulación de espacios de celdas achatados.

En esta sección se repetirá la ejecución del ejemplo de la sección 5.3.1, pero considerando un coordinador achatado. En este caso, se evita por completo la interacción de mensajes entre los espacios de celdas, permitiendo la mejora de los tiempos de ejecución.

Cuando consideramos el procedimiento de la sección 4.3 para ejecutar este ejemplo, podemos ver que las listas de próximos eventos de todos los modelos de celdas están vacías, excepto para el caso del modelo C. Para este Coordinador se obtiene:

$$\text{Next-Events} = \{(1,9), 10\}$$

Al llegar un mensaje *, el coordinador se activa, y comienza su ejecución, tomando los datos del primer evento de la lista. Como primer medida, se actualizan los tiempos de simulación:

tl = 0, tn = 10 y e = 0.

En este caso, cuando se ejecuta la función de cálculo local, el resultado obtenido es New-state = 0. Como Cells[1,9].state = 1, el nuevo estado debe ser almacenado en la lista New-States. Por ende,

New-states = { (1,9), 0}

Luego, como la celda no está en la Ylist, ni se usan demoras inerciales, se ejecutan las siguientes sentencias

Next-event time = 10+2;
Next-events = { <(1,10), 12>, <(1,10), 12>, <(1,8), 12>, <(2,8), 12>, <(2,9), 12>,
<(2,10), 12>, <(4,8), 12>, <(4,9), 12>, <(4,10), 12> }.

La inserción en la lista de próximos eventos se hace usando la función select, debido a que todos los eventos tienen la misma hora.

Luego, como la lista de próximos eventos no tiene otros eventos con hora = 10, ha terminado un paso de la simulación. Por lo tanto, el espacio de celdas se actualiza haciendo:

c[1,9] = 0;
New-states = { }.

Cuando la lista de próximos eventos se ha actualizado, se envía un mensaje

< **listo**, 10, C >

al coordinador de nivel superior. El coordinador detecta que C es el hijo inminente, por lo tanto, luego de que el coordinador raíz actualiza el reloj global, se envía un mensaje * al coordinador C, que continúa con la simulación.

El simulador cicla, y hace tn = 12, tl = 10, y e = 0. La celda (1,10) es elegida de la lista de próximos eventos. En este caso,

New-state = 1; y
Cells[1,9].state = 0, luego

New-states = { <(1,10), 1> }

La celda está en la Ylist, por ende se crea un mensaje Y y se envía al coordinador M. El coordinador reaccionará de la misma forma que fuera explicada en la sección anterior. Cuando el mensaje

< Y, C(1,10), 1, 12 >

llega al coordinador M, se traduce al mensaje

< X, A(2,10), 12, 1 >

que será transmitido al coordinador chato A. Este coordinador lo insertará en su cola de próximos eventos, para ser tratado en el futuro.

Como puede verse, el pasaje de mensajes se reduce considerablemente, disminuyendo el overhead desperdiciado en interacción entre mensajes.

6. HERRAMIENTA DE SIMULACIÓN

En esta sección se hará un estudio detallado de una herramienta que fue desarrollada como derivación de los formalismos analizados en los capítulos anteriores. La finalidad de esta herramienta es poder implementar modelos de ejecución de espacios de celdas que, además, puedan ser combinados con modelos DEVS. Para ello, el primer paso fue desarrollar un entorno de simulación para modelos DEVS [BBW98a], que posteriormente fue extendido para hacer simulaciones de espacios de celdas [BBW98b]. En las siguientes secciones se hará un análisis detallado de ambas soluciones. Para analizar detalles específicos de la implementación de ambas herramientas, puede consultarse [BBW98c].

6.1 GAD: un entorno genérico para simulaciones DEVS

Como fuera mencionado, el primer paso en el desarrollo de una herramienta de simulación de espacios de celdas fue la construcción de una herramienta genérica de modelado DEVS [BBW98a], llamada GAD (General Application tool for DEVS modelling and simulation). Esta fue diseñada redefiniendo la jerarquía de modelos de Zeigler [Zei90, GFE95, Zei97], que fue tomada como base para el desarrollo. En GAD se permite la definición de modelos atómicos en C++, que se incorporan a una jerarquía de clases. También se usa un lenguaje de especificación que permite definir el acoplamiento entre modelos, la declaración de valores iniciales para estos, y el conjunto de eventos externos que influenciarán al modelo externo. La jerarquía de clases que implementa la herramienta puede verse en la siguiente figura.

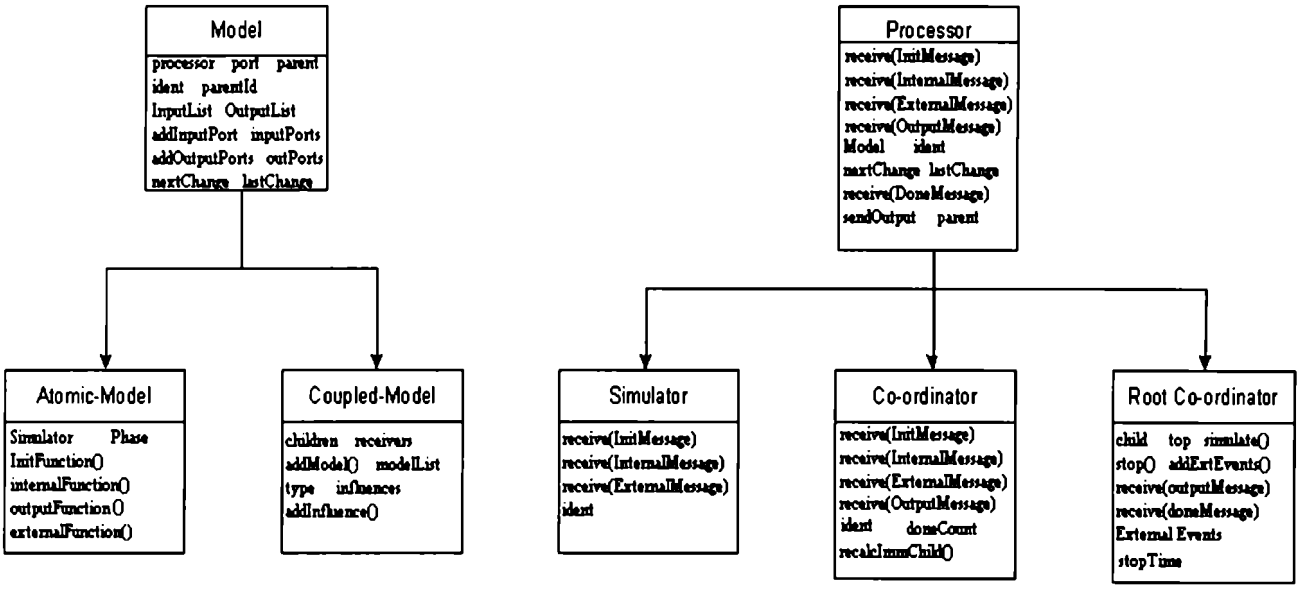


Figura 64. Jerarquía de clases de la herramienta GAD [BBW98c].

Todas las clases en la herramienta están derivadas de *Models* y *Processors*, que proveen los constructores básicos necesarios para la simulación de modelos DEVS. La clase *Models* es la clase base de *Atomic* y *Coupled* las cuales son especializadas en algunos casos más específicos. Es la clase base abstracta para la jerarquía de modelos, cuyas responsabilidades son administrar los ports de entrada y salida (*addInputPorts, inputPorts, outputPorts*), y conocer la hora de los eventos próximo y anterior (*nextChange, lastChange*). Los Modelos tienen variables de instancia *id* (que identifica al modelo de forma unívoca, y es asignada cuando el modelo se crea), *processor* (que identifican a los procesadores asociados), *parent* (que es un enlace al modelo acoplado que contiene este modelo), *inportList* y *outportList* (para especificar la interacción entre modelos). Finalmente, el método *sendOutput* envía la salida del modelo.

La clase *Atomic* es una especialización de la clase *Model* que se usa para representar los modelos atómicos básicos. Los métodos *internalFunction, externalFunction*, y *outputFunction* representan las transiciones interna, externa, y la función de salida respectivamente. Como especialización de *Models*, hereda las facilidades para manejo de ports, de consulta de

estado interno y planificación del próximo evento. Esta clase es *abstracta* ya que provee únicamente la interfaz para los modelos básicos, incluyendo las siguientes tareas: inicialización de los modelos (*initFunction*), ejecutar la función de transición interna (*internalFunction*), la de transición externa (*externalFunction*), y la de salida (*OutputFunction*). Estas funciones deben ser sobrecargadas por el modelador, de forma tal de poder definir el comportamiento dependiendo del modelo a implementar. *Atomic* también implementa algunos otros métodos utilizados para interactuar con la jerarquía de simulación: *holdIn*, que cambia el estado del modelo y programa el próximo evento interno; *passivate*, que cambia el estado del modelo a pasiva y la hora de próximo cambio en infinito; y *state*, que cambia o retorna el estado del modelo.

CoupledModel implementa las construcciones jerárquicas definidas por el formalismo DEVS. Esta clase es otra especialización de *Model* y representa a un modelo acoplado, que también hereda el comportamiento para manejo de ports. Por otro lado es responsable de la incorporación y manejo de componentes (*AddModel, ModelList*), y de registrar las dependencias entre los componentes hijos (*addInfluence*). Un modelo acoplado se define por sus componentes (*children*), y las relaciones de acoplamiento, que son especificadas por las variables de instancia *receivers* e *influences*, que permiten definir la función Z_{ij}.

Los *Processors* se construyen para ejecutar los procedimientos de simulación abstracta, implementando los conceptos definidos por Zeigler. La clase *Processor* es la clase abstracta base para toda la jerarquía de procesadores, y sus responsabilidades son recibir mensajes (*receive*), establecer la relación con su modelo asociado (*model*), y con su *Processor* padre (*parent*). La relación de acoplamiento entre procesos y modelos se completa por medio de las variables de instancia *Model* y *processor*. Los *Simulators* y *Coordinators* administran el acoplamiento con los modelos atómicos y acoplados respectivamente. Las horas de último y próximo evento (*lastChange, nextChange*) se almacenan para identificar el hijo inminente, así como para verificar la correctitud de los tiempos simulados.

El *RootCoordinator* representa la raíz del árbol de procesadores a partir de la cual se comienza con el mecanismo de simulación. Debe administrar la simulación en sus aspectos globales: mantiene la hora global (*clock*), y está a cargo de comenzar y terminar la simulación (*simulate, stop, stopTime*). El método *receive*, si llegó la hora de finalización especificada (o se consumieron todos los eventos externos y no hay ningún modelo activo), la simulación se da por finalizada; de no ser así, el analiza el próximo evento a ser tratado y se envía un mensaje al coordinador de mayor nivel avanzando de esta forma la hora de la simulación. Es una especialización de la clase *Processor*, que representa la raíz del árbol de procesadores a partir de la cual se comienza con el mecanismo de simulación. Este procesador no tiene un modelo asociado, sino que está relacionado con el coordinador de mayor nivel (*child, top*). Sus responsabilidades incluyen, además, administrar los eventos externos (*addExternalEvents*), encolándolos en una lista (*ExternalEvents*), de forma tal que puedan ser usados como eventos exteriores a todo el modelo. También avanza la hora de simulación, y genera la salida en función de los mensajes de salida recibidos (*sendOutput*). El método *initialize* carga las horas de eventos próximo y anterior en cero, limpia la lista de eventos externos, e inicializa la hora de terminación en infinito. Finalmente, el método *simulate* envía un mensaje de inicialización al modelo acoplado de mas alto nivel en la jerarquía, y ejecuta el método *run* de la clase *MessageAdmin*, comenzando de esta forma el ciclo de envío y recepción de mensajes.

La clase *Simulator* es una especialización de *Processor*, que administra el mecanismo de simulación para los modelos atómicos. Sus responsabilidades son recibir mensajes y canalizarlos hacia el modelo atómico asociado. Si se activa *receive(InitMessage)*, se invoca a la función de inicialización y se envía un mensaje **done** con la hora del próximo evento al padre. Si se activa *receive(ExternalMessage)*, se invoca a *Model.externalFunction*, y se envía un mensaje **done** con la hora del próximo evento al padre. Al recibir *receive(InternalMessage)*, se invoca a *Model.outputFunction*, y luego a *Model.internalFunction*. Por último envía un mensaje **done** al coordinador padre, indicando la hora del próximo evento de su modelo asociado. En todos los casos se actualizan las horas de próximo y último evento en base al mensaje recibido.

Coordinator es otra especialización de la clase *Processor*. Los coordinadores representan el mecanismo de simulación para los modelos acoplados. Tiene diversas responsabilidades. Una de ellas es redireccionar los mensajes de inicialización hacia todos sus hijos (*receive(InitMessage)*). También redirecciona los mensajes externos por medio de las influencias de los ports (*receive(ExternalMessage), imminentChild, Model.influences*). Para ello envía un mensaje externo (de tipo **X**) a todos los procesadores influenciados por el port desde donde ingresó el mensaje. Por otro lado, redirecciona los mensajes de salida por medio de las influencias de los ports y el esquema de acoplamiento externo reenviando los mensajes hacia los hijos o hacia afuera a modelos externos (*recalcImmChild*), calculando y registrando el hijo inminente y su hora de activación entre todos los modelos que componen al acoplado. También debe redireccionar los mensajes internos hacia el *Processor* inminente, y enviar a su procesador padre la hora del próximo evento interno (*receive(doneMessage)*), que permite indicar al padre la hora del próximo evento interno. Finalmente debe reaccionar al recibir mensajes * y **X**, de la forma descripta en la sección 2.2.3. En todos los casos se usa la variable de instancia

doneCount para registrar la cantidad de mensajes enviados (cuando recibe mensajes **done** de respuesta de cada uno de los hijos).

Como fuera mencionado, la interacción entre los procesos es llevada a cabo a través de pasaje de mensajes. Los mensajes incluyen información relacionada con el origen (o el destino), la hora del evento relacionado, y su contenido, consistente de un número de port y un valor asociado. Se implementan los cuatro tipos de mensajes mencionados en las secciones anteriores: **X**, **Y**, * y **done.** Para lograr estos objetivos, *Message* implementa una clase básica que define los posibles mensajes intercambiados por los modelos. Cada mensaje acarrea datos acerca del modelo que generó el valor y su tiempo de evento.

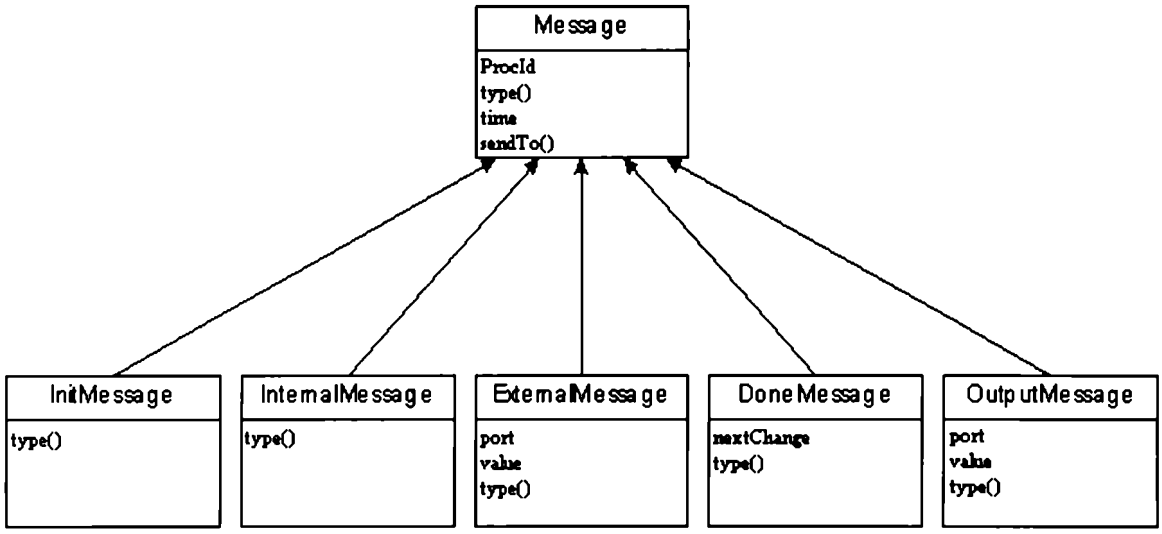


Figura 65. Jerarquía de clases para Messages.

La estructura para cada uno de los mensajes difiere: los mensajes **done** deben devolver la hora del siguiente cambio para el modelo asociado; los mensajes externos y de salida tienen un valor asociado y un destino. Todos devuelven un tipo y tienen un destino, y un *Processor* asociado (de destino u origen).

Estos mensajes interactúan entre los modelos de la forma que fuera analizada en la sección 4.2. Para ello se implementó la clase *MessageAdm*, dedicada a recibir la invocación entre módulos, y administrar su comunicación. Esta clase provee un método único para enviar y recibir mensajes, proveyendo una solución centralizada que permite que la política de pasaje de mensajes quede encapsulada en el administrador. En la versión actual, la política implementada es FIFO, pero se podría usar cualquiera de los métodos de simulación paralela mencionados en la sección 2.4. Cambiar a otra política de pasaje de mensajes implicaría únicamente modificar el método de distribución de mensajes del administrador.

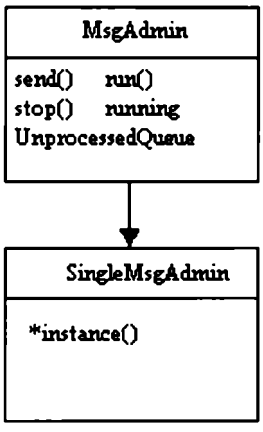


Figura 66. Jerarquía de clases para MsgAdmin.

El método *send* está encargado de transmitir un mensaje entre un módulo y el siguiente. La *UnprocessedQueue* se usa para el caso que sea necesario encolar mensajes para ser transmitidos a un *Processor* determinado. En esta implementación la política de encolado seleccionada es FIFO, forzando así una simulación secuencial, en la que el despacho de un mensaje se produce cuando el modelo finaliza el procesamiento del anterior. Es por esto que la simulación comienza cuando *Root* invoca el método *run* para que comience el ciclo de pasaje de mensajes. El método *stop* finaliza el ciclo de pasaje de mensajes. Finalmente, la clase *SingleMsgAdmin* está encargada de enviar los mensajes entre procesadores.

La clase *ModelAdm* administra los modelos creados. Sus principales funciones incluyen:

- Creación de nuevos modelos: crea una instancia de un modelo y le asigna un identificador único. Esta es la única clase que puede usarse para crear modelos nuevos;
- Asociación de identificadores de modelos con los modelos existentes: todos están contenidos en una lista que es mantenida por el administrador del modelo. El administrador debe referenciar a los modelos usando una clave única.

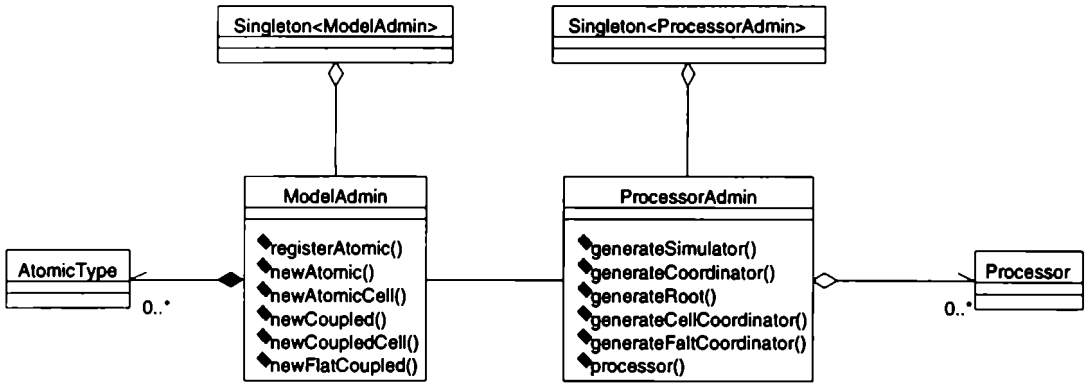


Figura 67. Jerarquía de clases para ModelAdm.

Tanto los procesadores como los modelos utilizados en la simulación son creados cuando se lee la especificación de modelos y eliminados al final de la ejecución. Durante el transcurso de la simulación se hace referencia constantemente a modelos padres e hijos por esto debe proveerse un mecanismo para conocer a estos modelos o procesadores. Delegar esta responsabilidad en un administrador permite distinguir el uso de los modelos de su construcción, destrucción y ubicación. De esta forma fácilmente puede aplicarse una política distribuida sin impactar en la aplicación. La clase *ProcessorAdmin* administra todos los procesadores de la simulación. Debe crear instancias de cada uno de los distintos coordinadores, desasignándolos al finalizar la simulación (*generateRoot, generateProcessor*).

El simulador es extensible a partir de nuevos modelos atómicos que implementan modelos básicos. La instanciación de modelos es responsabilidad del administrador de modelos y es por esto que existe un método para registrar todos los modelos derivados de la clase *Atomic* que se desea incluir dentro de la simulación. Para ello debe registrarse usando el método *registerAtomic*. Los demás métodos son invocados automáticamente al crear nuevos modelos atómicos de celdas o modelos acoplados (de cualquier tipo). Al activarse los métodos relacionados con los distintos lenguajes de especificación (de modelos de celdas o de modelos asociados), se registran los nuevos modelos como modelos de base que pueden ser reusados.

Se definió un conjunto de métodos (*SimLoader*), que proveen una interfaz para cargar la configuración del simulador. Esta clase es abstracta ya que sus derivadas implementan el método concreto de obtención de parámetros, incluyendo dos métodos diferentes para iniciar una simulación. *La* primera usa la clase *StandAloneLoader*, que es la responsable de cargar los parámetros usando un intérprete de comandos. La clase *NetworkLoader* es la responsable de obtener los mismos parámetros usando servicios TCP/IP (en esta implementación). De esta forma, el simulador puede ser ejecutado como un servidor de simulación, y los parámetros cargarse de forma remota, obteniendo los resultados de la misma forma. En este caso, el simulador esperará la especificación de la simulación en un port (*CommChannel*), la ejecutará, y devolverá los resultados de la misma forma. La instanciación del cargador del simulador se realiza al comienzo del programa ya que sin el no es posible cargar los modelos, y por lo tanto llevar a cabo la simulación. Las variables de instancia *modelStream, eventStream, logStream* y *outputStream* implementan la definición de los archivos usados para entrada/salida del modelo. La variable *stopTime* define la hora de fin de la simulación. Una vez tomada la decisión sobre

el cargador a instanciar debe indicársele al simulador la instancia elegida. Las funciones *loadData* y *writeResults* son las encargadas de cargar y grabar datos y resultados respectivamente.

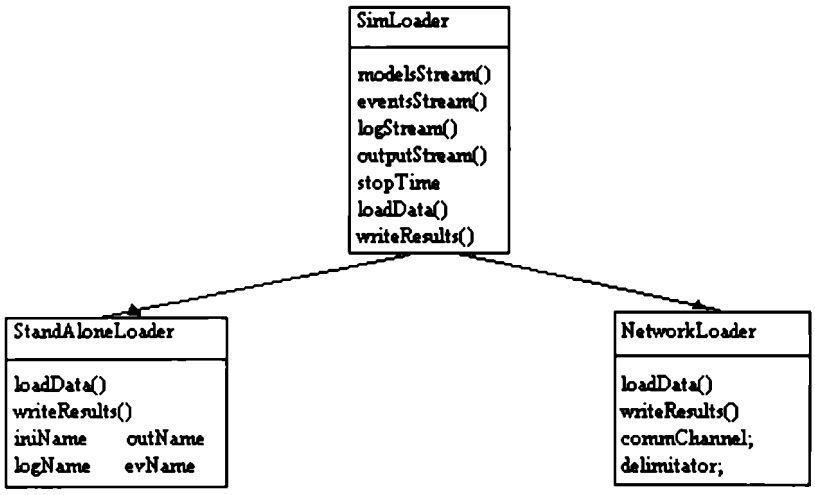


Figura 68. Jerarquía de clases para SimLoader.

Finalmente, la clase *MainSimulator* es la responsable de crear el árbol de modelos, y de establecer los enlaces entre los ports usando la especificación. Para hacerlo, toma los archivos de entrada del cargador, y se usa el objeto *IniFile* (instancia de *Ini*) con el fin de recorrer y analizar la especificación del modelo (*parse*). El coordinador raíz debe cargar el modelo, y una vez construida la jerarquía de modelos, la simulación puede comenzar (cargando los eventos externos, y creando una lista de eventos, así como el tiempo de parada). El entorno de simulación se completa con los archivos de salida para el logging de mensajes y la salida del simulador. Una vez armado el entorno de simulación la instancia de *MainSimulator* le informa a la instancia de *Root* los eventos externos especificados y la hora a la cual debe detener la simulación (por defecto es infinito).

MainSimulator es la responsable de crear el árbol de modelos/procesadores y los vínculos entre sus ports a partir de una especificación. Para realizar esta tarea toma los archivos de entrada de *SimLoader*, y utiliza la clase *Ini* para analizar la especificación de los modelos. Una vez armado el entorno de simulación la instancia de *Simulator* le informa a la instancia de *Root* los eventos externos especificados y la hora a la cual debe detener la simulación (por defecto es infinito). En este momento se invoca el método run para que la simulación comience. El método *loadModels* invoca al método *parse* de la clase *Ini*. Una vez que la especificación se encuentra procesada y disponible por la clase *Ini*, se procede a invocar al método *loadModel* para que cree en forma jerárquica todos los modelos/procesadores que componen la simulación empezando por *Root*, que es el procesador de mayor nivel. *LoadModel* invoca al método *loadPorts* para que cree los puertos que posee el modelo que se está creando. Luego, carga los componentes. Por último procede a crear los enlaces internos al modelo y de conexión con el exterior a través del método *loadLinks*. Este método Crea los vínculos entre dos puertos, en relación al modelo que se está configurando. Existen tres tipos de vínculos: internos (se trata una conexión entre dos puertos que pertenecen a modelos hijos del modelo acoplado que se está configurando), de entrada (el primer puerto pertenece al acoplado y el segundo a un modelo hijo), y de salida (el primer puerto se encuentra en un hijo y el otro pertenece al acoplado).

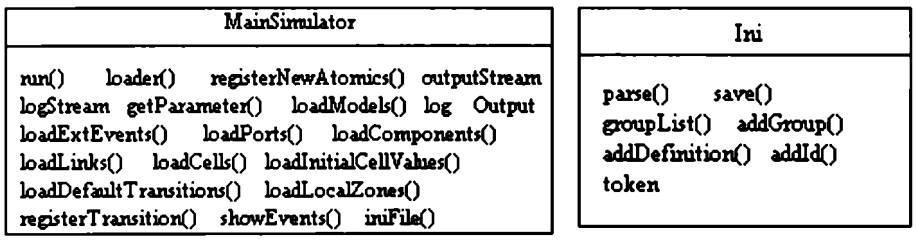


Figura 69. Descripción de MainSimulator.

Para comenzar la simulación, deben especificarse los modelos (y su acoplamiento), los eventos de entrada externa, la hora de terminación de la simulación, y los archivos de entrada y salida que serán utilizados. La especificación del modelo se hace por medio del uso de un lenguaje de propósito específico, en el cual una descripción para un modelo

debe incluir los ports de entrada/salida y el acoplamiento con otros modelos. Mayores detalles acerca del uso de esta herramienta pueden encontrarse en [BBW98c]. En la sección 7.2 se darán algunos ejemplos de su uso.

6.2 Interacción de mensajes en GAD

En esta sección se describen las distintas etapas por las que atraviesa el simulador de GAD para cumplir con los requerimientos del usuario. La simulación comienza cuando el usuario especifica los argumentos de la simulación usando el intérprete de comando. El método principal *main* toma los parámetros, y los analiza buscando si se desea invocar al simulador en modo servidor. Una vez determinado el modo, se instancia el tipo de *SimLoader* que corresponda, se le indica a *MainSimulator* qué cargador debe utilizar, y se activa el entorno ejecutando el método *run*.

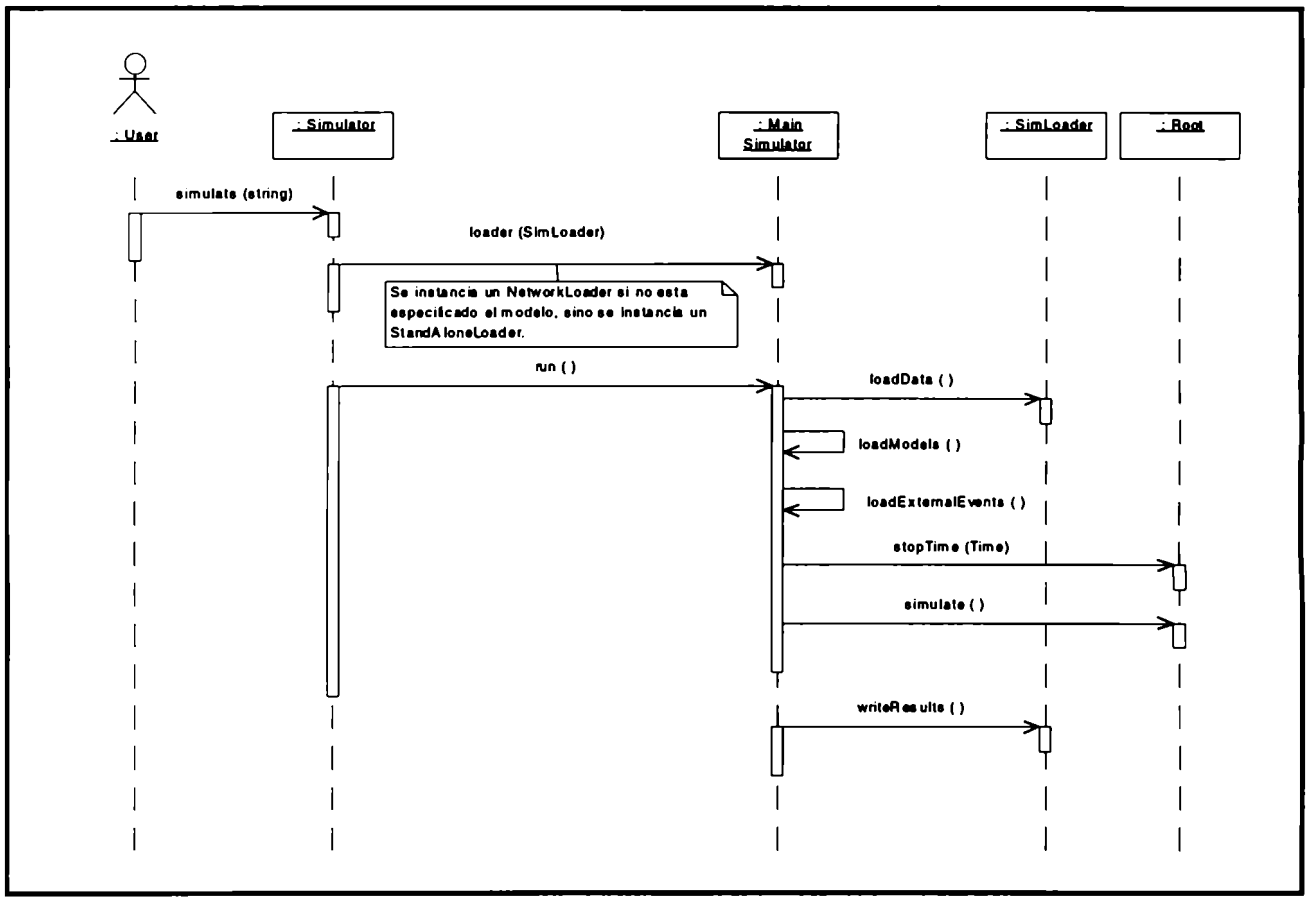


Figura 70. Métodos activados al iniciar la simulación [BBW98c].

MainSimulator utiliza el cargador para obtener la configuración de la especificación a simular, y carga los modelos y los eventos externos. A esta altura la jerarquía de *Processors-Models* está creada. Luego indica a *Root* la hora de finalización especificada e inicia la simulación invocando el método *simulate()*. Al finalizar se le indica al cargador que retorne los resultados obtenidos.

La simulación comienza con el envío de un mensaje de inicialización al acoplado de mayor nivel. Este reenvía el mensaje a todos sus hijos, y espera un mensaje **done** de cada uno de ellos con la hora de su próximo evento. Una vez recibidos todos los mensajes en respuesta a los enviados, calcula cual de todos sus hijos es el inminente, es decir el que tenga la hora de próximo evento menor, y envía un mensaje **done** a su padre indicando su hora de próximo evento interno. Cuando *Root* recibe un mensaje **done** toma la hora del mensaje como la hora del próximo cambio de estado de su hijo inminente (top, otro no tiene), y la compara con el próximo evento externo que tiene programado. Se queda con el menor de los dos y envía en un mensaje en función de ello. Si es un evento externo enviará un *ExternalMessage* y si es un interno enviará un *InternalMessage*, actualizando la hora global del simulador.

Si la planificación del hijo inminente propone como hora de próximo cambio infinito, esto significa que todos los

modelos que componen la simulación se han pasivado. En este punto la falta de eventos externos marca el fin de la simulación. Otra posibilidad de finalización puede darse si la hora del próximo evento elegido es mayor o igual que la hora de finalización especificada.

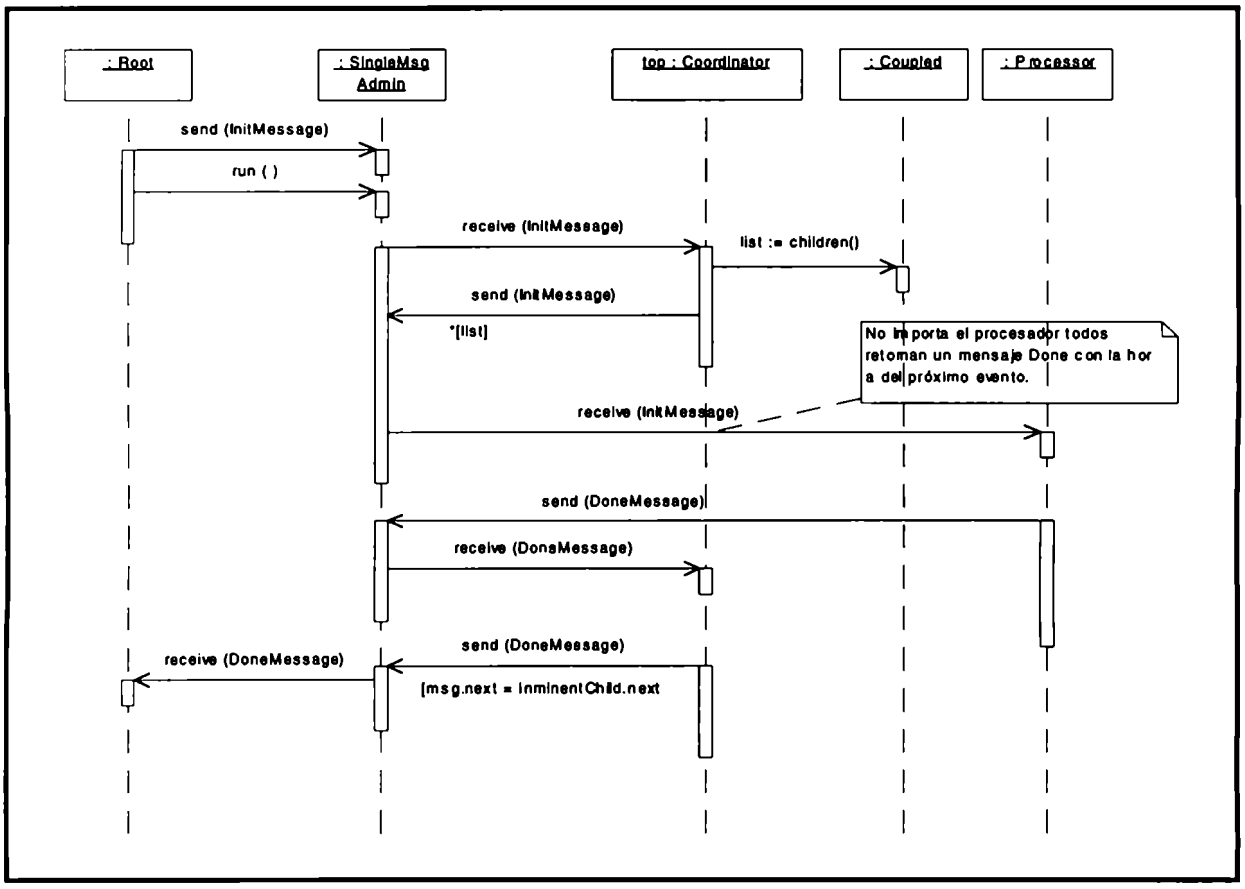


Figura 71. Mensajes y métodos usados durante la inicialización [BBW98c].

Para tratar los eventos internos, *Root* envía un *InternalMessage* al coordinador de mayor nivel indicando el arribo de un evento interno, es decir un cambio de estado programado para alguno de sus descendientes. Este mensaje llega al coordinador *top* y este lo redirecciona hacia el hijo inminente, este hijo es potencialmente cualquier especialización de *Processor*: *Simulator, Coordinator, CellCoordinator, FlatCoordinator.*

Una instancia de la clase *Simulator* al recibir un mensaje interno invoca el método *outputFunction()* y luego *internalFunction()* de su modelo atómico asociado. Al retornar de este último método arma un *DoneMessage* en respuesta indicando la hora de su próximo cambio de estado. Si el modelo atómico genera una salida esta será enviada por su simulador asociado a su coordinador padre vía el administrador de mensajes.

Una instancia de la clase *Coordinator* responde a un evento redireccionando el mensaje hacia el procesador hijo inminente. Cuando reciba el *DoneMessage* de su hijo toma la hora de su hijo inminente y la envía en un mensaje done hacia su procesador padre.

(Nota: de aquí en más se ha obviado la ilustración del administrador de mensajes, sólo mostrando la invocación y el resultado obtenido de su intervención ya que su funcionalidad ya ha sido descripta en la figura anterior).

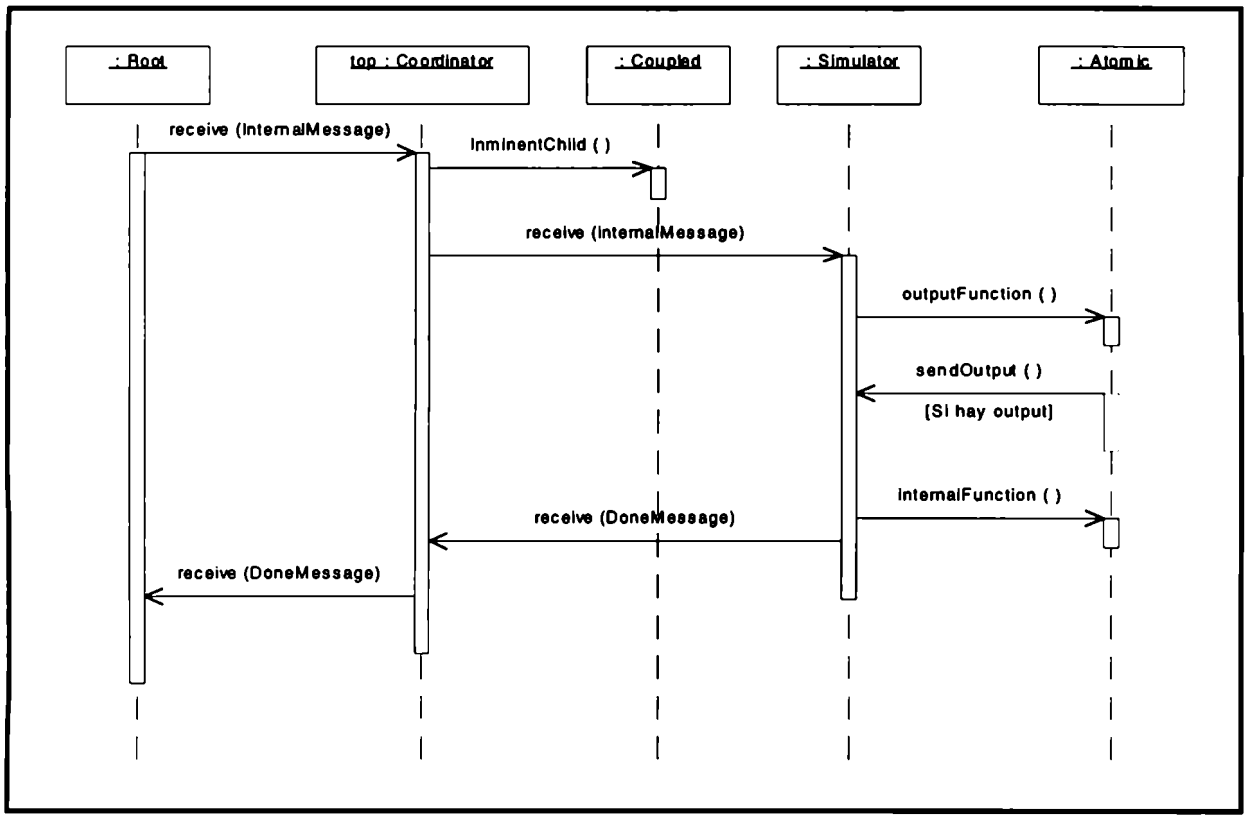


Figura 72. Mensajes y métodos correspondientes a eventos internos [BBW98c].

Para responder a la llegada de eventos externos, *Root* envía un mensaje externo al coordinador exterior, indicando el puerto por el cual ha llegad. Una instancia de *Coordinador* que recibe un mensaje externo debe analizar el mapeo de las influencias sobre el puerto destino del mensaje, usando la lista de influencia que posee el *Coupled* asociado. Por cada influencia se genera y envía un nuevo mensaje externo con el valor del original. Por último el coordinador registra la cantidad de mensajes done que debe esperar para luego enviar un mensaje done a su coordinador padre.

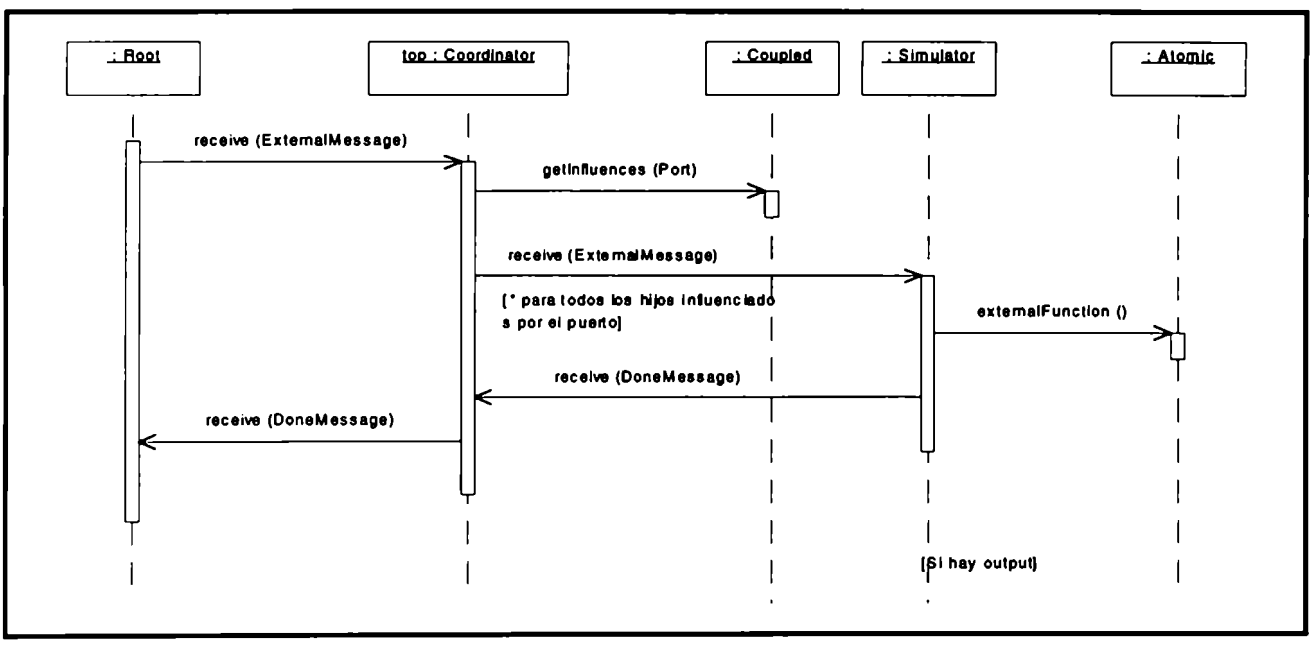


Figura 73. Mensajes y métodos correspondientes a eventos externos [BBW98c].

Finalmente, cuando una instancia de *Coordinator* recibe un mensaje de salida, debe analizar las influencias del puerto por el cual se produce la misma. Si la influencia tiene destino en un modelo externo al coordinador debe reenviarse el mensaje de salida hacia el coordinador padre. El otro destino posible para la influencia es un modelo dentro del coordinador. En este caso el mensaje de salida es traducido a un *ExternalMessage* y enviado por el puerto que indique la dependencia esperando un *DoneMessage* de cada uno. Cuando arriben todos y cada uno de ellos retornará un mensaje done con la hora del hijo inminente al coordinador padre.

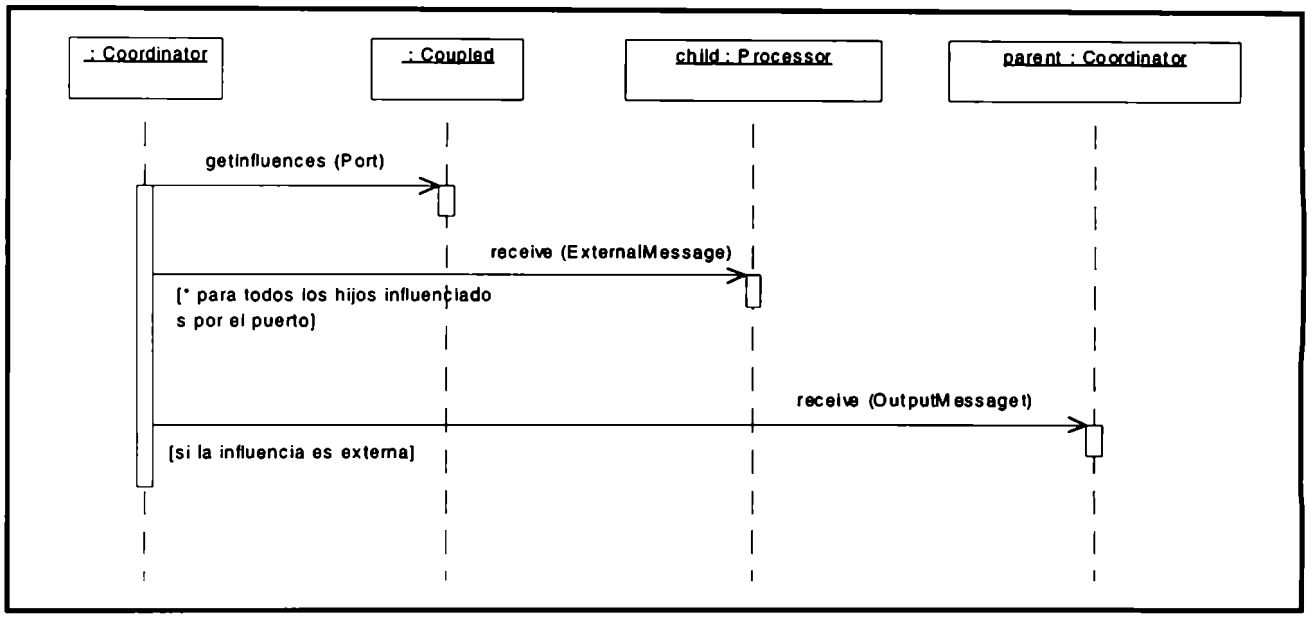


Figura 74. Mensajes y métodos correspondientes a eventos externos[BBW98c].

6.3 La herramienta CD++

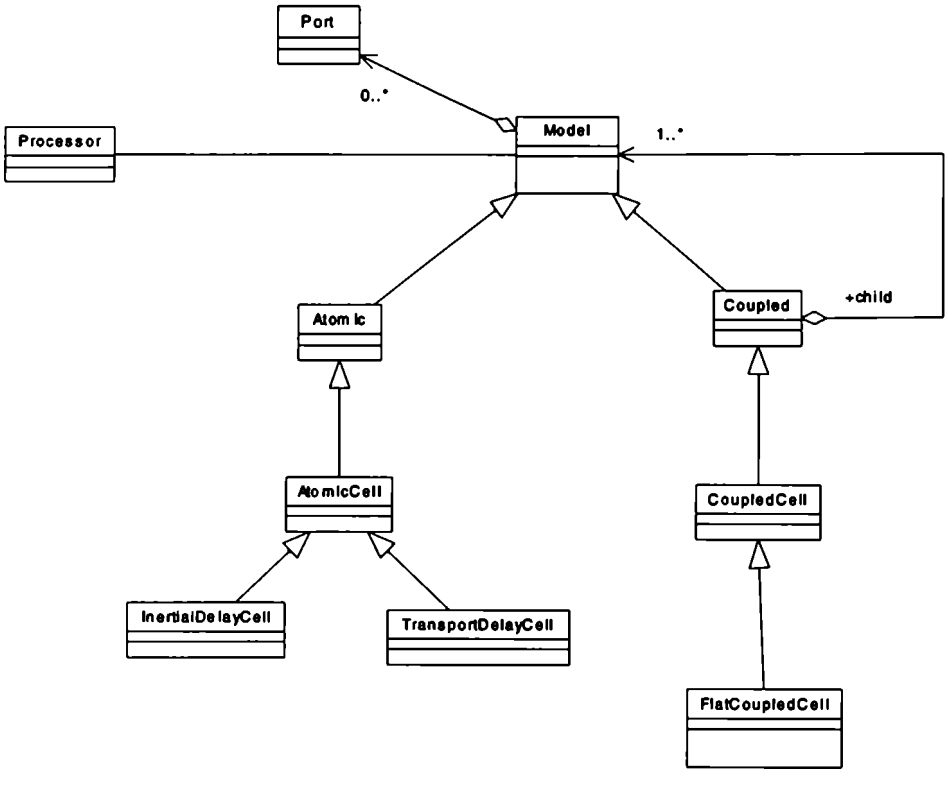
La herramienta CD++ es una extensión de GAD, que está basada en el formalismo Cell-DEVS, con el objetivo de proveer un ambiente para construir modelos celulares. La arquitectura del sistema de simulación deriva de los conceptos abstractos de simulación asociados con los formalismos DEVS y Cell-DEVS.

En esta sección se analiza el comportamiento de las clases básicas (definidas en [BBW98b] como implementación del trabajo analizado en [WFG97b]), y su extensión para simular modelos Cell-DEVS, que es una extensión a los trabajos mencionados en la sección anterior.

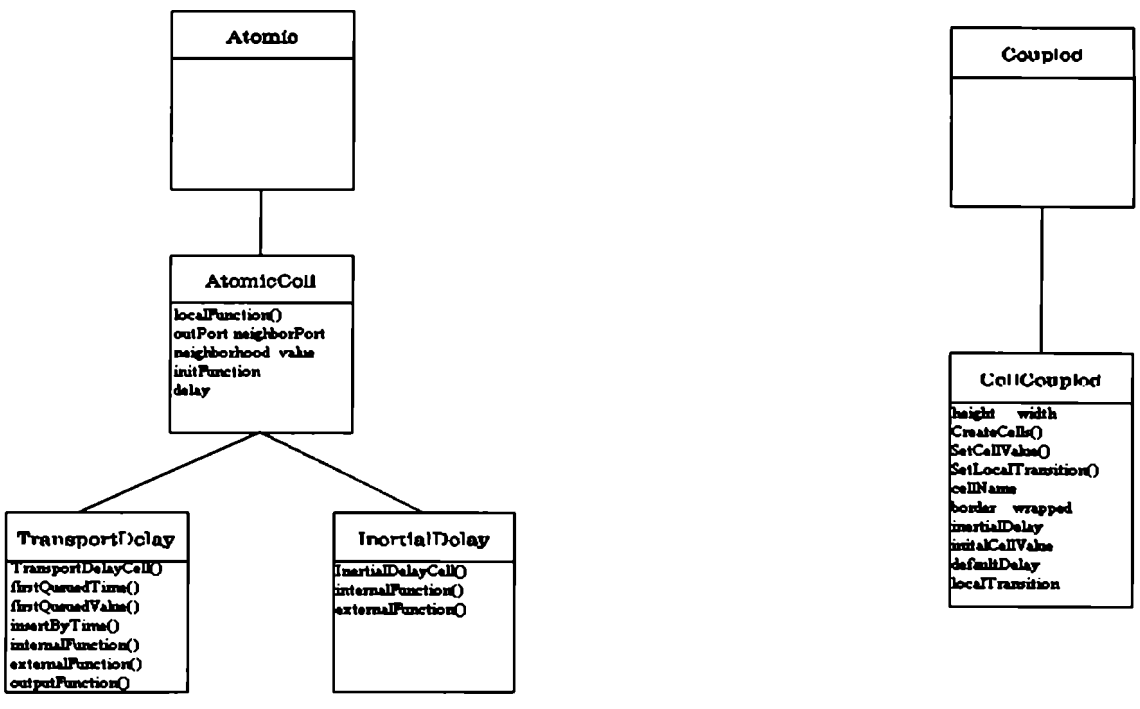
Al analizar el comportamiento de la herramienta con respecto a la simulación de espacios de celdas, se ejecutarán simuladores con el comportamiento definido en la sección 4.2. Aquí se pudo ver que cada celda posee un conjunto de estados, un comportamiento y una demora de actualización. Cuando el tiempo de demora ha sido consumido se ejecutan los procedimientos asociados. Estas acciones se repetirán para cada evento almacenado en la cola de espera, que si está vacía provoca que la celda pase a un estado pasivo. Los coordinadores envían los cambios a cada celda en la vecindad a medida que estas cambian.

CD++ provee un motor de simulación junto con las estructuras que permiten configurar la especificación y asociación de los modelos que componen el sistema Cell-DEVS a simular. La herramienta implementa una interfaz genérica para la comunicación entre modelos.

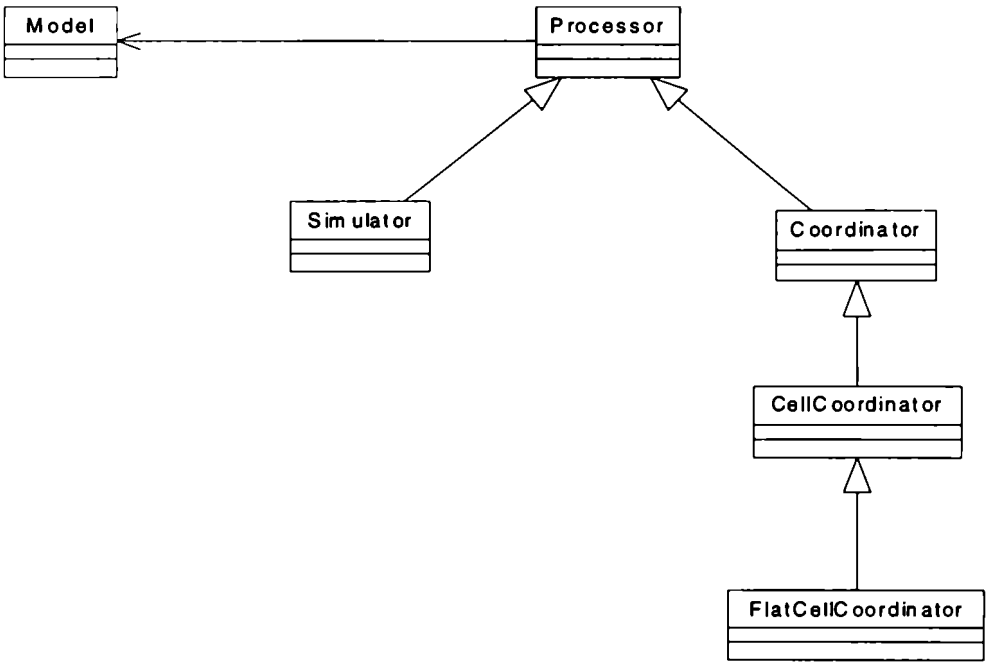
El desarrollo de un simulador usando la herramienta está dividido en tres áreas. La primera incluye la especificación de los modelos y eventos externos. La segunda está relacionada con la carga de la especificación del sistema que se desea simular. La tercera consiste en ejecutar la simulación del sistema y la obtención de resultados. Para cumplir con estos objetivos, la jerarquía de clases de la sección anterior fue extendida de la siguiente forma:


Port
0..*
Model
1..*
Processor
Atomic
Coupled
+child
AtomicCell
CoupledCell
InertialDelayCell
TransportDelayCell
FlatCoupledCell

(a)


Atomic
AtomicCell
localFunction()
outPort neighborPort
neighborhood value
initFunction
delay
TransportDelay
TransportDelayCell()
firstQueuedTime()
firstQueuedValue()
insertByTime()
internalFunction()
externalFunction()
outputFunction()
InertialDelay
InertialDelayCell()
internalFunction()
externalFunction()
Coupled
CellCoupled
height width
CreateCells()
SetCellValue()
SetLocalTransition()
cellName
border wrapped
inertialDelay
initialCellValue
defaultDelay
localTransition

(b)



(c)

Figura 75 - (a) Jerarquía de clases de Modelos en la herramienta CD++. (b) Detalle de las clases para manejo de espacios de celdas. (c) Jerarquía de Simuladores en la misma herramienta. [BBW98c]

En este caso, la clase *AtomicCell* es una especialización de la clase *Atomic* y representa la clase base para dos tipos de celdas: las que usan demoras de transporte, y las que usan demoras inerciales. Sus responsabilidades son ejecutar la función de transición local asociada (*localFunction*), así como conocer el vecindario y obtener los valores de entrada para la función de transición (*neighborhood*). También define la interconexión con otros modelos por medio de los ports (*outport, NeighborPort*), almacena el estado para la celda (*value*) y su demora (*delay*).

La clase *TransportDelayCell* es una especialización de *AtomicCell*. Esta debe implementar la especificación de las demoras de transporte usando los métodos analizados en la sección 3.2 (*internalFunction, externalFunction, outputFunction)*. Por otro lado, la clase *InertialDelayCell* debe implementar la especificación de demoras inerciales.

CoupledCell es una especialización de la clase *Coupled*. Esta debe conocer el tipo de demora por defecto para las celdas (*inertialDelay*), determinar el tiempo de demora por defecto para las celdas (*defaultDelay*), el tipo de borde (*borderWrapped*), el valor inicial por defecto para cada celda, la función de transición local por defecto para las celdas, las regiones definidas con otro comportamiento. Finalmente debe crear el conjunto de celdas y los vínculos entre ellas, y cargar el valor inicial especificado para cada celda.

Para realizar la simulación de autómatas celulares, CD++ provee un lenguaje de especificación para definir expresiones lógicas que representan el comportamiento de cada celda, evitando de esta forma la programación en un lenguaje de alto nivel. La definición de los modelos se carga de un archivo conteniendo especificaciones escritas en este lenguaje propietario. La descripción de cada modelo incluye sus ports de entrada/salida, y sus relaciones con el resto de los modelos. La carga del sistema a simular incluye la lectura e interpretación de los archivos de especificación y la posterior generación del entorno que conformará el sistema a simular. Estos procedimientos serán analizados en la próxima sección.

Finalmente la herramienta fue especializada en modelos achatados. Un acoplado celular achatado no debe crear la grilla de celdas ya guarda directamente el valor de la celda evitando al sobrecarga del pasaje de mensajes entre los modelos. Es por esto que debe proveer los métodos para aparente poseer un conjunto de celdas. Esta grilla de celdas virtuales permite que su uso sea completamente transparente. Para ello, se implementó la clase *FlatCoupledCell* como una

especialización de la clase *CoupledCell*. Representa a un *CoupledCell* pero sin ser un contenedor de modelos ya que las celdas que posee son virtuales. Sus responsabilidades son administrar las celdas virtuales (*createCells*), registrar la función de cálculo local para cada celda virtual (*setLocalTransition*), y administrar las dependencias entre las celdas virtuales con respecto a sus vecinos (*addInfluence*). Finalmente debe ejecutar los métodos analizados en la sección 4.3 (*initFunction, externalFunction, internalFunction*).

La clase *CellCoordinator* es una especialización de la clase *Coordinator*. Representa el mecanismo de simulación para los modelos celulares, y debe administrar los hijos inminentes con una política especifica para los modelos celulares, así como administrar las los mensajes de salida con una política especifica para los modelos celulares. En todos casos, usa el método *receive*, que implementa los algoritmos analizados en la sección 4.2.

Finalmente, la clase *FlatCoordinator* es una especialización de la clase *Coordinator*, que representa el mecanismo de simulación para los modelos celulares achatados. Sus responsabilidades son recibir los mensajes y direccionarlos hacia el modelo acoplado chato asociado. En este caso, el método *receive* implementa los mecanismos descriptos en la sección 4.3.

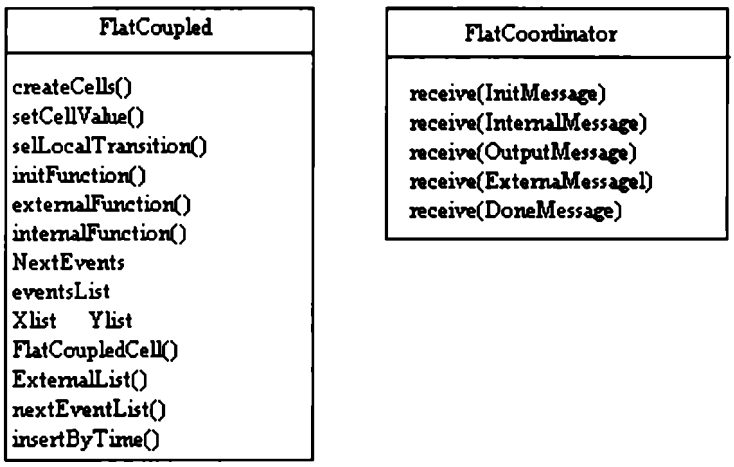


Figura 76 - Modelos achatados.

6.4 Interacción de mensajes en CD++

En este caso, la clase *CellCoupled* maneja a todas las celdas como modelos hijos. A diferencia de los modelos acoplados comunes todos los hijos posibles tienen la misma estructura. En la creación de las celdas se le asigna el comportamiento especificado a cada una y se la vincula con los modelos que especifique su vecindario. El comportamiento de las celdas está compuesto por dos factores, el tipo de demora y la función de transición local que a diferencia de los modelos atómicos tradicionales se especifica a través del lenguaje propietario.

La clase *CellCoordinator* da un comportamiento especial a la recepción de los eventos internos y de salida. Frente a un mensaje interno se enviarán tantos *InternalMessage* asincrónicamente como hijos inminentes existan. De esta forma se fuerza el orden en que los mensajes externos disparados por la salidas de las celdas serán tratados evitando así que una celda ejecute más de una vez en el mismo instante de tiempo.

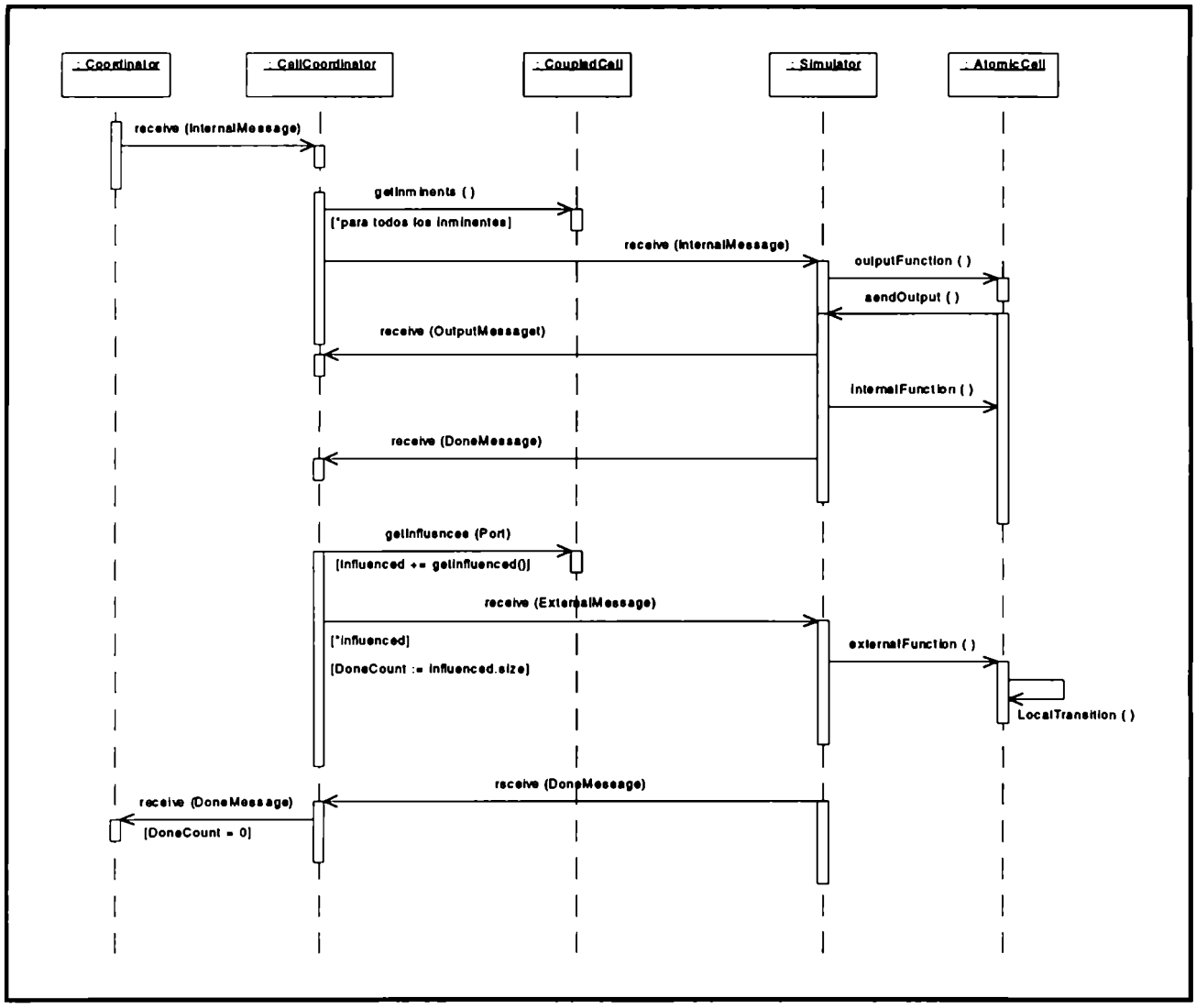


Figura 77 - Interacción de mensajes celulares [BBW98c].

Los modelos celulares convencionales (implementados a través de DEVS) están compuestos de tantos modelos básicos como celdas posea el autómata. Esto produce una gran inversión de tiempo en la creación y una sobrecarga de mensajes por cada evento que arriba al modelo celular. Dado que todas las celdas comparten la misma estructura, contienen los mismos puertos de entrada/salida, tienen el mismo comportamiento en función de su estado, y es posible obtener sus vecinos aplicando un desplazamiento a su ubicación (teniendo en cuenta el borde, que puede ser plano o circular), toda la estructura necesaria para llevar a cabo la simulación de este tipo de modelo puede simplificarse a una matriz de estados celulares junto a una especificación del lenguaje a utilizar. Esta simplificación permite disminuir drásticamente el tiempo de creación y elimina por completo el pasaje de mensajes dentro del modelo celular.

Cuando una instancia de *FlatCoordinator* recibe un mensaje externo invoca al método *externalFunction* del acoplado celular achatado asociado para todas las celdas virtuales (le indica la posición dentro del espacio de celdas) influencias por el puerto donde fue recibido el mensaje. El acoplado asociado calculará la función de transición local para la posición indicada y aplicará el algoritmo de demora correspondiente modificando la lista de próximos eventos en caso de ser necesario. Al finalizar se envía un *DoneMesage* al coordinador padre con la hora del hijo inminente (también virtual).

En el caso de recibir un *internalMessage* el *FlatCoordinator* invoca al método *internalFunction* del acoplado celular achatado asociado. Este último itera por todas las celdas con hora de próximo evento igual a la hora actual, y genera su output agregando todas las celdas influenciadas por la salida a la lista de próximos eventos. Por último se ejecuta el método *externalFunction* para todas las celdas virtuales que figuren en esta lista [BBW98c].

6.5 Incorporación de nuevos modelos

Para generar un nuevo modelo atómico, se debe comenzar por diseñar una nueva clase que sea derivada de la clase *Atomic* y se debe agregar al método *MainSimulator.registerNewAtomics()* el nuevo tipo de modelo atómico. Luego se deben sobrecargar obligatoriamente los métodos:

- *initFunction*: este método es invocado por el simulador una única vez al comenzar la simulación en el tiempo cero. El objetivo es permitir la inicialización que sea necesaria para que el modelo considere pertinente. Antes de invocar al método sigma vale infinito y el estado es pasivo.
- *externalFunction*: invocado cuando arriba un evento externo en alguno de los ports del modelo.
- *internalFunction*: antes de invocar al método sigma vale cero, ya que se ha cumplido el intervalo para la transición interna.
- *outputFunction*: antes de invocar al método sigma vale cero, ya que se ha cumplido el intervalo para la transición interna.
- *className*: nombre de la clase.

Estos métodos pueden invocar ciertas primitivas predefinidas, que permiten interactuar con el simulador abstracto:

- **holdIn(estado, tiempo):** indica al simulador que el modelo debe mantenerse en el *estado* durante un *tiempo*, que, luego de transcurrido, provocará una transición interna;
- **passivate():** indica al simulador que el modelo entra en modo pasivo y que únicamente deberá ser reactivado por la llegada de un evento externo;
- **sendOutput(hora, port, valor):** envía un mensaje de salida;
- **nextChange():** método de consulta para obtener el tiempo restante para su próximo cambio de estado (sigma);
- **lastChange():** método de consulta para obtener la hora de su último cambio de estado;
- **state():** método de consulta de la fase en la que se encuentra el modelo; y
- **getParameter(nombreModelo, nombreParámetro):** método de acceso a los parámetros de configuración de la clase.

Para definir un modelo acoplado, se debe inicializar un archivo de configuración cuyo contenido está compuesto por grupos de definiciones de modelos acoplados (obligatoria para el *top* de los demás acoplados que componen la simulación) y de configuración de modelos atómicos (opcional). Cada definición indica el nombre de un modelo (entre "[ ]") y sus atributos.

Con respecto a cada modelo (atómico o acoplado), existen cuatro propiedades a configurar: componentes (**components**), ports de salida (**out**), ports de entrada (**in**) y conexiones entre modelos (**link**). Los nombres con los que se definen cada una de las partes del modelo son reservados. La sintaxis es la siguiente:

- **Components**: este elemento se utiliza para describir los modelos que integrarán el modelo acoplado. El formato es **nombre_de_modelo@nombre_de_clase**. El nombre de modelo es necesario, ya que es posible construir un modelo acoplado con más de una instancia del mismo modelo atómico. El nombre de clase puede hacer referencia a cualquier modelo de base, ya sean atómicos o acoplados. Estos últimos deben estar descriptos en el mismo archivo de configuración como un nuevo grupo.
- **Out**: enumera los nombres de los ports de salida. Este atributo es opcional ya que un modelo puede no tenerlos.
- **In**: enumera los nombres de los ports de entrada. Este atributo es opcional ya que un modelo puede no tenerlos.
- **Link**: describe el esquema de acoplamiento interno, externo de entrada y externo de salida. El formato de este elemento sintáctico está dado por el par: **port_origen[@modelo] port_destino[@modelo]**. El modelo es opcional, y si no se indica, se toma como un port correspondiente al modelo acoplado en cuestión.

Por otro lado, se deben configurar los modelos atómicos participantes de la simulación. En caso de no figurar ninguna referencia, se tomarán los valores por defecto que haya programado el desarrollador de dicha clase. En otro caso, la configuración se especifica de la siguiente forma:

```
[nombre_modelo_atomico]
nombre_variable1 : valor_var1
...
nombre_variablen : valor_varn
```

Figura 78. Configuración de un modelo atómico

Aquí, los nombres de las variables dependerán de los definidos por el desarrollador de la clase que se desea configurar, por lo cual deberán estar documentados junto con los fuentes de la clase. Cada instancia de un tipo de modelos atómico podrá ser configurada independientemente de otras instancias del mismo tipo.

Los modelos Cell-DEVS son una variante de los modelos acoplados DEVS, y por esto se utiliza la misma especificación agregándole ciertos parámetros inherentes a las características del modelo, los que se analizan a continuación.

- *type* : [cell | flat]. Indica si el modelo es de celdas, o de celdas achatado (si no se especifica, es un modelo acoplado DEVS).
- *width* : INTEGER. Cantidad de columnas.
- *height* : INTEGER. Cantidad de filas.
- *link*: se especifica como para otros modelos acoplados, pero el port origen/destino debe usar el nombre del modelo de celdas correspondiente, junto con la posición (x,y) de la celda, o sea, Modelo(x,y).
- *border*: [WRAPPED | NOWRAPPED] (por default NOWRAPPED). Indica si el modelo es o no toroidal.
- *delay*: [TRASPORT | INERTIAL] (por default TRANSPORT). Especifica el tipo de demora.
- *neighbors*: nombreCelular(x_1, y_1), ..., nombreCelular(x_n, y_n), donde x_i,$y_i \in$ [-1,1]. Esto determina quiénes pertenecen al vecindario de cada celda (sólo celdas adyacentes).
- *initialvalue*: [0 | 1 | ?]. Representa el valor inicial para todo el espacio de celdas. Los valores posibles son: 0, 1 ó indefinido.
- *initialRowValue*: $fila_i$ $valor_1$...$valor_{width}$, con $fila_i \in$ [0, height], $valor_j \in$ [0 | 1 | ?]. Representa los valores iniciales para la fila i. Los valores se ingresan sin usar separadores.
- *localTransition*: transitionName. Indica la descripción de la especificación de comportamiento que se va a utilizar para la función de transición local en todas las celdas.
- *zone*: transitionName {$rango_1$..$rango_n$}. Asocia el nombre de una especificación de comportamiento con las celdas incluidas dentro del rango definido por la sentencia. De esta forma, distintos subrangos pueden tener distinto comportamiento. Cada $rango_i$ tiene la forma (x,y), describiendo una celda, o la forma (x,y)..(z,w), describiendo una zona rectangular de celdas.

Figura 79. Parámetros de un modelo acoplado celular

Como fuera mencionado previamente, la definición de las reglas que describen el comportamiento de una celda se hace en forma independiente a los modelos celulares que la utilizan. Esto permite que más de un modelo celular utilice la misma especificación, como así también que varias zonas dentro de un espacio de celdas la utilicen sin necesidad de redefinirla. Para ello se utiliza un lenguaje de especificación de alto nivel, que permite definir distintos grupos dentro de la especificación, en los que cada componente del grupo es una regla con la siguiente sintaxis:

rule : resultado delay { condición }

Donde:
resultado: [0 | 1 | ?]
delay: INTEGER (expresado en milisegundos)
condición: la condición es una expresión lógica que responde a la siguiente gramática expresada en BNF.

```
BoolExp := IntRelExp | NOT BoolExp | BoolExp AND BoolExp | BoolExp OR BoolExp
IntRelExp := IdRef | IntExp OpRel IntExp
IntExp := IdRef | IntExp Oper IntExp
IdRef := CellRef | '(' BoolExp ')' | Constant | Function
Constant := Int | Bool
Function := TRUECOUNT | FALSECOUNT | UNDEFCOUNT
CellRef := '(' IntExp ',' IntExp ')'
OpRel := = | != | > | < | >= | <=
Oper := + | - | * | /
Int := [Sign] Digit {Digit}
Bool := 0 | 1 | t | f | ?
Sign := [+] | -
Digit ::= 0 | 1 | ... | 9
```

Figura 80. Sintaxis del lenguaje de especificación implementado.

Una vez finalizada la especificación y la definición de parámetros, deben ingresarse los eventos externos por medio de otro archivo de configuración, compuesto por una secuencia de líneas describiendo eventos con el siguiente formato:

HH:MM:SS:MS PORT VALOR

Aquí, HH:MM:SS:MS indica la hora en que ocurrirá el evento, *Port* indica el nombre del port por el cual se registrará la ocurrencia del evento, y *Valor*, que almacena el valor numérico asociado al evento. Por otro lado, el archivo de salida tiene el siguiente formato: **Hora Port Valor.**

También se puede generar, a pedido del modelador, un archivo de log que permite analizar el flujo de mensajes en el modelo simulado, con el fin de detectar errores en la especificación. Este archivo también fue usado con el fin de construir una herramienta básica de visualización del espacio de celdas. Se muestra el flujo de mensajes entre los modelos que integran la simulación. Cada línea del archivo muestra el tipo de mensaje, la hora a la que se produce, quien lo emite y el destinatario. Esta información es común a todos los mensajes. En caso de ser un mensaje **X** o **Y** aparecerá además el port y su valor. Para el mensaje **D** se agrega la hora del próximo evento, o "..." en caso de que la hora sea ∞. Los números que figuran junto al nombre del simulador son a solo efecto de información para el desarrollador.

Archivo de configuración de entrada
00:00:10:00 in 1
00:00:15:00 done 1
00:00:30:00 in 1

Archivo de Salida
00:00:01:00 out 0.000
00:00:02:00 out 1.000

Archivo de Log
Mensaje I / 00:00:00:00 / Root(00) para Acoplado(06)
Mensaje I / 00:00:00:00 / Acoplado(06) para Queue(04)
Mensaje I / 00:00:00:00 / Acoplado(06) para Transducer(01)
Mensaje I / 00:00:00:00 / Acoplado(06) para Generator(02)
Mensaje I / 00:00:00:00 / Acoplado(06) para Processor(03)
Mensaje D / 00:00:00:00 / Queue(04) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Transducer(01) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Generator(02) / 00:00:10:00 para Acoplado(06)
Mensaje D / 00:00:00:00 / Processor(03) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Acoplado(06) / 00:00:10:00 para Root(00)
Mensaje * / 00:00:10:00 / Root(00) para Acoplado(06)
Mensaje * / 00:00:10:00 / Acoplado(06) para Generator(02)
Mensaje Y / 00:00:10:00 / Generator(02) / out / 3.000 para Acoplado(06)
Mensaje D / 00:00:10:00 / Generator(02) / 00:00:10:00 para Acoplado(06)
Mensaje X / 00:00:10:00 / Acoplado(06) / in / 3.000 para Queue(04)
Mensaje D / 00:00:10:00 / Transducer(01) / ... para Acoplado(06)

Figura 81. Parámetros de un modelo acoplado celular

7. RESULTADOS OBTENIDOS. DISCUSION.

En este capítulo se resumirán algunas de las experiencias de desarrollo realizadas, y se estudiarán los resultados obtenidos en las mismas. En una primera fase, se hicieron estudios comparativos entre autómatas celulares sincrónicos y asincrónicos. Posteriormente, se desarrollaron ejemplos usando la herramienta GAD para simular modelos DEVS, y posteriormente se ampliaron los estudios a la herramienta CD++. En las próximas secciones se analizarán los resultados obtenidos en los distintos experimentos, y se discutirán los resultados obtenidos.

7.1 Experiencias con autómatas celulares sincrónicos y asincrónicos

Como primer experiencia se desarrollaron simuladores simples con el fin de hacer estudios comparativos entre autómatas celulares sincrónicos y asincrónicos con distintos tipos de características.

Los autómatas **sincrónicos** se construyeron siguiendo un mismo mapeo para todos los ejemplos. Por un lado, se definieron dos matrices idénticas, bidimensionales, de N posiciones por lado. Cada una de ellas fue declarada de un tipo de datos dado, dependiendo de la aplicación. Una de las matrices se usa para almacenar el estado actual de las celdas, y la segunda se usa para construir la próxima generación. El autómata ejecuta de acuerdo con las siguientes reglas:

Inicialización
Arreglo A = estado global inicial del modelo.
CLOCK = t

Transición de estado
Cambiar de A a B (o viceversa). Para ello usar dos punteros: ACTUAL = apunta a la que sea la actual; FUTURO: apunta a la otra.

Secuencia de cálculo
Recorrer la matriz ACTUAL en algún orden hasta que se hayan tratado todos las celdas.
Aplicar la función t a la lista de vecinos. Almacenarlo en FUTURO(i,j).
Cuando se hayan tratado todas las celdas, ejecutar una transición de estado.

Avance del tiempo
Incrementar CLOCK
Si CLOCK es menor que el tiempo de finalización, empezar todo de vuelta.

Figura 82. Comportamiento para autómatas celulares sincrónicos

En este caso no basta una sola copia del arreglo, ya que esto impediría obtener el estado global para el autómata: los vecinos se están usando para calcularlo, y una vez hecho el cómputo, puede perderse el estado original del espacio de celdas (dependiendo de la secuencia), produciendo un autómata celular no determinístico, tal como fuera analizado en la sección 2.3.1.

La ejecución comienza generando un estado inicial, en el que se definen estados específicos o aleatorios (según el estudio que se quiera llevar a cabo). En todos los casos se consideraron autómatas toroidales.

Para el caso de los autómatas celulares **asincrónicos**, se utilizaron las ideas básicas definidas en los modelos NEVS (analizados previamente en la sección 2.3.1). Por un lado, se define una matriz conteniendo los estados de las celdas; por el otro, se define una lista de próximos eventos que almacena pares de coordenadas. Esta lista contendrá almacenadas las coordenadas de las celdas que deben ser evaluadas en el momento de producir la siguiente generación. A su vez, esta generación inducirá cambios en otras celdas, cuyas coordenadas serán almacenadas en la lista.

Para calcular la próxima generación se recorren uno por uno todos los elementos de la lista, de forma tal de obtener las coordenadas de las celdas no latentes. Luego se analiza el estado actual de la celda y las reglas definidas para cada caso, y se genera el resultado en una lista de resultados nuevos. La información contenida en la matriz no se altera hasta completar el procesamiento de la generación actual. El nuevo estado para una celda de la generación actual que haya sido registrada en la lista de estados nuevos consistirá en la asignación de dicho estado a la misma celda de la matriz. Si

el cambio de estado de una celda afecta a sus vecinos, las coordenadas de sus vecinos deberán ser almacenadas en la lista de próximos eventos.

La ejecución comienza generando un estado global inicial, en el que se definen estados específicos o aleatorios, según el estudio que se quiera llevar a cabo. Los estados iniciales se vuelcan en la matriz, y se incluyen las coordenadas de estas celdas (y las de sus vecinos) en la lista. En este caso también se usaron espacios toroidales.

Se ejecutaron autómatas sincrónicos y asincrónicos para diversos ejemplos, incluyendo el Juego de la Vida, el modelo Wa-Tor y un modelo de tráfico de múltiples carriles y sentido único. A continuación se analizan los resultados obtenidos para cada uno de ellos.

Para empezar, se hicieron ejecuciones variadas de modelos sincrónicos y asincrónicos del Juego de la Vida. Básicamente se analizó el tiempo que insume el cálculo de las nuevas generaciones, según la cantidad de celdas activas en un momento dado. El objetivo es analizar el desempeño de ambas soluciones. Inicialmente, las pruebas se hicieron con un tamaño de matriz de 40 x 40 celdas, con un número variable de celdas iniciales activas, cuyos resultados se ven en la próxima figura.

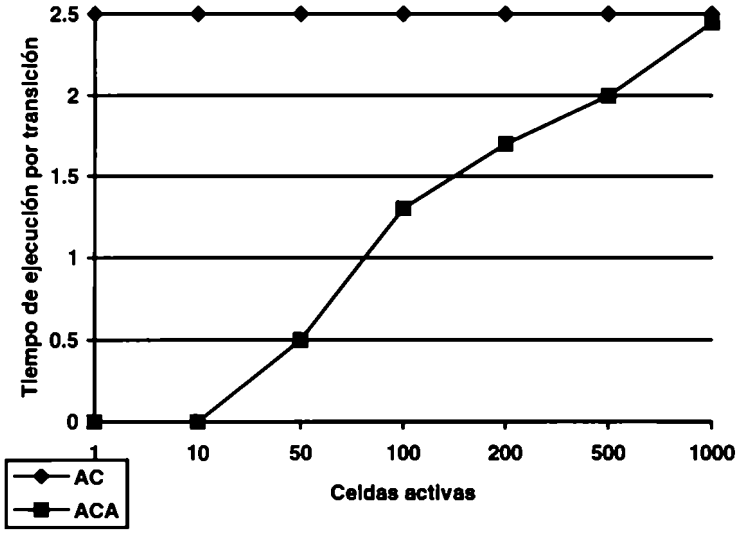


Figura 83. Resultados de ejecución del Juego de la Vida.

Puede notarse que el modelo sincrónico mantiene una demora constante en cada generación, ya que no se determinan cuáles son las celdas que deben cambiar de estado, y siempre se recorren todas las celdas de la matriz de la misma manera. En cambio, el modelo asincrónico arroja una función creciente dependiendo del número de celdas activas: si hay pocas celdas a verificar, el tiempo está en el orden de los milisegundos por transición; pero a medida que aumentan las celdas no latentes, el tiempo que insume el mantenimiento de la lista de cambios hace aumentar el tiempo de procesamiento de forma proporcional a la longitud de la misma.

En todos los casos se analizaron los peores casos de los tiempos de ejecución. Esta medida fue considerada pensando en los autómatas celulares asincrónicos, que se comportan mejor dependiendo del número de celdas activas. Mientras que en el caso de los autómatas celulares sincrónicos, los peores casos no difirieron de la media del tiempo de ejecución (un 3% de máxima diferencia), en el caso de los asincrónicos el desvío estándar llegó al 60%.

Como segunda fase de la experiencia, se realizaron estudios comparando el tiempo de ejecución de cada transición en un espacio de celdas de tamaño creciente, en el que se mantiene fijo el número de celdas, cuyos resultados se ven en la siguiente figura. En este caso se estudió el peor caso de los tiempos de ejecución partiendo con autómatas celulares con 1000 celdas activas.

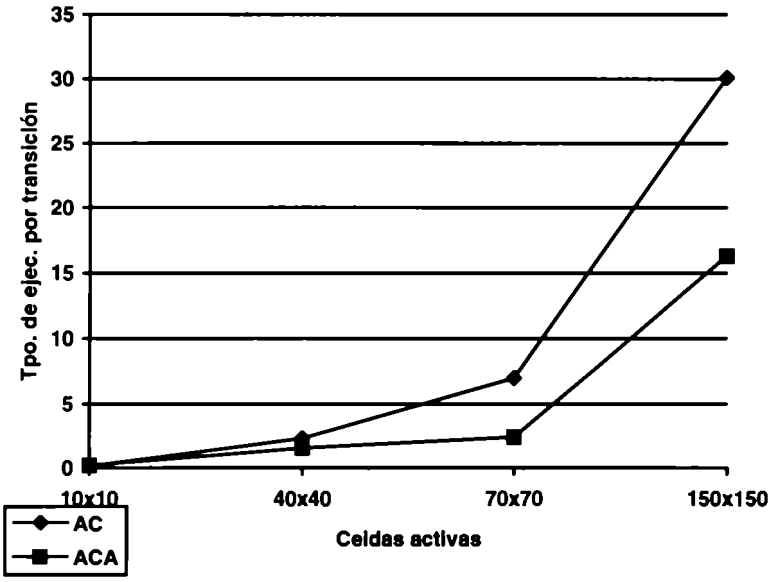


Figura 84. Tiempos de ejecución del Juego de la Vida con respecto a la variación en la dimensión del espacio.

Los resultados arrojados por las ejecuciones de los modelos sincrónico y asincrónico del modelo Wa-Tor fueron similares. La siguiente comparación muestra la cantidad de tiempo que insume el cálculo de las nuevas generaciones según la cantidad de celdas activas (es decir, las que tienen peces o tiburones). El modelo sincrónico mantiene una demora que varía muy levemente en función de la cantidad de celdas activas (0.1 seg. cada 250 celdas). Esto se debe a que no se determinan cuáles son las celdas que deben cambiar de estado, sino que siempre recorre todas las celdas de la matriz de la misma manera, pero de todas formas requiere más tiempo la administración de un tiburón o un pez que la de una celda vacía (buscar donde comer o moverse, reproducirse o morir, etc.). En cambio, el modelo asincrónico crece de manera continua pero más abrupta. Aproximadamente a las 400 celdas activas, deben incluirse gran cantidad de coordenadas a la lista de cambios, por lo que los tiempos de administración de la lista aumentan y el modelo sincrónico pasa a ser más eficiente.

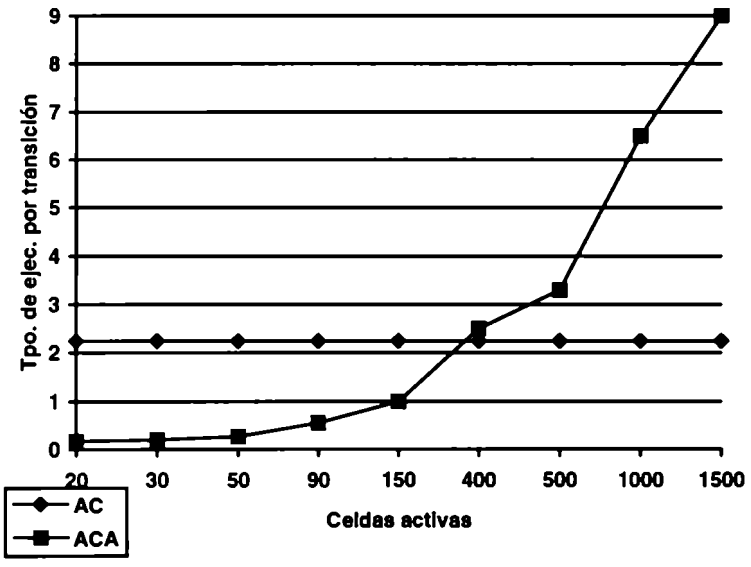


Figura 85. Resultados de ejecución de Wa-Tor.

Finalmente, se simuló un conjunto de automóviles desplazándose en carriles múltiples con un sentido único. Los automóviles se desplazan hacia adelante con una velocidad constante (dependiente de una demora de cada una de las celdas), y si la celda del frente está bloqueada, trata de sobrepasarlo, primero por la izquierda, y si está bloqueada, por la derecha. Si ambas están bloqueadas, el coche se queda en la misma celda que estaba.

En los siguientes gráficos se muestran los tiempos de ejecución para modelos sincrónicos y asincrónicos, considerando en cada caso un número variable de vehículos (que se muestran en el eje x), para una grilla de 80 celdas por 10 carriles.

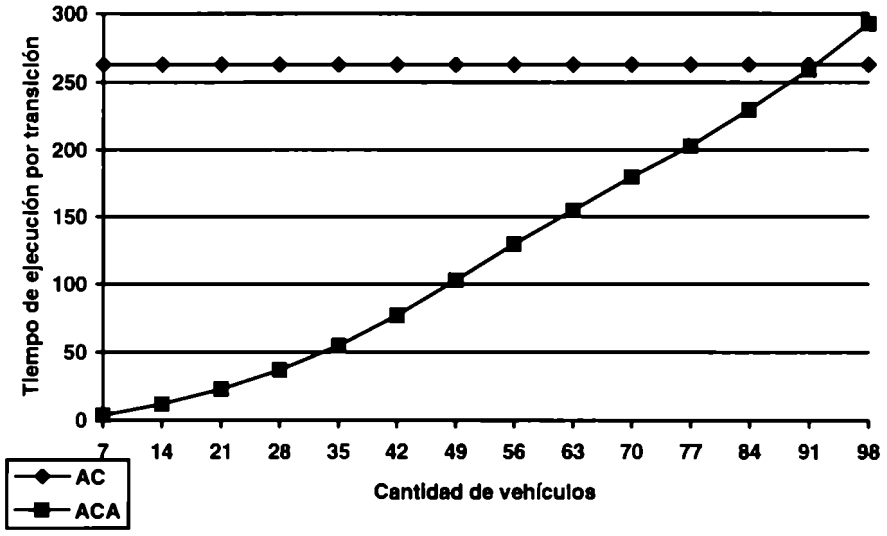


Figura 86. Comparación entre autómatas celulares sincrónicos y asincrónicos. Modelo de circulación de múltiples vías en una sola mano. Demora entre ejecuciones: 50 ms.

Posteriormente se analizó la influencia de la variación de rendimiento dependiendo del tiempo de cálculo de las funciones de transición local. Esto permite analizar la influencia del luso de funciones locales más complejas. Vemos que en estos casos las diferencias son aún mayores.

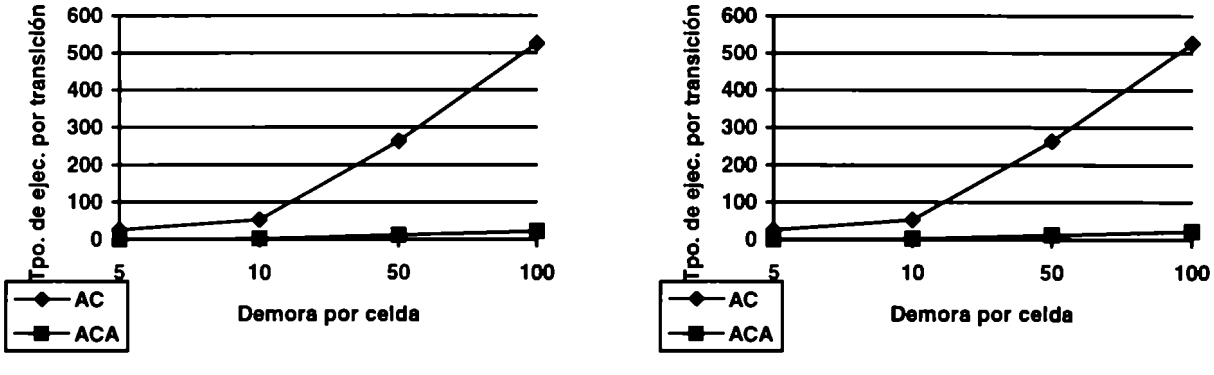


Figura 87. Comparación entre autómatas celulares sincrónicos y asincrónicos con distintas demoras por celda. (a) 7 vehículos; (b) 49.

A continuación se hizo un estudio pormenorizado de la cantidad de memoria utilizada por ambas soluciones. En muchos casos será preferible usar la aproximación asincrónica, ya que en este caso, el uso de una lista en memoria dinámica permite mejor utilización de la memoria disponible (sólo es necesaria una matriz de estados, en comparación con las dos usadas para el caso sincrónico). A pesar de ello, si el número de celdas activas crece, es necesario mantener información sobre las celdas no latentes (que en el peor de los casos pueden ser todas), y puede producirse una peor utilización de la memoria que en el caso sincrónico. Por otro lado, el uso de memoria dinámica para la lista de próximos eventos produce un mayor overhead en el uso del procesador, deteriorando los tiempos de ejecución de las soluciones asincrónicas. Por ende, se debe ser cuidadoso e intentar usar una estructura de datos estática en los casos en los que los requerimientos de memoria no sean prohibitivos.

El uso de la memoria fue estimado de la siguiente forma: si se llama N al tamaño del autómata en cada una de sus dimensiones, S al tamaño de un puntero, y T al tamaño de la estructura de datos necesaria para registrar el estado de una

celda, se pueden calcular cuatro estimaciones, que serán denominadas MACS (uso de Memoria de un Autómata Celular Sincrónico), MAE (uso de Memoria de un autómata celular Asincrónico con una estructura de datos Estática para la lista de próximos eventos), MADmin y MADmax (el mínimo y máximo de la Memoria usada por un autómata Asincrónico usando una estructura de datos Dinámica). Estos se calculan como:

- $MACS = 2 * N^2 * T$
- $MAE = 2 * N^2 * T$
- $MADmin = N * T$
  $MADmax = N^2 * (T+S)$

En la siguiente figura se muestra una comparación entre estas cuatro soluciones para distintos tamaños de espacios de celdas en el Juego de la Vida implementado.

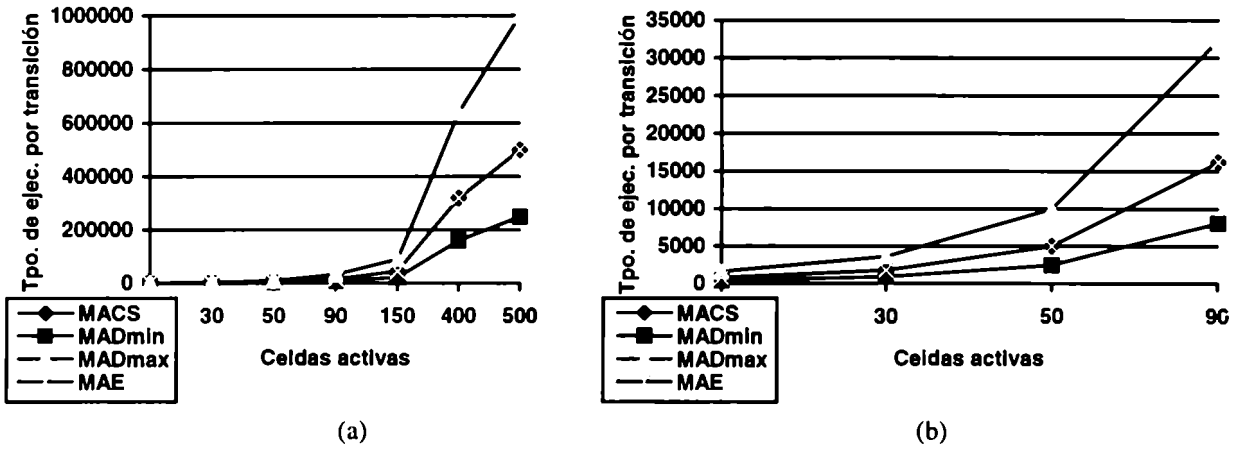


Figura 88. (a) Peores casos de uso de memoria para el Juego de la Vida. (b) Detalle.

Los resultados para los autómatas celulares varían de forma radical si se considera, en lugar de los peores casos posibles de activación para el autómata, la media calculada estadísticamente entre las ejecuciones realizadas. Ello se debe a que un número inferior de elementos promedio en la cola de próximos eventos reduce el overhead de memoria. Los resultados obtenidos para estos casos pueden verse en la siguiente figura.

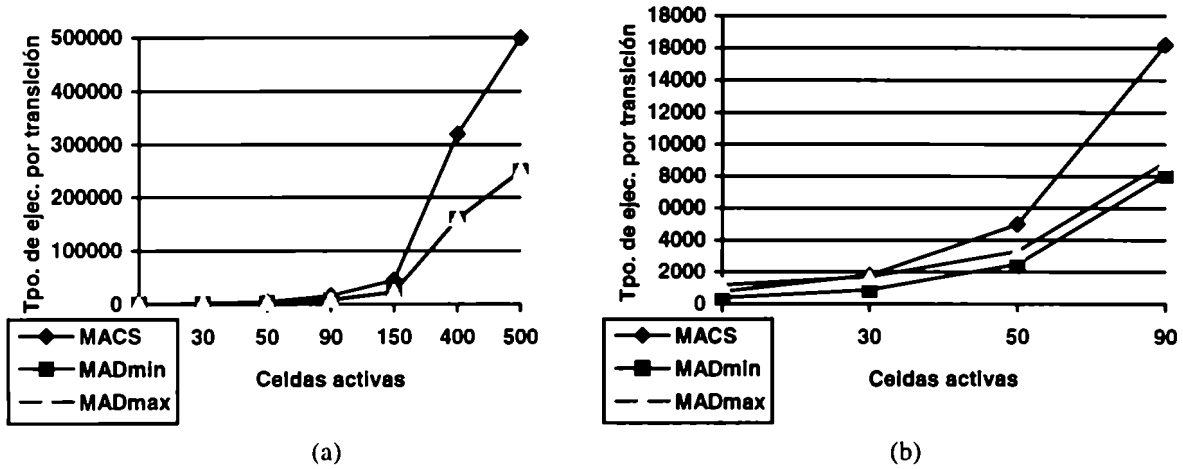


Figura 89. (a) Casos promedio para 1000 celdas iniciales. (b) Detalle.

Como puede verse, no se puede afirmar absolutamente que un método sea mejor que otro en todos los casos. Con una cantidad moderada de celdas activas, las soluciones asincrónicas presentan mejor performance. Las diferencias son

mayores aún cuando el tamaño del espacio de celdas aumenta. Con una cantidad relativamente grande, los sincrónicos presentan mayor rendimiento.

Pudo verse que en los modelos sincrónicos, el tamaño de la matriz es la principal causa de la demora en el cálculo de cada generación, a diferencia de los modelos asincrónicos, en los cuales este factor no es relevante. En este caso, los modelos sincrónicos presentan una demora casi constante en el cálculo de cada generación, y la cantidad de celdas activas prácticamente no incide en ella.

Por otro lado, los modelos asincrónicos generan una lista de próximos eventos que crecerá y decrecerá abruptamente durante la evolución del AC, y esta lista requiere un mantenimiento costoso (por ejemplo, verificar duplicados al agregar cada elemento). Pudo verse que los autómatas celulares cuya vecindad tienen menor tamaño (por ejemplo, la simulación de tráfico tiene seis vecinos, en lugar de los nueve que tienen los otros ejemplos) tienen mejor desempeño con respecto al tiempo de procesamiento, ya que este overhead se reduce.

Por el contrario, en los modelos en los que el cambio de una celda no afecta directamente a su vecindad, la lista requiere un mantenimiento menor. En los modelos asincrónicos, la longitud de la lista y el mantenimiento de la misma son las principales causas de la demora en el cálculo de las generaciones.

La decisión de implementar un método sincrónico o uno asincrónico tendrá fuertes consecuencias en el desempeño de la simulación. Previamente se deben analizar con cuidado las características del sistema y su evolución, y estos factores son los que guiarán hacia el método idóneo. Por ende, el uso del formalismo Cell-DEVS puede no ser adecuado para todos los casos.

A pesar de esto, en todos los casos analizados, el desempeño de los autómatas celulares asincrónicos fue muy superior al de los sincrónicos. Esto se debió a que dadas las características de las aplicaciones, el número de celdas activas se mantuvo bajo. Para que una solución sincrónica sea más eficiente que una asincrónica, se precisa que la proporción de celdas activas en el total se mantenga alta, lo cual no ocurre en la mayor parte de las aplicaciones que se desean modelar con este formalismo (sistemas ecológicos, diseño de circuitos, etc.).

En el caso particular de simulaciones de tráfico, el mayor grado de actividad en el autómata asincrónico ocurre cuando hay un número intermedio de vehículos. Esto se debe a que si el número de automóviles es alto, se producen situaciones de embotellamiento, que provocan un gran número de estados latentes, lo mismo que ocurre cuando la cantidad de automotores es baja. Pudo verse que aún en los peores casos, los autómatas asincrónicos tuvieron mejor desempeño que los sincrónicos para este tipo de aplicaciones.

En el caso del juego de la Vida se pudieron obtener resultados similares a los recién analizados para simulación de tráfico. Esto ocurre porque un número alto de celdas activas provoca un gran número de estados inmóviles. Los peores casos para los autómatas asincrónicos se obtuvieron para el modelo Wa-Tor. Aquí, el mayor número de peces o tiburones sólo provoca mayor actividad de movimiento, reproducción, alimentación y muerte. Cuanto mayor sea el número de animales, empeorará el desempeño, ya que se provoca actividad intensa en la lista de eventos. A pesar de ello, se debe considerar que los resultados obtenidos son buenos incluso para esta aplicación, ya que el número de animales en el océano, en proporción con el espacio disponible, no puede alcanzar los números para los cuales la solución sincrónica fue mejor. Ello implicaría un caso de superpoblación de alguna de ambas especies, lo que provocaría muertes de la especie superpoblada y una reducción automática de los animales en el espacio de celdas simulado, obteniendo finalmente un número bajo de celdas activas.

Otra ventaja obvia de las aproximaciones asincrónicas es su simplicidad para paralelizar la aplicación, obteniendo aún mayor aceleración en la simulación. Un autómata sincrónico puede paralelizarse, pero el alto grado de sincronización necesario aumentará el overhead desperdiciado en intercambio de mensajes, empeorando los resultados de una solución centralizada.

Con respecto al uso de memoria, puede verse que, a pesar de que, en teoría los resultados obtenidos son favorables para el caso sincrónico, la ejecución de casos reales permitió ver que el desempeño es mejor en el caso asincrónico. Los resultados favorables obtenidos para los autómatas asincrónicos son aún mucho mejores si se consideran espacios de celdas mayores, en los que el porcentaje de celdas activa sea menor. Por ende, se considera que el formalismo Cell-DEVS y su implementación pueden provocar mejoras de eficiencia en la simulación.

7.2 Experiencias con modelos clásicos DEVS

Para ilustrar los procedimientos utilizados para implementar modelos DEVS en la herramienta GAD, se detallarán dos ejemplos. En el primero se podrá observar un modelo tradicional implementando el comportamiento de un buffer que contiene una cola. Luego se analizará un ejemplo de acoplamiento de dicha cola en un modelo más complejo.

En este caso se analiza la implementación de una cola con política FIFO, tal como la presentada en la sección 2.2. La cola posee tres ports de entrada y uno de salida. Los datos de entrada arriban en el port **in**. Por cada valor que se envía por el port **out** se esperará la confirmación de su recepción en el port **done.** Si el cliente de la cola se encuentra saturado, podrá manejar el control de flujo a través del port **stop-send**.

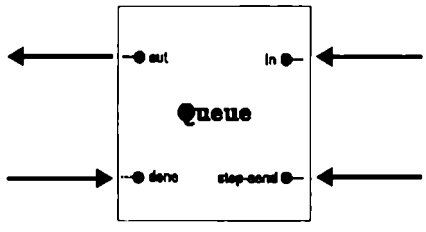


Figura 90. Modelo Atómico Queue

Como fuera mencionado, el primer paso para implementar un nuevo modelo atómico en la herramienta es definir una clase derivada de Atomic, que redefina los métodos para el manejo de transiciones. En el siguiente ejemplo se ve la definición de variables para hacer referencia a los ports de la cola y el tiempo que demora en preparar el dato antes de enviarlo por el port de salida. Además se encuentra la definición de una lista de valores para albergar los valores ingresados y el tiempo restante ante una interrupción por control de flujo.

```
class Queue : public Atomic {
public:
  Queue(); // Default constructor
  virtual string className() const ;

protected:
  Model &initFunction();              // Declaración de funciones de manejo del modelo
  Model &externalFunction( const ExternalMessage & );
  Model &internalFunction( const InternalMessage & );
  Model &outputFunction( const InternalMessage & );

private:
  const Port &in;                     // Ports de E/S
  const Port &stop;
  const Port &done;
  Port &out;
  Time preparationTime;               // Parámetro

  typedef list<Value> ElementList;  // Variables de estado
  ElementList elements ;

  Time timeLeft;

}; // class Queue
```

Figura 91. Definición del modelo atómico Queue.

En la siguiente figura se muestra la definición de las estructuras de datos usadas por el modelo, y sus métodos de manejo.

```
Queue::Queue()
  : preparationTime( 0, 0, 10, 0 ), in( this->addInputPort( "in" ) ), stop( this->addInputPort( "stop" ) )
  , done( this->addInputPort( "done" ) )
  , out( this->addOutputPort( "out" ) )        // Definición de los ports de E/S y parámetros por default
{
  this->description( "Queue" ) ;               // Definición del identificador del modelo

  string time( Simulator::Instance().getParameter(this->description(), "preparation" ) ) ;   // Carga del parámetro

  if( time != "" )
    preparationTime = time ;
}
```

Figura 92. Constructor para la clase Queue.

En este caso, el constructor de la clase deberá crear los tres ports de entrada, el port de salida e inicializar la variable *preparationTime*. El parámetro *preparation* puede estar especificado en el archivo de configuración del modelo, y de ser así, el tiempo de preparación puede ser modificado.

```
Model &Queue::initFunction() {
  elements.remove( elements.begin(), elements.end() ) ;        // Vaciar la cola
  return *this;
}

Model &Queue::externalFunction( const ExternalMessage &msg ) {
  if( msg.port() == in ) {                                     // Llega un evento por el port 'in
    elements.push_back( msg.value() ) ;                        // Encolar el nuevo elemento
    if( elements.size() == 1 ) this->holdIn( active, preparationTime );   // Si es el único, prepararlo para transmitir
  }

  if( msg.port() == done ) {                                   // El mensaje fue trnsmitido
    elements.pop_front() ;                                     // Eliminarlo de la cola
    if( !elements.empty() )
      this->holdIn( active, preparationTime );                 // Si hay otro elemento, prepararse para transmitirlo
  }

  if( msg.port() == stop )                                     // Frenar la actividad de la cola
    if( this->state() == active && msg.value() ) {             // Un valor distinto de cero indica que se debe parar
      timeLeft = msg.time() - this->lastChange();              // Calcular el tiempo restante
      this->passivate();                                       // Poner al modelo en estado pasivo
    }
    else
      if( this->state() == passive && !msg.value() )           // Un valor cero indica continuar
        this->holdIn( active, timeLeft );                      // Se ejecuta el tiempo restante
  return *this;
}

Model &Queue::outputFunction( const InternalMessage &msg ) {
  this->sendOutput( msg.time(), out, elements.front() ) ;      // Se envía el valor del primer elemento
  return *this ;
}
Model &Queue::internalFunction( const InternalMessage & ) {
  this->passivate();
  return *this ;
}
```

Figura 93. Funciones de transición para la clase Queue.

Por otro lado, la función de inicialización debe borrar la lista que guarda los elementos encolados. Si en este momento se deseara planificar el próximo cambio de estado, se deberá invocar al método *holdIn*, que mantiene al método activo

durante el tiempo especificado. En cambio, en este caso el próximo cambio de estado se producirá ante la llegada de un evento externo.

Con respecto a la ejecución de la función de transición externa, la clase Queue tiene tres puertos de entrada por los cuales puede recibir eventos externos. Si un evento arriba por el port *in* significa que es un dato nuevo de entrada, por lo tanto debe encolarse. Si es el único en la cola se debe preparar el dato para ser enviado. Si un evento arriba en el port *done* significa que el último elemento enviado fue recibido correctamente, por lo tanto debe ser eliminado de la cola. Si hay más elementos a enviar se debe preparar el próximo dato para ser enviado. Por último el evento puede arribar en el port *stop* indicando que se desea pausar o recomenzar el trabajo de la cola. Si la cola se encuentra en estado *active* y el valor del mensaje es distinto de cero esto indica que se desea hacer una pausa, por eso se calcula cuanto tiempo faltaba procesar para el próximo cambio de estado (fin de la preparación del dato a enviar) y luego la cola cambia su estado a *passive* llamando al método *passivate*. Si la cola se encuentra en estado *passive* y el valor del mensaje es cero la cola vuelve a estar activa planificando el próximo cambio de estado en base al tiempo que restaba procesar cuando la cola fue pausada.

La función de salida se invoca antes de la de transición interna, e indica que debe enviarse el valor del tope de la cola por el puerto de salida ya que finalizó el intervalo de preparación del dato.

Finalmente, la invocación de la función de transición interna indica que se cumplió el tiempo de preparación (la función de salida fue invocada previamente), por lo tanto debe esperar un evento en el puerto *done* indicando la correcta recepción del dato.

Una vez finalizado el nuevo modelo se incorpora por vinculación estática a la biblioteca de simulación, generando un nuevo simulador que tiene incorporado al nuevo modelo atómico **Queue**. Para comenzar la simulación, se debe definir un archivo que contiene la descripción de los componentes de la misma y los enlaces entre los modelos.

```
[top]
components : queue@Queue
out : out
in : in done stop
Link: in in@queue
Link : out@queue out
Link : done done@queue
Link : stop stop@queue

[queue]
preparation : 0:0:20:0
```

Figura 94. Archivo de definición de componentes y enlaces para la clase Queue.

Supongamos ahora que se desea modelar una unidad de procesamiento que recibe trabajos con una cierta distribución, los encola con política FIFO y luego los consume. Para simular este comportamiento se definieron dos componentes: un generador de trabajos (*Generator*, modelo atómico) y un *Consumidor* (modelo acoplado). Por ende, el entorno a modelar incluirá un usuario que provee tareas a ejecutar, un planificador de tareas con cierta política, y un procesador, siguiendo el Ejemplo 8. Cuando llega una nueva tarea, el planificador decidirá cuánto tiempo ejecutar para comenzar su procesamiento. Cuando la tarea comienza, ejecutará durante una cantidad fija de tiempo. El planificador se considerará sin desalojo (la tarea ejecuta sin ser interrumpida) usando una política FIFO.

En este caso, la simulación ejecutará un modelo compuesto por cuatro modelos atómicos, cada uno de los cuales representa una tarea distinta dentro del entorno. El generador está dedicado a proveer un marco experimental para generar nuevas tareas. El modelo Queue (que ya fue analizado) permite simular el comportamiento del planificador de tareas. El tercer modelo (*Processor*) se usa para especificar la ejecución del procesador. Finalmente, el modelo transductor (*Transducer*) está dedicado a registrar las mediciones generadas por la simulación.

El comportamiento genérico para cada uno de los modelos es el siguiente:

Generator: el modelo genera nuevas tareas en la simulación, y se usa como parte del marco experimental. Tiene un port de salida (**out**) que se usa para obtener las tareas que deben ser procesadas, de las cuales cada valor generado representa el identificador de una tarea (que es único durante el proceso de simulación). El intervalo de creación del proceso se

genera usando números aleatorios con distribuciones de probabilidad Poisson y χ^2 (que pueden ser seleccionadas durante el proceso de configuración).

Processor: este modelo es responsable de simular la ejecución de las tareas. Una tarea que llega se recibe a través del port de entrada **in**, y cuando ha terminado, debe emitir el identificador del proceso a través del port **out**. Cuando una tarea se está ejecutando, el procesador permanece ocupado hasta que el procesamiento haya terminado. El tiempo de procesamiento se genera usando números aleatorios con distribución de probabilidad exponencial.

Queue: como el procesador ejecuta de a una tarea por vez, la cola recibe las nuevas tareas y las almacena mientras el procesador está ocupado. Esta cola fue implementada usando una política FIFO sin desalojo.

Transducer: este modelo está encargado de recolectar las métricas producidas por la simulación, y de calcular estadísticas. Se han considerado dos variables: rendimiento (*Throughput*, medido como el número de tareas ejecutadas por unidad de tiempo) y uso de CPU (*CpuUsage*, calculado como un promedio de las tareas esperando en la cola de listos).

El código de estos modelos especificados usando la herramienta puede verse en el Apéndice II.

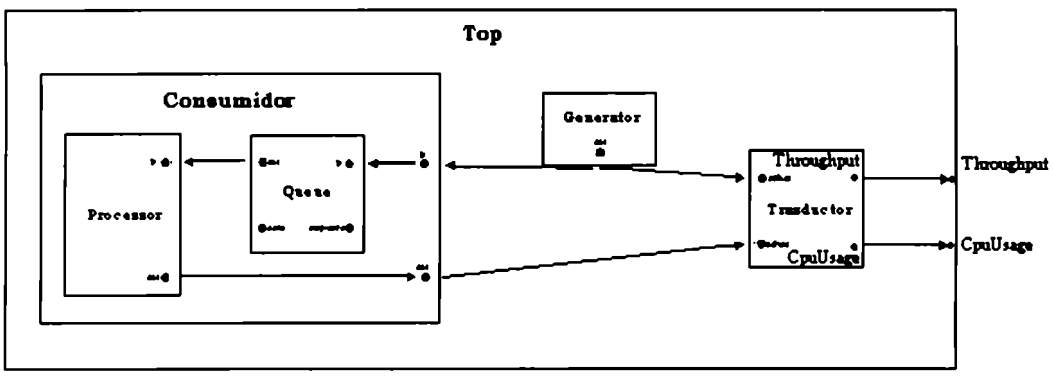


Figura 95. Acoplamiento de los modelos [BBW98a].

```
[top]
components : transducer@Transducer generator@Generator Consumidor
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumidor
Link : out@Consumidorr solved@transducer
Link : out@transducer out

[Consumidor]
components : queue@Queue processor@CPU
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
[transducer]
frequence : 0:0:2:0

[generator]
distribution : poisson
mean : 10

[processor]
distribution : exponential
mean : 10

[queue]
preparation : 0:0:20:0
```

Figura 96. Definición del acoplamiento en la herramienta

Los modelos se acoplan formando un modelo multicomponente, cuya interconexión se puede ver en la Figura 95. Como puede verse, la salida del Generador está conectada con la cola de listos (para registrar una nueva tarea), y con el Transductor (que almacena la hora del nuevo proceso para calcular estadísticas). Cada vez que termina una tarea, se sacan los resultados usando el port **out** de la cola, y se ingresan en el modelo Processor para que se ejecuten. Luego que una tarea ha terminado, el Processor saca el número de la tarea a través de su port **out**, y se envía a la cola y al transductor. Este último almacena información acerca de los procesos, y manda los resultados a través de los ports *Throughput* y *CpuUsage*. El acoplamiento entre los modelos definidos usando se incluye a continuación.

Sobre estos modelos se ejecutaron diversas pruebas con distintas distribuciones probabilísticas, variando los parámetros. Los resultados obtenidos permitieron ver que el desempeño del procesador (medido en tiempos de respuesta y en rendimiento) mejoró cuando los tiempos promedio de ejecución fueron inferiores a los tiempos de generación de nuevas tareas [BBW98a].

La siguiente figura muestra los resultados que se obtuvieron usando un generador con distribución de probabilidades Poisson (con 10 unidades de tiempo de promedio), y distribución χ^2 para el tiempo de procesamiento (con 30 unidades de tiempo de promedio). Como resultado, las tareas quedan encoladas esperando el procesador. El rendimiento es de 2 tareas por minuto (el tiempo medio de procesamiento es de 30 segundos, y el período considerado es de un minuto). Chequeos similares fueron ejecutados cambiando la distribución probabilística y sus parámetros, obteniendo siempre resultados similares.

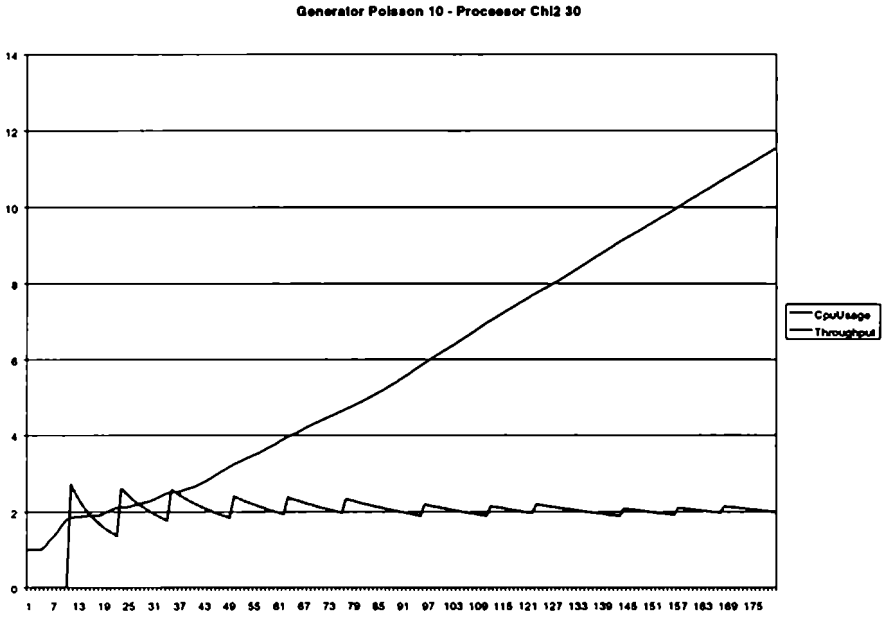


Figura 97. Generación de trabajos con mayor frecuencia que la de consumo [BBW98a].

Los resultados de la siguiente figura fueron obtenidos usando un generador con probabilidad de distribución Poisson (30 segundos de promedio) y un tiempo de procesamiento con distribución exponencial (con 10 unidades de tiempo promedio). En este caso, el rendimiento también tiende a 2 tareas por minuto, pero el procesador queda libre la mayoría del tiempo. Esto ocurre debido a que los trabajos son consumidos a mucho mayor velocidad que la generación de nuevas tareas.

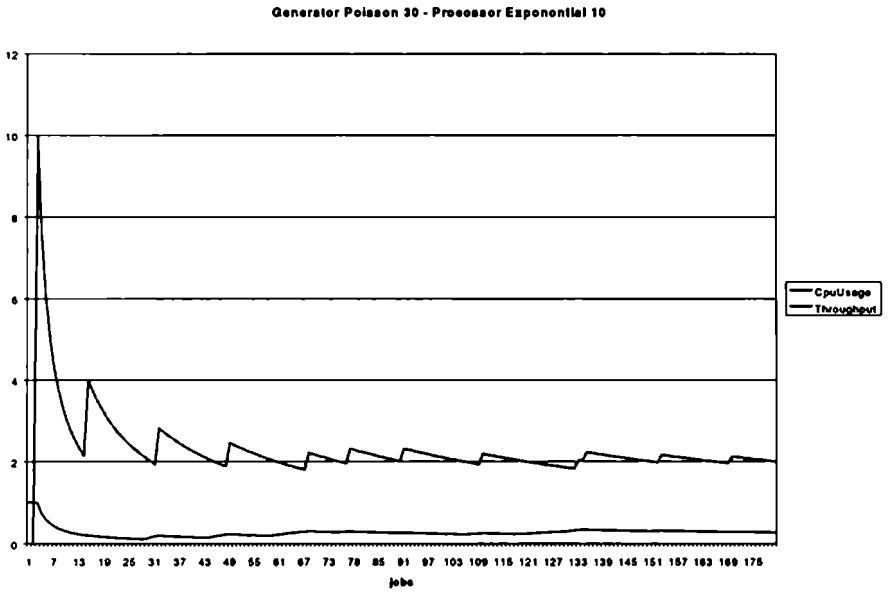


Figura 98. Generación de trabajos con menor frecuencia que el consumo [BBW98a].

Estas experiencias permitieron mostrar la utilidad de la herramienta GAD al desarrollar simulaciones con distintas instancias para un mismo modelo, reduciendo el tiempo de desarrollo. El disponer de una herramienta que sigue el formalismo DEVS, en la que se pueden desarrollar modelos atómicos modulares simplemente redefiniendo algunos métodos, permite reducir los tiempos de desarrollo y chequeo. Algunos de estos tiempos serán analizados al finalizar este capítulo. Por otro lado, se pudo probar la reutilización de modelos chequeados; en este caso, el modelo Queue, que originalmente fuera especificado para modelizar una cola, fue reutilizado para modelizar una cola de listos sin necesidad de hacer modificaciones.

7.3 Experiencias con modelos Cell-DEVS

Como fuera mencionado, la herramienta CD++ permite la especificación de modelos Cell-DEVS que pueden ser simulados. Para empezar, se permite describir el comportamiento de cada una de las celdas por medio de un lenguaje de expresiones lógicas a través del cual se define la función de transición local. El tamaño del espacio celular, el tipo de delay que usará el modelo y si tiene bordes o es circular, también son parámetros de la especificación. Para completarla es necesario definir cuál es el vecindario de cada celda. Con esta información la herramienta puede construir el modelo celular junto con los vínculos que lo unen con sus vecinos. A partir de esta definición se puede ver al modelo celular como a un modelo acoplado con una cantidad arbitraria de modelos atómicos idénticos, todos vinculados de acuerdo a la definición del modelo Cell-DEVS.

Dentro de la ejecución del espacio de celdas, la función de transición local de cada celda es la que determina cuál es el valor futuro que tendrá la celda. Generalmente, este resultado se calcula usando como entrada los valores emitidos por las celdas del vecindario de la celda en cuestión. Además de obtener como resultado el nuevo estado de la celda la ejecución de la función de transición local definida, considera el tiempo que la celda deberá demorarse (valor de *delay*) antes de producir su cambio de estado interno. El tipo de delay (*inertial* o *transport*) se especifica para el autómata celular indicando que todas sus celdas tienen el mismo tipo de *delay*.

Como primera experiencia de modelado Cell-DEVS, se definió una especificación del Juego de la Vida, cuya definición puede verse en la siguiente figura.

```
[top]
components : life
[life]
type : cell
width : 20
height : 15
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1,-1) life(1,0) life(1,1)
initialvalue : 0
initialrowvalue : 0 01111000111100011110
initialrowvalue : 1 00010001111000000000
initialrowvalue : 2 00110111100010111100
initialrowvalue : 3 00110000011110000010
initialrowvalue : 4 00101111000111100011
initialrowvalue : 10 01111000111100011110
initialrowvalue : 11 00010001111000000000
initialrowvalue : 12 00110111100010111100
initialrowvalue : 13 00110000011110000010
initialrowvalue : 14 00101111000111100011
localtransition : life-rule
[life-rule]
rule : 1 100 { (0,0) = 1 and truecount = 4 }
rule : 1 100 { (0,0) = 0 and truecount = 3 }
rule : 0 100 { t }
```

Figura 99. Especificación del Juego de la Vida usando la herramienta.

Este ejemplo fue simulado usando distintas aproximaciones. Por un lado, se hicieron simulaciones básicas del modelo usando el coordinador jerárquico. Los resultados de dicha ejecución pueden verse en la siguiente figura, que también fue generada usando la opción de graficación de la herramienta.

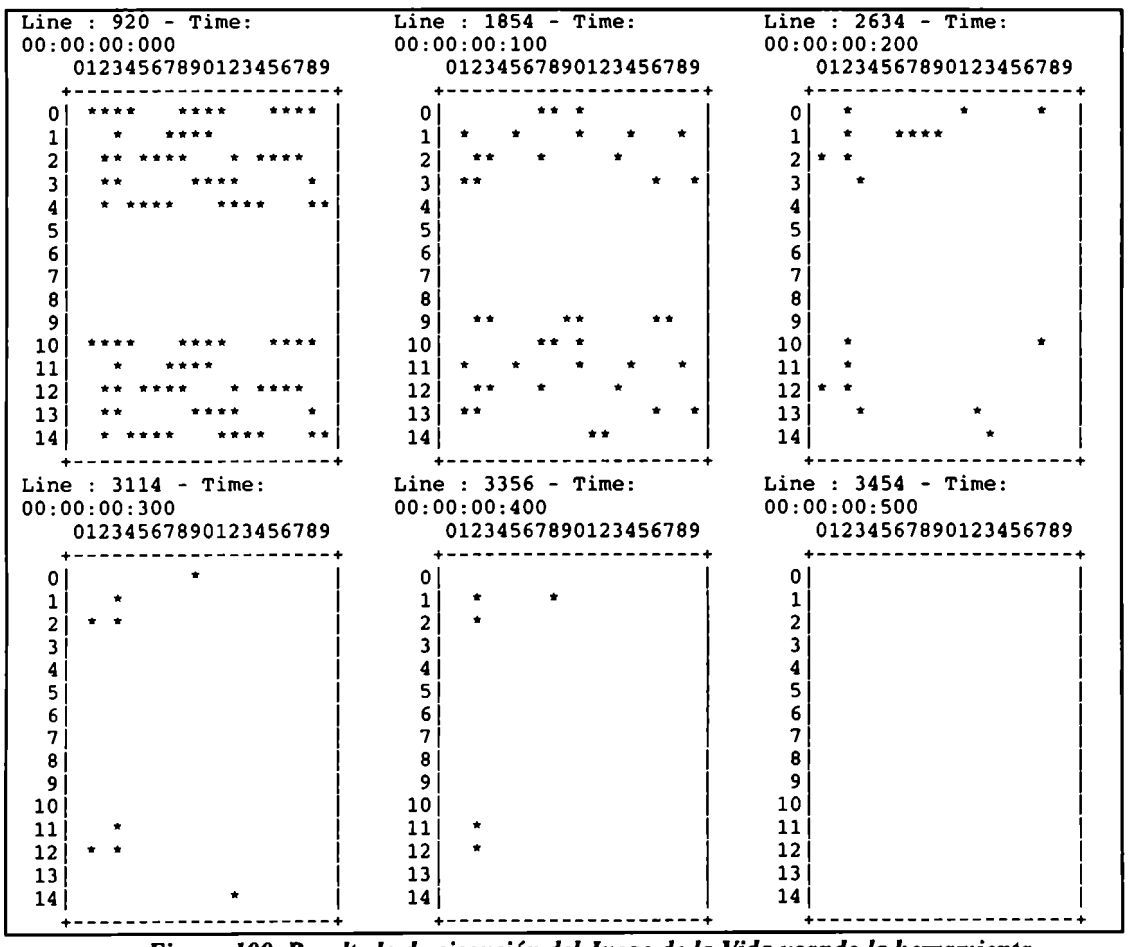


Figura 100. Resultado de ejecución del Juego de la Vida usando la herramienta.

Como resultado de la simulación, el espacio de celdas avanza siguiendo las reglas definidas en la especificación. En la siguiente figura se pueden ver algunas líneas del archivo de Log, que muestra los distintos mensajes emitidos durante la ejecución del ejemplo anterior.

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)                              (a)
Mensaje I / 00:00:00:000 / top(01) para life(02)
Mensaje I / 00:00:00:000 / life(02) para life(0,0)(03)
Mensaje I / 00:00:00:000 / life(02) para life(0,1)(04)
...
Mensaje I / 00:00:00:000 / life(02) para life(19,19)(402)
Mensaje D / 00:00:00:000 / life(0,0)(03) / ... para life(02)
Mensaje D / 00:00:00:000 / life(0,1)(04) / 00:00:00:100 para life(02)         (b)
Mensaje D / 00:00:00:000 / life(0,2)(05) / 00:00:00:100 para life(02)
...
Mensaje D / 00:00:00:000 / life(19,19)(402) / ... para life(02)
Mensaje D / 00:00:00:000 / life(02) / 00:00:00:100 para top(01)               (c)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:100 / Root(00) para top(01)                              (d)
Mensaje * / 00:00:00:100 / top(01) para life(02)
Mensaje * / 00:00:00:100 / life(02) para life(0,1)(04)
...
Mensaje * / 00:00:00:100 / life(02) para life(19,17)(400)
Mensaje Y / 00:00:00:100 / life(0,1)(04) / out / 0.000 para life(02)          (e)
Mensaje D / 00:00:00:100 / life(0,1)(04) / ... para life(02)
Mensaje Y / 00:00:00:100 / life(0,2)(05) / out / 0.000 para life(02)
...
Mensaje D / 00:00:00:100 / life(19,16)(399) / ... para life(02)
Mensaje Y / 00:00:00:100 / life(19,17)(400) / out / 1.000 para life(02)
Mensaje D / 00:00:00:100 / life(19,17)(400) / ... para life(02)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(19,0)(383)    (f)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(19,1)(384)
...
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 1.000 para life(18,18)(381)
Mensaje D / 00:00:00:100 / life(19,0)(383) / ... para life(02)
Mensaje D / 00:00:00:100 / life(19,1)(384) / ... para life(02)
Mensaje D / 00:00:00:100 / life(19,2)(385) / 00:00:00:100 para life(02)
...
Mensaje D / 00:00:00:100 / life(18,18)(381) / ... para life(02)
Mensaje D / 00:00:00:100 / life(02) / 00:00:00:100 para top(01)
Mensaje D / 00:00:00:100 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:200 / Root(00) para top(01)
Mensaje * / 00:00:00:200 / top(01) para life(02)
Mensaje * / 00:00:00:200 / life(02) para life(19,2)(385)
...
[Se repite el ciclo de simulacion]
Mensaje D / 00:00:00:500 / life(12,8)(251) / ... para life(02)
Mensaje D / 00:00:00:500 / life(12,9)(252) / ... para life(02)
Mensaje D / 00:00:00:500 / life(02) / ... para top(01)                        (g)
Mensaje D / 00:00:00:500 / top(01) / ... para Root(00)
```

Figura 101. Mensajes producidos durante la ejecución del modelo achatado.

Como fuera discutido en la sección 6.4, aquí se comienza enviando un mensaje de inicialización al modelo externo, que se encargará de distribuirlos por todo el espacio de celdas (ver (a) en la figura). Cada una de las celdas responde con un mensaje **done** (ver (b) en la figura), indicando la hora de su próximo evento interno (en este caso, 100 milisegundos, que es la duración de la demora definida en la especificación).

Cuando el coordinador tope recibe todos los mensajes **done** - (c) en la figura -, transmite al coordinador raíz la hora del hijo inminente, para que avance la hora global del simulador. Este coordinador responderá con un mensaje * (ver (d) en la figura), el que se difundirá entre las celdas inminentes. Estas se ejecutarán, transmitiendo mensajes **Y** y **done**.

Finalmente, se analiza qué salida de una celda no latente debe enviarse a las vecinas - (f) en la figura -, en forma de mensajes **X**. El ciclo se repetirá hasta que no existan más eventos externos, o cuando termine la hora de simulación (en este caso, 500 milisegundos, como puede verse en (g) de la figura).

Como siguiente fase, el mismo modelo fue ejecutado de forma achatada. Los resultados de esta ejecución pueden verse en la siguiente figura.

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
```

```
Mensaje I / 00:00:00:000 / top(01) para life(02)
Mensaje D / 00:00:00:000 / life(02) / 00:00:00:100 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:100 / Root(00) para top(01)
Mensaje * / 00:00:00:100 / top(01) para life(02)
Mensaje D / 00:00:00:100 / life(02) / 00:00:00:100 para top(01)
Mensaje D / 00:00:00:100 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:200 / Root(00) para top(01)
Mensaje * / 00:00:00:200 / top(01) para life(02)
Mensaje D / 00:00:00:200 / life(02) / 00:00:00:100 para top(01)
Mensaje D / 00:00:00:200 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:300 / Root(00) para top(01)
Mensaje * / 00:00:00:300 / top(01) para life(02)
Mensaje D / 00:00:00:300 / life(02) / 00:00:00:100 para top(01)
Mensaje D / 00:00:00:300 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:400 / Root(00) para top(01)
Mensaje * / 00:00:00:400 / top(01) para life(02)
Mensaje D / 00:00:00:400 / life(02) / 00:00:00:100 para top(01)
Mensaje D / 00:00:00:400 / top(01) / 00:00:00:100 para Root(00)
Mensaje * / 00:00:00:500 / Root(00) para top(01)
Mensaje * / 00:00:00:500 / top(01) para life(02)
Mensaje D / 00:00:00:500 / life(02) / ... para top(01)
Mensaje D / 00:00:00:500 / top(01) / ... para Root(00)
```

Figura 102. Mensajes producidos durante la ejecución del modelo achatado.

Como vemos, en este caso los mensajes transmitidos se han reducido notablemente, ya que sólo se produce una interacción entre los coordinadores de mayor nivel y el coordinador raíz, evitando toda otra interacción.

Como primer análisis, se realizaron estudios comparativos exhaustivos entre los coordinadores achatados y no achatados aplicados a este ejemplo. Se analizó la evolución de diez ciclos de simulación, usando ambas soluciones. Los resultados de este análisis pueden verse en la siguiente figura.

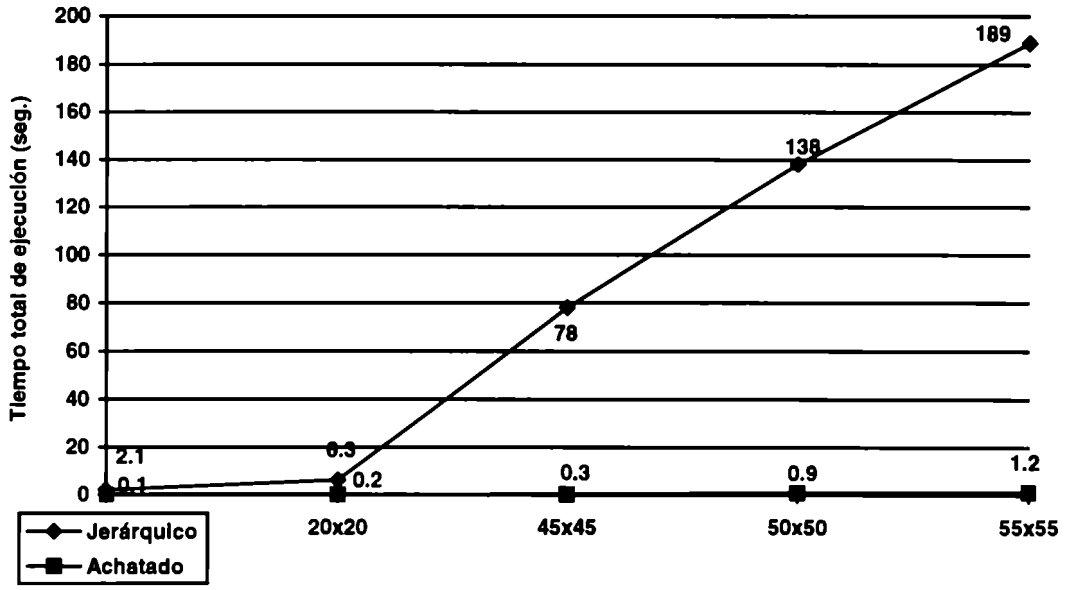


Figura 103. Resultados de ejecución del Juego de la Vida.

La figura muestra los resultados de los chequeos ejecutados. Se muestran los peores casos de ejecución de los ejemplos usando la herramienta, considerando distintos tamaños del espacio de celdas (comenzando con 100 modelos atómicos, y terminando con más de 2500). En todos los casos se comenzó la ejecución con aproximadamente el 75% del espacio de celdas activo.

A continuación se repitieron los chequeos para un espacio de celdas de 2500 modelos atómicos, pero variando la duración de la simulación. Los resultados pueden verse en la siguiente figura.

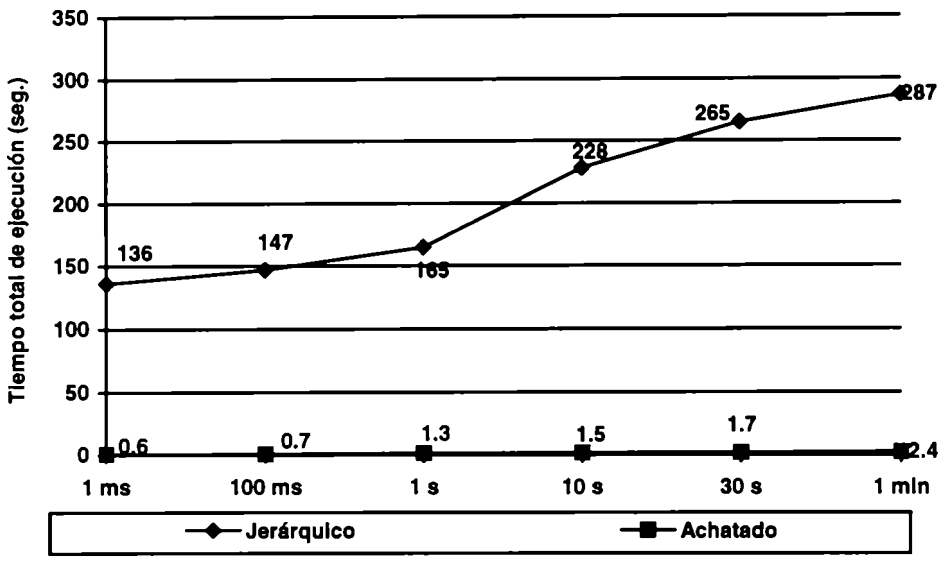
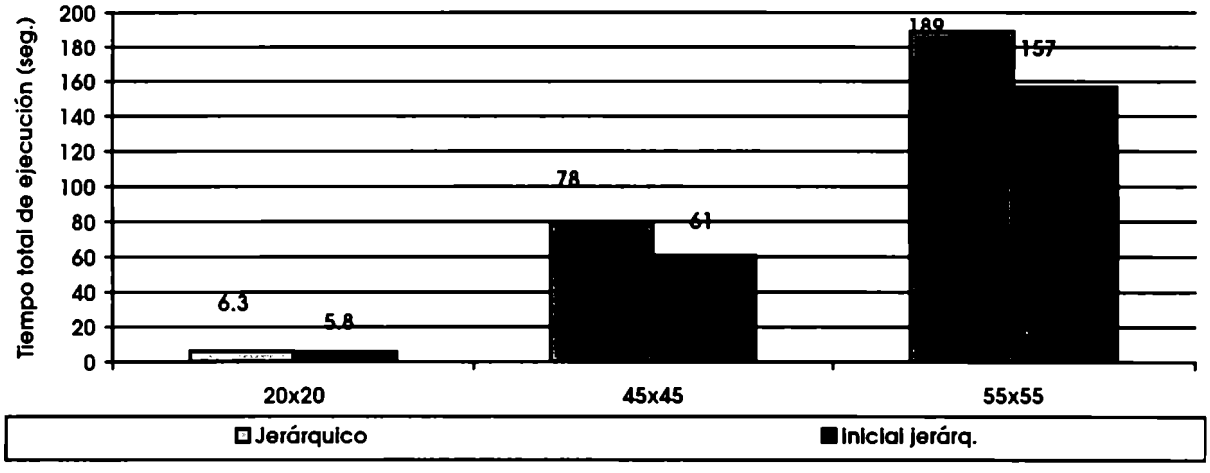
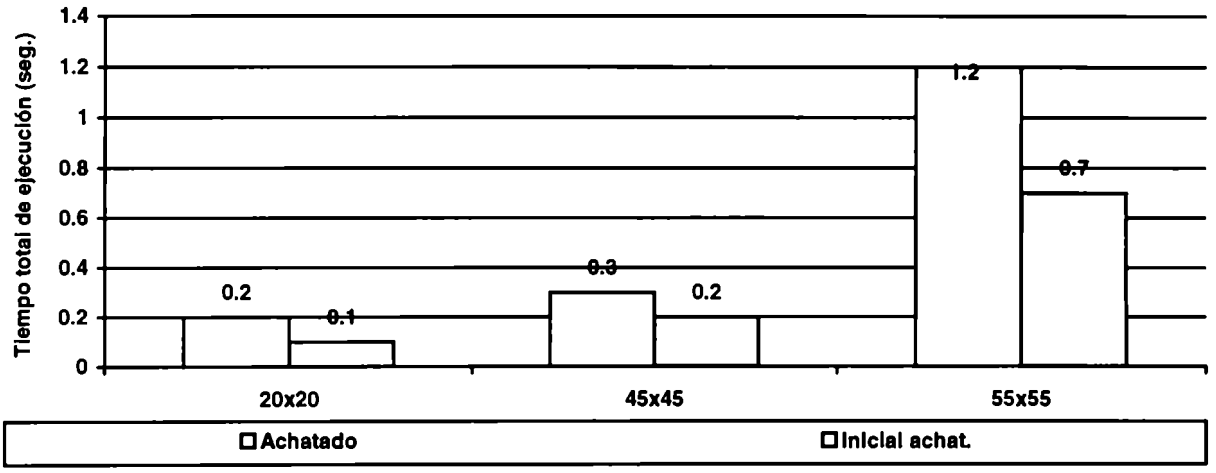


Figura 104. Resultados obtenidos variando la duración de la simulación.

En base a estos mismos chequeos se analizó la influencia del overhead gastado en mensajes de inicialización. Para ello se ejecutó la simulación terminándola instantáneamente luego de haber comenzado. Los resultados obtenidos fueron los siguientes.



(a)



(b)

Figura 105. Tiempos de inicialización (a) Jerárquico (b) Achatado.

Finalmente se hizo un estudio comparativo de la aplicación ejecutada usando el entorno versus la ejecución del mismo modelo implementado en el mismo procesador (con igual lenguaje de programación y sistema operativo), pero programando la aplicación desde el comienzo, sin la ayuda de la herramienta. Se programaron soluciones sincrónicas y asincrónicas, cuyos tiempos de ejecución fueron comparados con los modelos achatados. Los resultados pueden verse en la siguiente figura.

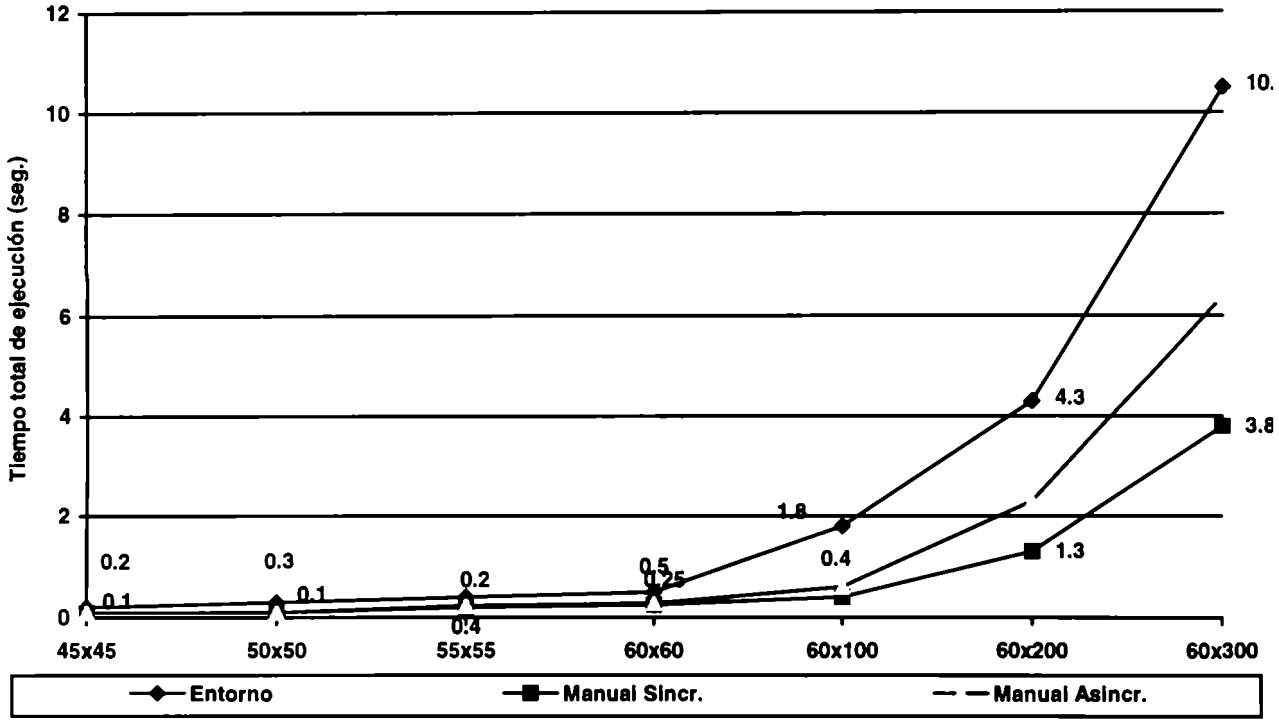


Figura 106. Resultados de ejecución del Juego de la Vida.

Posteriormente se analizó un modelo que representa el tráfico en una sola vía. La regla para describir este movimiento usando el lenguaje de especificación es similar a la que será presentada posteriormente, pero con el mismo comportamiento para todo el espacio de celdas y sin estados indefinidos.

Este modelo se ejecuta como fuera analizado en la sección 3.3. En la siguiente figura podemos ver el intercambio de mensajes entre los distintos submodelos.

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para cellcoupled(02)
Mensaje I / 00:00:00:000 / cellcoupled(02) para cellcoupled(0,0)(03)
...
Mensaje D / 00:00:00:000 / cellcoupled(0,0)(03) / 00:00:00:010 para cellcoupled(02)
...
Mensaje D / 00:00:00:000 / cellcoupled(02) / 00:00:00:006 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:006 para Root(00)
Mensaje * / 00:00:00:006 / Root(00) para top(01)
Mensaje * / 00:00:00:006 / top(01) para cellcoupled(02)
Mensaje * / 00:00:00:006 / cellcoupled(02) para cellcoupled(0,1)(04)
...
Mensaje * / 00:00:00:006 / cellcoupled(02) para cellcoupled(9,45)(498)
Mensaje * / 00:00:00:006 / cellcoupled(02) para cellcoupled(9,48)(501)
Mensaje Y / 00:00:00:006 / cellcoupled(0,1)(04) / out / 1.000 para cellcoupled(02)
Mensaje D / 00:00:00:006 / cellcoupled(0,1)(04) / ... para cellcoupled(02)
...
Mensaje Y / 00:00:00:006 / cellcoupled(9,48)(501) / out / 1.000 para cellcoupled(02)
Mensaje D / 00:00:00:006 / cellcoupled(9,48)(501) / ... para cellcoupled(02)
Mensaje X / 00:00:00:006 / cellcoupled(02) / neighborchange / 1.000 para cellcoupled(9,1)(454)
...
Mensaje X / 00:00:00:006 / cellcoupled(02) / neighborchange / 1.000 para cellcoupled(0,45)(48)
Mensaje D / 00:00:00:006 / cellcoupled(9,1)(454) / 00:00:00:006 para cellcoupled(02)
...
```

```
Mensaje D / 00:00:00:006 / cellcoupled(0,45)(48) / ... para cellcoupled(02)
Mensaje D / 00:00:00:006 / cellcoupled(02) / 00:00:00:004 para top(01)
Mensaje D / 00:00:00:006 / top(01) / 00:00:00:004 para Root(00)
Mensaje * / 00:00:00:010 / Root(00) para top(01)
Mensaje * / 00:00:00:010 / top(01) para cellcoupled(02)
Mensaje * / 00:00:00:010 / cellcoupled(02) para cellcoupled(0,0)(03)
...
Mensaje * / 00:00:00:010 / cellcoupled(02) para cellcoupled(9,44)(497)
Mensaje Y / 00:00:00:010 / cellcoupled(0,0)(03) / out / 0.000 para cellcoupled(02)
Mensaje D / 00:00:00:010 / cellcoupled(0,0)(03) / ... para cellcoupled(02)
...
Mensaje D / 00:00:00:040 / cellcoupled(02) / ... para top(01)
Mensaje D / 00:00:00:040 / top(01) / ... para Root(00)
```

Figura 107. Intercambio de mensajes en el modelo celular de tráfico urbano.

Los resultados obtenidos para el caso de los modelos achatados son los siguientes:

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para cellcoupled(02)
Mensaje D / 00:00:00:000 / cellcoupled(02) / 00:00:00:006 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:006 para Root(00)
Mensaje * / 00:00:00:006 / Root(00) para top(01)
Mensaje * / 00:00:00:006 / top(01) para cellcoupled(02)
Mensaje Y / 00:00:00:006 / cellcoupled(02) / out / 1.000 para top(01)
Mensaje D / 00:00:00:006 / cellcoupled(02) / 00:00:00:004 para top(01)
...
Mensaje D / 00:00:00:042 / top(01) / 00:00:00:006 para Root(00)
Mensaje * / 00:00:00:048 / Root(00) para top(01)
Mensaje * / 00:00:00:048 / top(01) para cellcoupled(02)
Mensaje D / 00:00:00:048 / cellcoupled(02) / ... para top(01)
Mensaje D / 00:00:00:048 / top(01) / ... para Root(00)
Mensaje X / 00:00:10:000 / Root(00) / in / 0.000 para top(01)
Mensaje X / 00:00:10:000 / top(01) / in / 0.000 para cellcoupled(02)
Mensaje D / 00:00:10:000 / cellcoupled(02) / 00:00:01:000 para top(01)
Mensaje D / 00:00:51:000 / top(01) / ... para Root(00)
...
Mensaje X / 00:00:53:000 / Root(00) / in / 1.000 para top(01)
Mensaje X / 00:00:53:000 / top(01) / in / 1.000 para cellcoupled(02)
Mensaje D / 00:00:53:000 / cellcoupled(02) / 00:00:01:000 para top(01)
Mensaje D / 00:00:53:000 / top(01) / 00:00:01:000 para Root(00)
Mensaje * / 00:00:54:000 / Root(00) para top(01)
Mensaje * / 00:00:54:000 / top(01) para cellcoupled(02)
Mensaje D / 00:00:54:000 / cellcoupled(02) / ... para top(01)
Mensaje D / 00:00:54:000 / top(01) / ... para Root(00)
```

Figura 108. Intercambio de mensajes usando el coordinador achatado.

El estado inicial para este ejemplo es el siguiente:

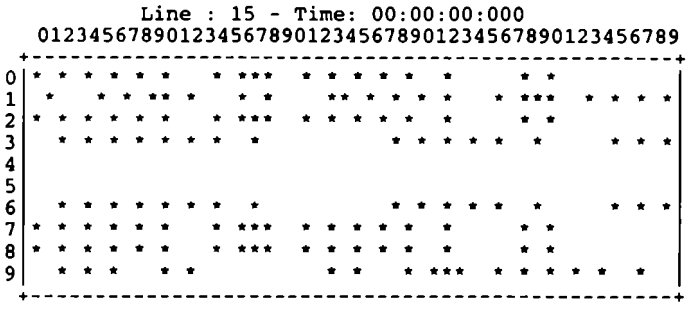


Figura 109. Estado inicial para la simulación de tráfico.

Nuevamente se hicieron estudios comparativos entre las soluciones jerárquicas y achatadas, considerando los peores casos de tiempo de ejecución. En este caso se consideró un espacio de celdas de tamaño fijo con un 60% del espacio de celdas activo, y se varió la duración de la simulación. Los resultados obtenidos fueron los siguientes.

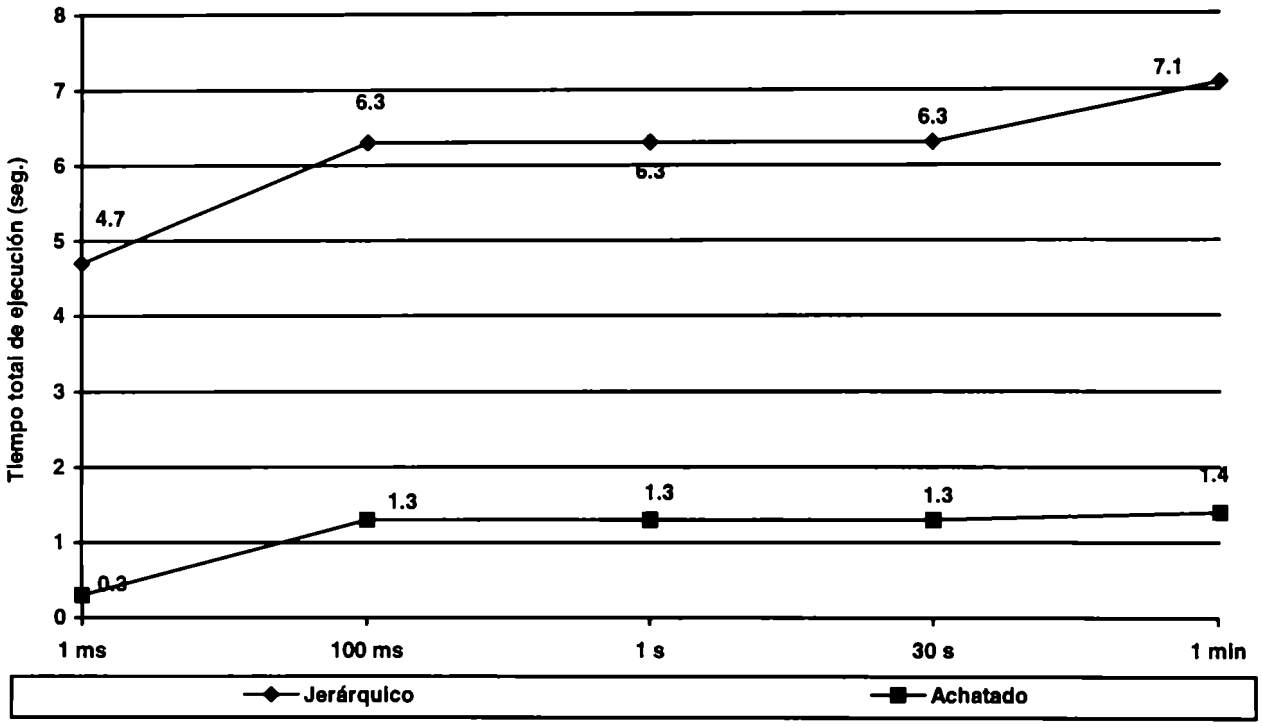


Figura 110. Comparación entre jerárquicos y achatados para la simulación de tráfico.

Finalmente se analizó un caso de simulación de tráfico más complejo, en el cual no se conocen los estados iniciales en algunas de las celdas, hay zonas con distinto comportamiento que en el resto de las celdas, así como varias celdas de entrada y salida. La especificación de este modelo es el siguiente:

```
[top]
components : CellCoupled
out : out
in : in
link : in in@CellCoupled
link : out@cellcoupled out

[CellCoupled]
type : flat
width : 50
height : 10
delay : transport
defaultDelayTime : 1000
border : wrapped
neighbors :                CellCoupled(-1,0)
neighbors : CellCoupled(0,-1) CellCoupled(0,1)
neighbors :                CellCoupled(1,0)
in : in
out : out
link : in in@CellCoupled(0,0)
link : in in@CellCoupled(7,1)
link : in in@CellCoupled(6,3)
link : out@cellcoupled(9,0) out
link : out@cellcoupled(9,1) out
link : out@cellcoupled(9,2) out
initialvalue : 1
initialrowvalue : 0 10101010101000101110010101010100?????101000000000
initialrowvalue : 1 01000101011017?0101??001101010101000101110010101010
initialrowvalue : 2 10101010101000101110010101010100?????101000000000
initialrowvalue : 3 00101010101010100?000000000101010101001?????10101
initialrowvalue : 4 00101010??10100000000000101000101110010101010100??
initialrowvalue : 5 10101010101010100?000000000101010101001?????10101
initialrowvalue : 6 00101010101010100?000000000101010101001?????10101
initialrowvalue : 7 10101010101000101110010101010100?????101000000000
```

```
initialrowvalue : 8  10101010101000101110010101010100?????101000000000
initialrowvalue : 9  00101010??101000000000001010001011100101010101001??
localtransition : transito
zone : bache { (8,8)..(9,9) (1,2)..(4,4) }


[transito]
rule : 1 5 { (0,0) = 0 and (1, 0) = 1  }
rule : 1 5 { ( 0, 0 ) = 0 and (0, -1) = 1  }
rule : 1 5 { (0,0) = 1 and (0,1) = 1  }
rule : 1 5 { ( 0,0) = 1 and (1,0) = 1  }
rule : 0 10 { (0,0) = 1 and (-1, 0) = 0 }
rule : 0 10 { (0,1) = 0 and (0,0) = 0 }
rule : 0 10 { ( 0,0) = 1 and (1,0) = 1 }
rule : 1 5 { t }

[bache]
rule : 0 1 { t }
```

Figura 111. Definición de un modelo celular de tráfico urbano usando la herramienta.

La función de transición local evalúa si una posición esta siendo transitada por un vehículo. Usando el lenguaje de especificación, esto equivale a comparar el valor de la celda con el valor uno (1). En caso de tener la vía libre, el vehículo puede avanzar dejando la celda libre, y por ende su valor resultado es cero (0). Cada línea de especificación de la función de transición local determina bajo qué circunstancias debe retornarse cero o uno. Estas condiciones deben ser disjuntas. Esta restricción es verificada o notificada al leer la especificación del modelo celular.

Aquí hay tres puntos de entrada (celdas (0,0) (7,1) y (6,3)) y dos de salida (celdas (9,0) y (9,1)). Además se especificó distinto comportamiento en distintos lugares del espacio de celdas, introduciendo distintos comportamientos para un bache, y distintas demoras en las reglas de tránsito. Las diferentes demoras permiten diferenciar (medidas en milisegundos) las reglas en las que un automóvil abandona una celda de aquellas en las que el automóvil ingresa en la siguiente celda.

En este caso no se mostrarán los resultados de la ejecución de la herramienta. Se realizaron estudios comparativos similares al caso anterior, cuyos resultados son los siguientes:

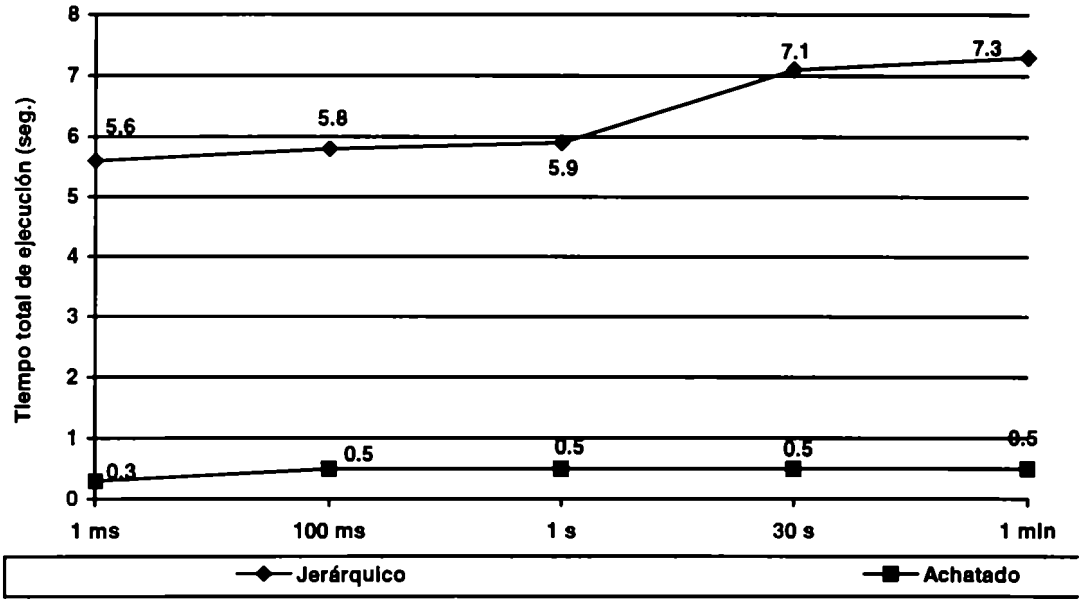


Figura 112. Comparación entre jerárquicos y achatados para la simulación de tráfico.

Un objetivo primordial del nuevo formalismo, así como el de la herramienta desarrollada, reside en la reducción en los tiempos de desarrollo y mantenimiento de la simulaciones. A pesar de que las experiencias desarrolladas son sólo prototipos simples que sirvieron para chequear el uso de la herramienta y su adecuación al formalismo, pudieron obtenerse resultados altamente favorables con respecto al no uso del formalismo. Se registraron datos relacionados con

los tiempos de desarrollo de las distintas soluciones, clasificándolas entre distintos tipos de usuarios. Los resultados obtenidos se resumen en las siguientes tablas:

Usuarios Novatos

Tiempo de aprendizaje de uso de la herramienta: 16 h/h				
	Juego de la Vida		Simulación de tráfico	
	Desarrollo	Chequeo	Desarrollo	Chequeo
Especificación de los primeros problemas	0.75 h/h	2.50 h/h	2.50 h/h	4.3 h/h
Especificación de los últimos problemas	0.25 h/h	0.75 h/h	1.00 h/h	1.7 h/h
Mantenimiento	0.25 h/h	0.30 h/h	0.25 h/h	0.3 h/h

Usuarios Expertos

	Juego de la Vida		Simulación de tráfico	
	Desarrollo	Chequeo	Desarrollo	Chequeo
Especificación de los primeros problemas	0.50 h/h	1.30 h/h	0.50 h/h	0.75 h/h
Especificación de los últimos problemas	0.10 h/h	0.30 h/h	0.30 h/h	0.75 h/h
Mantenimiento	0.25 h/h	0.30 h/h	0.25 h/h	0.50 h/h

Especificación, diseño y programación directa (autómatas celulares asincrónicos).

	Juego de la Vida		Simulación de tráfico	
	Desarrollo	Chequeo	Desarrollo	Chequeo
Especificación de los primeros problemas	5 h/h	9 h/h	8 h/h	11 h/h
Especificación de los últimos problemas	3 h/h	5 h/h	2 h/h	3 h/h
Mantenimiento	4 h/h	12 h/h	3 h/h	7 h/h

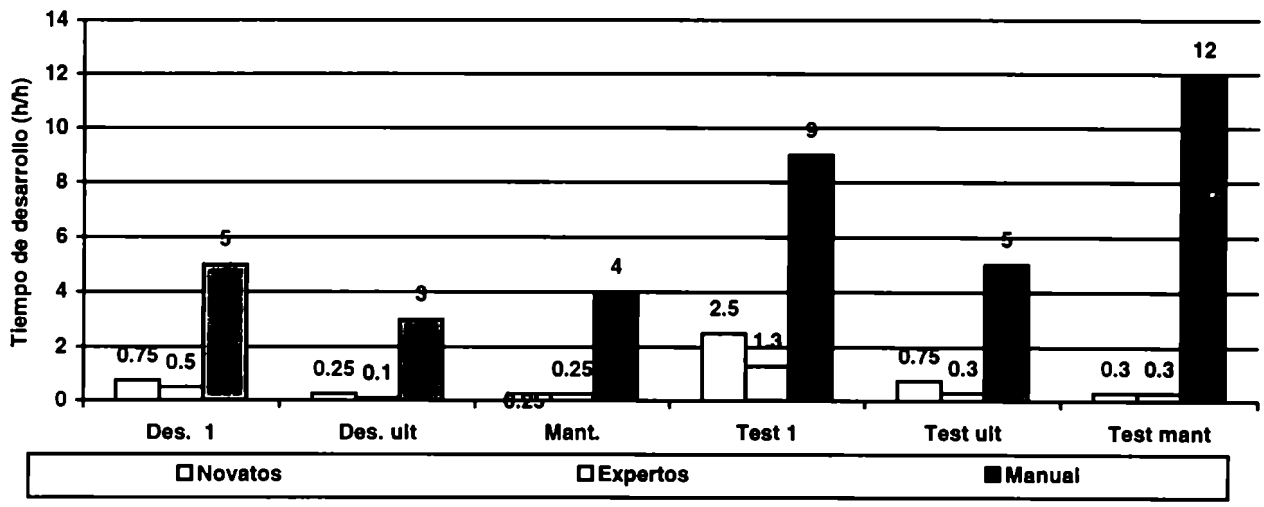


Figura 113. Comparación de tiempos de desarrollo para el Juego de la Vida.

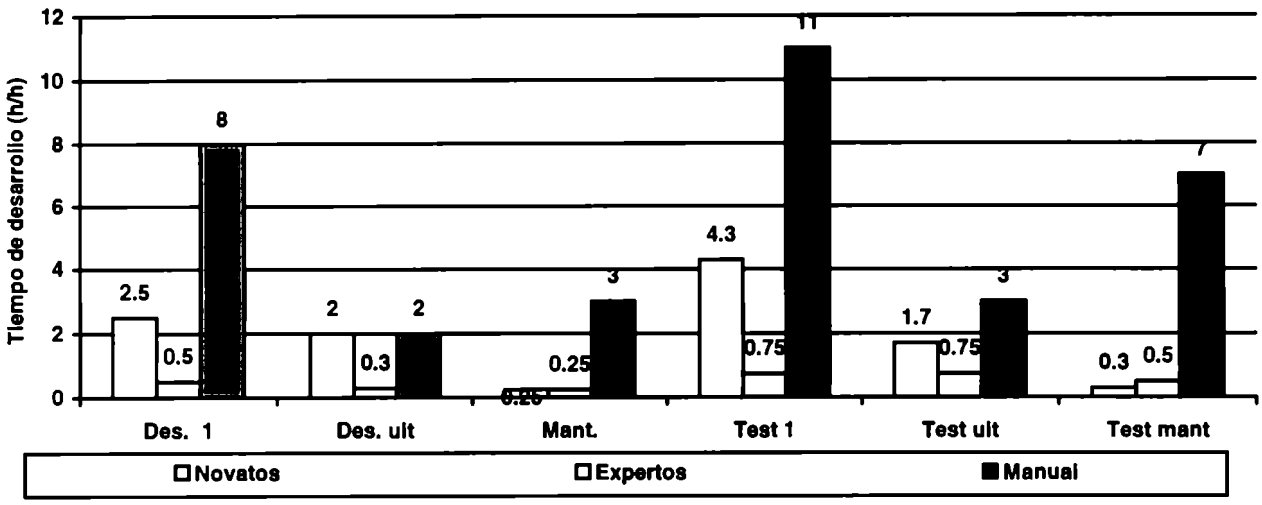


Figura 114. Comparación de tiempos de desarrollo para simulación de tráfico.

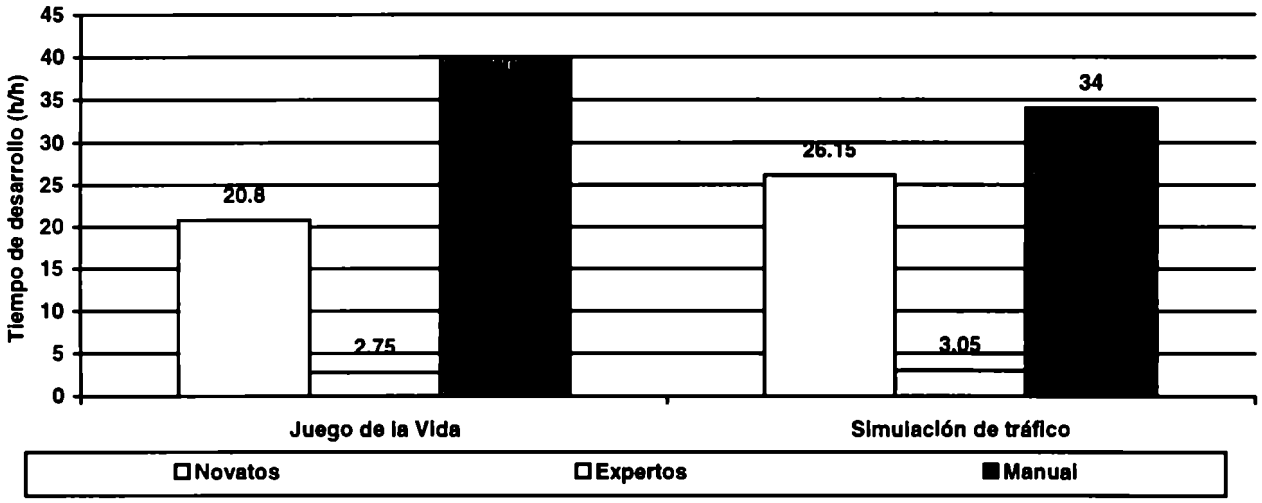


Figura 115. Comparación de tiempos totales de desarrollo para ambas aplicaciones.

Los modelos usados para hacer los estudios comparativos fueron son simples, pero permitieron obtener cifras comparativas de los tiempos de desarrollo y ejecución de modelos de ejemplo.

8. CONCLUSION. TRABAJO FUTURO.

En este trabajo se introdujeron nuevas descripciones formales para la modelización y simulación de espacios de celdas. Los formalismos se basan en los paradigmas de Autómatas Celulares asincrónicos y DEVS. Además permiten la utilización de demoras de transporte o inerciales.

Se presentó una descripción formal de modelos Cell-DEVS de n-dimensiones con distintos tipos de estados, pero se puso especial énfasis en modelos binarios y de tres estados. Se construyó una herramienta que permite implementar estos formalismos.

Los métodos presentados permiten definición automática de espacios de celdas usando un lenguaje de especificación de alto nivel. Se pudieron observar mejoras en los rendimientos de tiempos de desarrollo en hasta 10 veces. Las principales ventajas se obtuvieron en las fases de chequeo y mantenimiento, ya que la utilización de un método formal permite validación automática de los modelos especificados, permitiendo que el desarrollador se concentre en los métodos asociados con el modelo a simular. Por otro lado, la herramienta traduce automáticamente la especificación en un modelo simulable que puede ejecutarse.

Se presentó un mecanismo de achatamiento de los modelos jerárquicos, de tal forma de poder ejecutarlos con alto grado de eficiencia. A su vez se presentó un enfoque para extender las soluciones actuales a entornos paralelos y distribuidos. Esta última extensión se propuso debido a que en [MZBG97], se obtuvieron resultados con aceleraciones de hasta 1000 veces para espacios de celdas, basados en el formalismo DEVS ejecutado en paralelo, debido a la combinación del enfoques asincrónicos junto con técnicas de simulación paralela asincrónica.

Como los métodos presentados se basan en el formalismo DEVS, se permite la integración de múltiples vistas para cada submodelo, permitiendo integrar distintos paradigmas de modelización de forma eficiente.

En la actualidad se están realizando diversos trabajos en relación con el presente, que serán resumidos a continuación.

- Implementación de una versión paralela de la herramienta. Con este fin se ha obtenido un entorno de simulación Time-Warp, que está siendo modificado para incuir mecanismos pesimistas. El objetivo final será realizar estudios comparativos de los distintos algoritmos existentes con respecto a su eficiencia al ejecutar modelos Cell-DEVS. Con tal fin se analizarán algoritmos de distribución automática de la carga. A su vez se realizarán estudios comparativos del Modelo de Frecuencia de Rollbacks presentado en la sección 4.4.3. Actualmente se están implementando estas soluciones, y se ha hecho un trabajo exhaustivo de survey de soluciones de programación paralela y distribuida asincrónica de eventos discretos [ASW98] (cuyos contenidos se resumieron en la sección 2.4).

- Definición de un lenguaje de especificación para analizar tráfico en centros urbanos. A su vez se definirá un mecanismo de traducción de este lenguaje en modelos Cell-DEVS, para permitir su simulación. En este sentido se realizó un estudio exhaustivo de aplicaciones de autómatas celulares existentes [DW98a], y se especializó en soluciones utilizadas para simulación de tráfico. Se propuso una taxonomía de estas soluciones, analizando fallas en los modelos existentes [DW98b]. Esta taxonomía permitió identificar problemas en los modelos existentes, permitiendo la propuesta de nuevos esquemas basados en autómatas celulares. El objetivo final es que un usuario pueda especificar el comportamiento de tráfico en una sección urbana, y proceder a su simulación en base a las herramientas Cell-DEVS desarrolladas en el presente trabajo.

- Definición de un lenguaje de especificación basado en grafos para aplicaciones industriales. La idea básica será poder especificar cada modelo atómico DEVS usando un lenguaje basado en diagramas de transición de estados, de forma tal que un usuario de la herramienta no deba programar los modelos, sino especificarlos para proceder a su simulación. En base a las soluciones obtenidas se puede proceder a la síntesis de los controladores asociados.

- Extensión del formalismo Cell-DEVS considerando el formalismo R-DEVS [ChZ94], que ha sido definido para permitir ejecución paralela de los modelos. En este caso se propone una forma alternativa de especificar las prioridades para tratar eventos simultáneos. Aquí se tratará de clasificar las celdas inminentes en el espacio de celdas, usando la función de transición confluente δ_{con}.

- Extensiones a la herramienta: se proveerá una interfaz gráfica, se considerarán extensiones a tres dimensiones, así como a celdas hexagonales. El lenguaje de especificación se modificará para permitir estados en los enteros y los reales, permitiendo de esta forma modelizar sistemas complejos en otros dominios.

9. REFERENCIAS

NOTA: En numerosos casos, las referencias bibliográficas fueron obtenidas por medios electrónicos, por lo que no se cuenta con datos de volúmen y páginas de las mismas.

[AAD97] AGELL, N.; AGUADO, M.; DOMENECH, M.; SANCHEZ, M.; PIERA, N. "Mathematical foundations of qualitative reasoning". *Internal Report. Universitat Politécnica de Catalunya.* 1997.

[ASW98] ALVARIÑO, A.; SAEZ, A.; WAINER, G. "Un informe sobre técnicas de simulación paralela y distribuida". *Informe técnico No. 98-013* del Departamento de Computación. FCEN/UBA. 1998.

[BBW98a] BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "GAD: a General Application DEVS environment". *Proceedings of IASTED Applied Modelling and Simulation 1998.* Hawaii, U.S.A. *Technical Report No. 98-005.* Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

[BBW98a] BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "CD++: una herramienta de implementación de modelos Cell-DEVS binarios". *Anales de la XXII Conferencia Latinoamericana de Informática.* Quito, Ecuador. 1998.

[BBW98c] BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "Implementation of a Cell-DEVS modeling environment". Technical Report No. 98-006. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

[BEA97] BLUE, V.; EMBRECHTS, M.; ADLER, J. "Cellular Autómata Modeling of Pedestrian Movements". *Proceedings of IEEE Systems, Man and Cybernetics 1997.* 1997.

[ChM81] CHANDY, K; MISRA, J. "Asynchronous distributed simulation via a sequence of parallel computations". *Communications of the ACM*, (24):4, pp 198-206. April 1981.

[ChM79] CHANDY, K; MISRA, J. "Distributed simulation: a case study in design and verification of distributed programs". *IEEE Transactions on Software Engineering.* September 1979.

[Cho95] CHO, Y. "Parallel Implementation of container using parallel virtual machine". *M.Sc. Thesis. Department of Electrical and Computer Engineering.The University of Arizona.* 1995.

[ChZ94] CHOW, A.; ZEIGLER, B. "Revised DEVS: a parallel, hierarchical, modular modeling formalism". *Proceedings of the SCS Winter Simulation Conference.* 1994.

[CCB97] COCKSHOTT, P.; MC. CASKILL, G.; BARRIC, P. "Use of a high speed cellular autómata machine to simulate road traffic". Technical Report. 1997.

[EDPWJ89] EBLING, M.; DI LORETO, J.; PRESLEY, B.; WIELAND, J. JEFFERSON, D. "An ant foraging model implemented on the time warp operating system". *Distributed Simulation 1989.* pp. 21-30. 1989.

[Fuj89b] FUJIMOTO, R. "Time Warp on a shared memory multiprocessor". *Transactions of the SCS.* 6, 3. pp. 211-239. July 1989.

[Fuj90] FUJIMOTO, R. "Parallel Simulation of Discrete Events". *Communications of the ACM.* Vol. 33. No. 10. pp. 30-53. 1990.

[FZ92] FISHWICK, P.; ZEIGLER, B. "A multimodel methodology for qualitative model engineering". *ACM TOMACS* 2(1): pp. 52-81, 1992.

[Gaf88] GAFNI, A. "Rollback mechanisms for optimistic distributed simulation systems". *Proceedings of the SCS Multiconference on Distributed Simulation.* 19(3). pp. 61-67. July 1988.

[Gar70] GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'.". *Scientific American.* 23 (4). pp. 120-123. April 1970.

[GFE95] GIAMBIASI, N.; FRYDMAN, C.; ESCUDE, B. "Hierarchical/Multi-view modeling and simulation". *Proceedings of the 7th. SCS European Conference on Computer Simulation.* 1995.

[GG96] GHOSH, S.; GIAMBIASI, N. "On the need for consistency between the VHDL language constructions and the underlying hardware design". *Proceedings of the 8th. European Simulation Simposium.* Genoa, Italy. Vol. I. pp. 562-567. 1996.

[Gia96] GIAMBIASI, N. "Introduction à la modélisation et à la simulation". *Materiales del curso de D.E.A.; Université d'Aix-Marseille III.* 1996.

[GLNW94] GREENBERG, A.; LUBACHEVSKY, B.; NICOL, D.; WRIGHT, P. "Efficient Massively Parallel Simulation of Dynamic Channel Assignment Schemes for Wireless Cellular Comunications". *Proceedings of the 8th. Workshop on Parallel and Distributed Simulation.* pp. 187-194, 1994.

[GM76] GIAMBIASI, N.; MIARA, A. "SILOG: A practical tool for digital logic circuit simulation. *Proceedings of the 16th. D.A.C.*, San Diego, U.S.A. 1976.

[Gut95] GUTOWITZ, H. "Cellular Autómata and the sciences of Complexity. Part I-II". *Complexity.* November 1995.

[Ho89] HO, Y. "Special issue on discrete event dynamic systems". *Proceedings of the IEEE*, 77 (1), 1989.

[Jef85] JEFFERSON, D. "Virtual Time". *ACM Transactions on Programming Languages and Systems.* 7(3). pp. 404-425. July 1985.

[LM90] LIPTON, R.; MIZELL, D. "Time-Warp vs. Chandy-Misra: a worst-case comparison". *Proceedings of the SCS Multiconference on Distributed Simulation.* 22, . pp. 137-143. 1990.

[Lub89] LUBACHEVSKY, B. "Efficient distributed event-driven simulations of multiple-loop networks". *Communications of the ACM.* 32. pp. 111-123. January 1989.

[LWS91] LUBACHEVSKY, B.; WEISS, A.; SCHWARZ, A. "An analysis of rollback-based simulation". ACM TOMACS, 1(2):154-193, April 1991.

[MWM88] MADISETTI, V.; WALRAND, J; MASSERSCHMIT, D. "Wolf: a rollback algorithm for optimistic distributed simulation systems". *Proceedings of Winter Simulation Conference.* pp. 296-305. 1988.

[MZBG97] MOON, Y.; ZEIGLER, B.; BALL, G.; GUERTIN, D.P. "DEVS representaion of spatially distributed systems: validity, complexity reduction". *IEEE Transactions on Systems, Man and Cybernetics.* pp. 288-296. 1996.

[OS93] OVEREINDER, B.; SLOOT, P. "Application of Time-Warp to parallel simulations with asynchronous cellular autómata". In *European Simulation Symposium.* Delft, Netherlands. October 1993.

[OSH92] OVEREINDER, B.; SLOOT, P.; HERZBERGER, L. "Time-Warp on a transputer platform: pilot study with asynchronous cellular autómata". In *Parallel computing and transputer applications.* Amsterdam, Netherlands. Washington: IOS Press. September 1992.

[Rey82] REYNOLDS, P. Jr. "A shared resource algorithm for distributed simulation". *Proceedings of the 9th. Annual Symposium on Computer Architectures.* pp. 259-266. 1982.

[RW89] RIGHTER, R.; WALRAND, J. "Distributed simulation of discrete event systems". *Proceedings of the IEEE.* pp. 99-119. January 1989.

[SBW88] SOKOL, L.; BRISCOE, D; WIELAND, A. "MTW: a strategy for scheduling discrete simulation events for concurrent execution", *Proc. of SCS conf on distributed simulation*, pp. 34-42. 1988.

[SG96] SIMON, P. , GUTOWITZ, H. "A Cellular automaton model for Bi-directional traffic". Technical Report. 1996.

[Tad96] TADAKI, T. "Two - dimensional cellular automaton model of traffic flow with open boundaries". Technical Report. 1996.

[Tof94] TOFFOLI, T. "Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular autómata)". *Proceedings of ACRI'94*. 1994.

[Wai96] WAINER, G. "Introducción a la Simulación de Eventos Discretos". *Technical Report n.: 96-005. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires*. 1996.

[WFG97a] WAINER,G.; FRYDMAN, C.; GIAMBIASI, N. "An environment to simulate cellular DEVS models". *Proceedings of the SCS European Multiconference on Simulation*. Istanbul, Turkey. 1997.

[WFG97b] WAINER,G.; GIAMBIASI, N.; FRYDMAN, C. "Cell-DEVS models with transport and inertial delays". *Proceedings of the 9th. European Simulation Symposium and Exhibition*. Passau, Germany. 1997.

[WG97] WAINER, G.; GIAMBIASI, N. "Specification, modeling and simulation of timed Cell-DEVS spaces". *Technical Report n.: 98-007. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Submitted to ACM Transactions on Modelling and Computer Simulation*. 1998.

[Wol84] WOLFRAM, S. "Universality and complexity in cellular autómata". *Physica*, 10D, pp. 1-35. 1984.

[Wol86] WOLFRAM, S. "Theory and applications of cellular autómata". Vol. 1, *Advances Series on Complex Systems*. World Scientific, Singapore, 1986.

[Wul69] WULF, W. "Performance monitors for multiprogramming systems". *Proceedings of the 2nd. ACM symposium on Operating Systems Principles*. 1969. pp. 175-181.

[Zei76] ZEIGLER, B. "Theory of modeling and simulation". *Wiley*, 1976.

[Zei84] ZEIGLER, B. "Multifaceted Modelling and discrete event simulation". *Academic Press*, 1984.

[Zei90] ZEIGLER, B. "Object-oriented simulation with hierarchical modular models". *Academic Press*, 1990.

[Zei95] ZEIGLER, B. "Object-oriented simulation with hierarchical modular models. Revised to include source code for DEVS-C++." *Department of Electrical and Computer Engineering. University of Arizona*. 1995.

[Zei97] ZEIGLER, B. "Objects and systems: principled design with Java/C++ Implementation". *Springer-Verlag*. New-York, 1997.

10. BIBLIOGRAFIA ADICIONAL

[ACFS92] AKYLDIZ, L.; CHEN, S.R.; DAS, R.; FUJIMOTO, R.; SERFOZO, R. "Performance analysis of time warp with limited memory". *Proceedings of the 1992 ACM SIGMOD.* pp. 213-224. May 1992.

[AR92] AYANI, R.; RAJAEI, H. "Parallel simulation using conservative time windows". *Proceedings of the Winter Simulation Conference.* pp. 709-717. December 1992.

[AS92] ARVIND, D.; SMART, C. "Hierarchical PDES in composite ELSA". *Proceedings of the 6th. Workshop on PDES.* pp 147-156. 1992.

[BD89] BELLENOT, S., DI LORETO, M. "Tools for measuring the performance and diagnosing the behavior of distributed simulations using time warp". *Proceedings of the SCS Multiconference on Distributed Simulation.* 1989.

[Bel90] BELLENOT, S. "Global virtual time algorithms". *Proceedings of the SCS Multiconference on Distributed Simulation.* pp. 122-127. 1990.

[Bel91] BELLENOT, S. "State skipping performance with the time warp OS". *Proceedings of the SCS Multiconference on Distributed Simulation.* 24(3): pp. 53- 58. 1991.

[BH90] BALL, D.; HOYT, S. "The adaptive Time-warp concurrency control algorithm". *Proceedings of the SCS Multiconference on Distributed Simulation.* pp. 174-177. 1990.

[BL90] BAGRODIA, R.; LIAO, W. Maisie: a language and optimizing environment for distributed simulation". *Proceedings of the SCS PDES '90..* pp. 205-210. 1990.

[BLMW96] BASSO, G.; LOTTRUP, L.; MIGNEMI, A.; WAINER, G. "Simulación de sistemas de eventos discretos con autómatas celulares". . Publicación on-line en el Departamento de Computación, FCEN-UBA. 1996.

[BLU90] BAAZNER, D; LOMOW, G. ;UNGER, W. "Sim++: the transition to Distributed Simulation". *Proceedings of the SCS PDES '90.* pp. 211-218. 1990.

[Cho94b] CHOW, A.; ZEIGLER, B. "Abstract Simulador for the parallel DEVS formalism". *Technical Report. University of Arizona.* 1994.

[Cho95b] CHOW, A. "A C++ binding of the parallel DEVS formalism". *Proceedings of the SCS Multiconference on PDES '95.* pp. 38. 1995.

[ChS89] CHANDY, K; SHERMAN, R. "Space, time and simulation". *Proceedings of the SCS Multiconference on Distributed Simulation.* March 1989.

[Con89] CONCEPCION, A. "A hierarchical computer architecture for distributed simulation". *IEEE Transactions on Computers.* February 1989.

[Con91] CONCEPCION, A. et al. "Computing global virtual time using the multi-level token passing algorithm". *Proceedings of the SCS Multiconference on Distributed Simulation.* 1991.

[CWM95] CAMERON, G.; WYLIE, B.; MC. ARTHUR, D. "PARAMICS: Moving Vehicles on the Connection Machine". *Proceedings of IEEE Supercomputing '95.* 1995.

[CZ88] CONCEPCION, A.; ZEIGLER, B. "DEVS Formalism: a framework for hierarchical model development". *IEEE Transactions on Software Engineering.* pp. 228-241. Vol. 14, No. 2. February 1988.

[DN66] DAHL, O.; NYGAARD, K. "Simula: an algol-based simulation language". *Communications of the ACM.* 9, pp. 671-688. 1966.

[DR90] DICKENS, D.; REYNOLDS, P. "SRADS with local rollback". *Proceedings of the SCS Multiconference on Distributed Simulation.* 22(1). pp. 161-164. 1990.

[Eck95] ECKART, J.D. "A Cellular autómata simulation system". *Technical Report Radford University*. 1995.

[Esc95] ESCUDE, B. "Conception et realisation d'un simulateur hierarchise, modulaire et a evenements discrets". *Memoire du D.E.A d'Informatique et d'automatique*. Universite de Droit, d'Economie et des Sciences d'Aix-Marseille III. Institut Universitaire des Sciences pour l'Ingenieur de Marseille. 1995.

[Esc95b] ESCUDE, B. "Simulateur 1.0v. Manuel du Utilizateur". *Universite de Droit, d'Economie et des Sciences d'Aix-Marseille III. Institut Universitaire des Sciences pour l'Ingenieur de Marseille*. 1995.

[Esc95c] ESCUDE, B. "Conception et realisation d'un simulateur hierarchise, modulaire et a evenements discrets.". *Memoire du D.E.A d'Informatique et d'automatique*. Universite de Droit, d'Economie et des Sciences d'Aix-Marseille III. Institut Universitaire des sciences pour l'Ingenieur de Marseille. 1995.

[Fer95b] FERSCHA, A. "Probabilistic adaptive direct optimism control in Time Warp". *Proceedings of the 9th. Workshop on Parallel and Distributed Simulation*. 1995.

[FK92] FELDERMAN, R.; KLEINROCK, L. "Two processor conservative simulation analysis". *Proceedings of the SCS Western Multiconference on advances in PDES*. 24(3). pp. 169-177. 1992.

[Fox90] FOX, G. "Solving problems on concurrent processors". Vol. 1. *Prentice-Hall, Englewood Cliffs, New Jersey*. 1990.

[FM89] FRIEDEMANN, J.; MATTERN, P. "Virtual time and global states of distributed systems". *Parallel and Distributed Algorithms*. pp. 215-226. 1989.

[FT95] FERSCHA, A.; TRIPATHI, S. "Parallel and distributed simulation of discrete event systems". *Technical Report CS-TR-3336, Dept. of Computer Science, University of maryland, College Park, MD. Proceedings of the 9th. Workshop on Parallel and Distributed Systems*. 1995.

[Fuj89] FUJIMOTO, R. "Performance measurements of distributed simulation strategies". *Transactions of the SCS*. 6, 2. pp. 21-26. March 1989.

[Fuj89c] FUJIMOTO, R. "The virtual time machine". *Proceedings of the 1989 ACM symposium on parallel algorithms and architectures*. pp. 199-208. June 1989.

[Fuj90b] FUJIMOTO, R. "Optimistic approaches to PDES". *Transactions of the SCS*. Vol 7. No. 2, pp. 153-191. 1990.

[GE97] GIAMBIASI, N.; ESCUDE, B. "Discrete Events Specification of hybrid coupled models". *Proceedings of the 9th. European Simulation Symposium and Exhibition*. Passau, Germany. 1997.

[GGZ86] GARZIA, R.F.; GARZIA, M.R.; ZEIGLER, B.P. "Discrete Event Simulation". *IEEE Spectrum*. pp. 32-36. December 1986.

[Gho93] GHOSH, K.; PANESAR, K.; FUJIMOTO, R.; SCHWAN, K. "PORTS: A parallel, optimistic, real-time Simulador". *Technical report GIT-CC-93/71. College of Compute, Georgia Institue of Technology*. December 1993.

[Gup95] GUPTA, A; AKYLDIZ, I.; FUJIMOTO, R. "Performance analysis of time warp with multiple homogeneous processors". *IEEE Transactions on Software Engineering*. 17, No. 10, pp. 1010-1027. 1995.

[GV95] GAVRILOV, D.; VINOGRADOV, O. "Object-Oriented Programming and Object-Oriented Simulation". *Proceedings of the SCSC '95*. pp. 143-152. 1995

[Jef85b] JEFFERSON, D. et al "Implementation of time warp on the Caltech hypercube". *Proceedings of the SCS Multiconference on PDES*. pp. 70-75. January 1985.

[Jef87] JEFFERSON, D. "Distributed simulation and the Time Warp Operating System". *11th. Symposium on Operating Systems Principles*. pp 77-93. November 1987.

[Jef90] JEFFERSON, D. "Virtual Time II. Storage management in distributed simulation". *Proceedings of the 9th. annual ACM Symposium on Principles of Distributed Computing*. pp. 75-90. 1990.

[JS85c] JEFFERSON, D.; SOWIZRAL, H. "Fast concurrent simulation using the time warp mechanism". *Proceedings of the SCS Distributed Simulation Conference*. pp 63-69. 1985.

[Kim94] KIM. T.; SEONG, Y.; PARK. K. "Mapping modular, hierarchical discrete event models in a hypercube computer". *Proceedings of International Conference on Massively Parallel Processors Application and Development*. 1994.

[Kim95] KIM, K. et al. "Distributed optimistic simulation of hierchical DEVS models". *Proceedings of the SCS multiconference on distributed simulation '95*. pp. 32-37. 1995.

[LF95] LIN, Y.; FISHWICK, P. "Asynchronous Parallel Discrete Event Simulation". *IEEE Transactions on Systems, Man and Cybernetics*. 1995.

[Lin92] LIN, Y. "Memory management algorithms for optimistic parallel simulation". *Proceedings of the 6th. workshop on PDES* pp. 43-52 y *Proceedings of Western Multiconference on Advances in Distributed Simulation*. 23(1). pp. 43-52. 1992.

[Lin92b] LIN, Y. "Parallelism analyzers for PDES". *ACM Transactions on Modelling and Computer Simulation*. 2(3). pp. 239-264. 1992.

[LL90] LIN, Y.; LAZOWSKA, E. "Optimality considerations of time-warp parallel simulation". *Proceedings of the SCS multiconference on distributed simulation*. 22(1). pp. 29-34. 1990.

[LL90b] LIN, Y.; LAZOWSKA, E. "Determining global virtual time in a distributed simulation". *Proceedings of the International conference on Parallel Processing*. III: pp. 201-209. 1990.

[LL90c] LIN, Y.; LAZOWSKA, E. "Reducing the state saving overhead for the time warp parallel simulation". *Technical Report 90-02-03. Department of Computer Science. University of Washington, Seattle*. February, 1990.

[LL91] LIN, Y.; LAZOWSKA, E. "A time-division algorithm for parallel simulation". *ACM Transactions on Modelling and Computer Simulation*. 1(1): pp. 73-83. January 1991.

[LL91b] LIN, Y.; LAZOWSKA, E. "A study of time warp rollback mechanisms". *ACM Transactions on Modelling and Computer Simulation*. 1(1): pp. 51-72. January 1991.

[LP91] LIN, Y; PREISS, B. "Optimal memory management for time warp simulation". *ACM Transactions on Modelling and Computer Simulation*. October 1991.

[LT90] LIU, L.; TROPPER, C. "Local deadlock detection in distributed simulations". *Proceedings of the SCS multiconference on Distributed simulation*. 22(1):64-69, January 1990.

[Lub88] LUBACHEVSKY, B. "Bounded lag distributed event simulation". *Proceedings of the SCS multiconference on Distributed simulation*. 1988.

[Lub88b] LUBACHEVSKY, B. "Simulating colliding rigid disks in parallel using bounded lag without time warp". *Proceedings of the SCS multiconference on Distributed simulation*. 1989.

[Meh91] MEHL, H. "Speedup of conservative distributed discrete event simulation method by speculative computing", *Proceedings of SCS multiconference on advances in PDES*. pp. 163-166. 1991.

[Meh92] MEHL, H. "A deterministic de-breaking scheme for sequential and distributed simulation". *6th. Workshop on PDES*, 24. pp. 199-200. January 1992.

[Mis86] MISRA, J. "Distributed discrete-event simulations". *ACM Computing surveys*. Vol. 18, No. 1, pp. 39-65. 1986.

[Nic93] NICOL, D. "The cost of conservative syncronization in PDES". *ICASE Technical report 90-20. Journal of the ACM*. 1993.

[Nic94] NICOL, D.; FUJIMOTO, R. "Parallel Simulation Today". *Annals of Operations research*, vol. 53, pp. 249-285. November 1994.

[Ove89] OVERGAARD, G. "A model of the time warp mechanism for implementation on a multitprocessor with shared memory". *Tech. rep. TRITA-TCS-8902, Dept. of Telecommunication and CS*. 1989.

[PJCH95c] PRAEHOFER, H.; JAHN, G.; JACAK, W.; HAIDER, G. "Supervising manufacturing system operation by DEVS-Based intelligent control". *Proceedigns of IEEE AIS '95*. 1995.

[PR94] PRAEHOFER, H.; REISINGER, G. "Distributed Simulation of DEVS-based multiformalism models". *Proceedings of AI Simulation and Planning in High-Autonomy systems*, pp. 150-156. 1994.

[PR95] PRAEHOFER, H.; REISINGER, G. "Object-Oriented realization of a Parallel Discrete Event Simulator". *Technical Report Johannes Kepler University, Department of System Theory and Information Engineering*. 1995.

[Pra95b] PRAEHOFER, H. "Multifacetted, Object Oriented modeling in the transportation domain". *Proceedings of EUROCAST '95, LNCS, Springer-Verlag*. 1995.

[Pre90] PREISS, B. "Performance of DES on a multitprocessor using optimistic and conservative syncrhonization". *Proceedings of the International Conference on Parallel Processing*. III: pp. 218-222. 1990.

[PZ95d] PRAEHOFER, H.; ZEIGLER, B. "Automatic abstraction of event-based control models from continuous base models". *IEEE Transactions of systems man and cybernetics*. 1995.

[RJ89] REIHER, P.; JEFFERSON, D. "Limitation of optimism in the time warp OS", *Proceedings of 1989 Winter Simulation Conference*, pp. 765-769. 1989.

[RMM90] REED, D.; MALONY, A.; MC.CREDIE, D. "PDES using shared memory". *IEEE Transactions on Software Engineering*. 14, no. 4. pp. 541-553. 1990.

[Sun94] SUNDERAM, V. et al. "The PVM Concurrent Computing System: evolution, experience and trends". *Parallel Computing*, Vol. 20, No. 4. pp. 531-545. 1994.

[Wag95] WAGNER, P. "Traffic simulations using cellular autómata: comparison with reality". *Proceedings of Traffic and Granular Flow*, 1995.

[Zei89] ZEIGLER, B. "DEVS representation of dynamical systems: event-based intelligent control". *Proceedings of IEEE*, Vol. 77 No. 1, pp. 27-80. 1989.

[Zei95a] ZEIGLER, B.; KIM, D. "Design of high level modelling / high performance simulation environments". Technical Report, Department of Electrical and Computer Engineering, University of Arizona. 1995.

[Zei95b] ZEIGLER, B.; KIM, D. "Extending the DEVS-Scheme knowledge-based simulation environment for real-time event-based control". *Technical Report, Department of Electrical and Computer Engineering, University of Arizona*. 1995.

[Zei96] ZEIGLER, B.; MOON, Y.; KIM, D.; KIM, D. "DEVS-C++: A high performance modelling and simulation environment". *Technical Report, Department of Electrical and Computer Engineering, University of Arizona. In Proceedings of 29t. Hawaii International Conference on System Sciences*, January 1996.

[ZZ80] ZEIGLER, B; ZHANG, G. "Mapping hierarchical discrete event models to multiprocessor systems: concepts, algorithms and simulation". *Parallel and distributed computing*; 10:pp. 271-281. 7, 1990.

APENDICE - Lenguaje de especificación de comportamiento de celdas (BNF)

Notación	Definición sintáctica
[a]	Opcional
{a}	Repitición ilimitada (0 a n)
a | b	Selección
A	Elemento sintáctico
A	Terminal del lenguaje
A ::=	Especificación de regla

alpha ::= **A | B | ... | Z**
digit ::= **0 | 1 | ... | 9**
char ::= alpha | digit
sign ::= **[+] | -**
int_const ::= [sign] digit {digit}
variable ::= **a** index index | **ALL** | **Row** index | **Column** index | **input**

expression ::= bool_exp
bool_exp ::= [**NOT**] int_exp rel_op int_exp | bool_exp rel_op bool_exp | delay

rel_op ::= **= | != | > | < | >= | <= | AND | OR**

int_exp ::= [sign] int_sent {ar_op int_sent}
int_sent ::= variable | int_const | func | (expression)

ar_op ::= **+ | - | * | / | ^**

func ::= **one_count() | zero_count() | fi_count()**

sentence ::= **0 | 1 |** ϕ (expression);

specification ::= sentence {sentence}

delay::= **tr_delay**(int_sent) | **in_delay**(int_sent);

index ::= 1 | 2 | 3

APENDICE II

```
/*********************************************************************
* DESCRIPTION: Atomic Model CPU
*
* DATE: 27/6/1998
*********************************************************************/

#include <math.h>       // fabs( ... )
#include "cpu.h"        // base header
#include "message.h"    // InternalMessage ....
#include "distri.h"     // class Distribution
#include "mainsimu.h"   // class MainSimulator
#include "except.h"     // MASSERT( ... )
#include "strutil.h"    // str2float( ... )

/** public functions **/

/*********************************************************************
* Function Name: CPU
* Description: constructor
*********************************************************************/
CPU::CPU( const string &name )
  : Atomic( name )
  , in( this->addInputPort( "in" ) )
  , out( this->addOutputPort( "out" ) ) {
}

/*********************************************************************
* Function Name: externalFunction
* Description: the CPU receives one job
*********************************************************************/
Model &CPU::externalFunction( const ExternalMessage &msg ) {
  pid = static_cast< int >( msg.value() ) ;
  this->holdIn( active, Time( static_cast<float>( fabs( this->distribution().get() ) ) ) ) ;
  return *this ;
}

/*********************************************************************
* Function Name: internalFunction
* Description:
*********************************************************************/
Model &CPU::internalFunction( const InternalMessage & ) {
  this->passivate() ;
  return *this ;
}

/*********************************************************************
* Function Name: outputFunction
* Description:
*********************************************************************/
Model &CPU::outputFunction( const InternalMessage &msg ) {
  this->sendOutput( msg.time(), out, pid ) ;
  return *this ;
}
```

```
/*********************************************************************
* DESCRIPTION: class Generator
*
* DATE: 27/6/1998
*********************************************************************/

#include <math.h>          // fabs( ... )
#include "generat.h"       // base header
#include "message.h"       // class InternalMessage
#include "mainsimu.h"      // class Simulator
#include "distri.h"        // class Distribution
#include "strutil.h"       // str2float( ... )

/*********************************************************************
* Function Name: Generator
* Description: constructor
*********************************************************************/
Generator::Generator( const string &name )
  : Atomic( name )
  , out( this->addOutputPort( "out" ) ) {
}

/*********************************************************************
* Function Name: ~Generator
* Description: Destructor
*********************************************************************/
Generator::~Generator() {
  delete dist ;
}

/*********************************************************************
* Function Name: initFunction
* Description:
*********************************************************************/
Model &Generator::initFunction() {
  pid = 0;
  this->holdIn( active, Time::Zero ) ;
  return *this ;
}

/*********************************************************************
* Function Name: internalFunction
* Description:
*********************************************************************/
Model &Generator::internalFunction( const InternalMessage & ) {
  this->holdIn( active, Time( static_cast< float >( fabs( this->distribution().get() ) ) ) ) ;
  return *this ;
}

/*********************************************************************
* Function Name: outputFunction
* Description:
*********************************************************************/
Model &Generator::outputFunction( const InternalMessage &msg ) {
  this->sendOutput( msg.time(), out, pid++ ) ;
  return *this ;
}
```

```
/*****************************************************************
* DESCRIPTION: class Transducer (implementation)
*
* DATE: 27/6/1998
*****************************************************************/
#include "transduc.h"   // base class
#include "message.h"    // class ExternalMessage
#include "except.h"    // MException
#include "mainsimu.h"       // class MainSimulator


/*****************************************************************
* Function Name: Transducer
* Description: Constructor
*****************************************************************/
Transducer::Transducer( const string &name )
  : Atomic( name )
  , arrived( this->addInputPort( "arrived" ) )
  , solved( this->addInputPort( "solved" ) )
  , throughput( this->addOutputPort( "throughput" ) )
  , cpuUsage( this->addOutputPort( "cpuusage" ) )
{
  frec = MainSimulator::Instance().getParameter( this->description(), "frecuence" ) ;
  if( frec == Time::Zero )
    frec = "0:1:0:0" ;

  tUnit = MainSimulator::Instance().getParameter( this->description(), "timeUnit" ) ;
  if( tUnit == Time::Zero )
    tUnit = "0:1:0:0" ;
}

/*****************************************************************
* Function Name: initFunction
* Description:
*****************************************************************/
Model &Transducer::initFunction() {
  procCount = 0 ;
  cpuLoad = 0 ;
  unsolved.erase( unsolved.begin(), unsolved.end() ) ;
  this->holdIn( active, this->frecuence() ) ;
  return *this ;
}


/*****************************************************************
* Function Name: externalFunction
* Description:
*****************************************************************/
Model &Transducer::externalFunction( const ExternalMessage &msg ) {
  cpuLoad += static_cast< long >( ( msg.time() - this->lastChange() ).asMsecs() * unsolved.size() );
  if( msg.port() == arrived )   {
    if( unsolved.find( msg.value() ) != unsolved.end() )      {
      MException e( string("Unresolved Work Id: ") + msg.value() + " is duplicated." );
      e.addLocation( MEXCEPTION_LOCATION() );
      throw e;
    }
    unsolved[ msg.value() ] = msg.time() ;
  }
```

```
  if( msg.port() == solved ) {
    JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

    if( cursor == unsolved.end() ) {
      MException e( string("Resolved Work Id: ") + msg.value() + " Not Found!" );
      e.addLocation( MEXCEPTION_LOCATION() );
      throw e;
    }
    procCount ++ ;
    unsolved.erase( cursor ) ;
  }
  return *this ;
}


/*****************************************************************
* Function Name: internalFunction
* Description:
*****************************************************************/
Model &Transducer::internalFunction( const InternalMessage &) {
  this->holdIn( active, this->frecuence() ) ;
  return *this ;
}


/*****************************************************************
* Function Name: outputFunction
* Description:
*****************************************************************/
Model &Transducer::outputFunction( const InternalMessage &msg ) {
  float time( msg.time().asMsecs() / this->timeUnit().asMsecs() ) ;
  this->sendOutput( msg.time(), throughput, procCount / time ) ;
  cpuLoad += static_cast< long >( ( msg.time() - this->lastChange() ).asMsecs() * unsolved.size() ) ;
  this->sendOutput( msg.time(), cpuUsage  , cpuLoad / msg.time().asMsecs() ) ;
  return *this ;
}
```

TABLA DE CONTENIDOS

1. INTRODUCCIÓN 5

2. BACKGROUND 8

2.1.1 SISTEMAS, MODELOS Y PARADIGMAS 8
2.1.2 MODELOS Y SIMULACIÓN 9
2.1.3 CATEGORÍAS DE PARADIGMAS DE MODELADO 11
2.1.4 SIMULACIÓN DE SISTEMAS DE EVENTOS DISCRETOS 12
2.1.5 PARADIGMAS DE ESPECIFICACIÓN DE MODELOS DEDS 12
2.1.6 ESPECIFICACIÓN DE MODELOS SIMULABLES 13
2.1.7 COMPORTAMIENTO Y ESTRUCTURA 14
2.2 FORMALISMO DEVS 15
2.2.1 MODELOS ATÓMICOS 16
2.2.2 MODELOS ACOPLADOS 20
2.2.3 PRINCIPIO DE SIMULACIÓN ABSTRACTA 24
2.3 AUTÓMATAS CELULARES 28
2.3.1 AUTÓMATAS CELULARES ASINCRÓNICOS. 29
2.3.2 EJEMPLOS DE APLICACIÓN 31
2.4 SIMULACION PARALELA 34
2.4.1 FORMAS DE DESCOMPOSICIÓN PARALELA 34
2.4.2 SINCRONIZACIÓN EN MODELOS DISTRIBUÍDOS 35
2.4.3 PROCESOS FÍSICOS Y LÓGICOS 36
2.4.4 MECANISMOS CONSERVADORES 37
2.4.5 MECANISMOS OPTIMISTAS 40

3. FORMALISMO CELL-DEVS 43

3.1 DEFINICIÓN FORMAL DE AUTÓMATAS CELULARES 43
3.1.1 AUTÓMATAS CELULARES CONCEPTUALES 43
3.1.2 AUTÓMATAS CELULARES EJECUTABLES SINCRÓNICOS 45
3.1.3 AUTÓMATAS CELULARES EJECUTABLES ASINCRÓNICOS 47
3.2 MODELOS CELL-DEVS CON DEMORAS DE TRANSPORTE 48
3.2.1 ESPECIFICACIÓN FORMAL DE MODELOS ATÓMICOS CELL-DEVS MODULARES CON DEMORAS DE TRANSPORTE. 48
3.2.2 MODELOS CELULARES CON DEMORAS INERCIALES 55
3.2.3 MODELOS ATÓMICOS CELL-DEVS CON TRES ESTADOS 60
3.3 ESPECIFICACIÓN FORMAL DE MODELOS ACOPLADOS CELL-DEVS CERRADOS CON DEMORAS DE TRANSPORTE. 61
3.4 ESPECIFICACIÓN FORMAL DE MODELOS CELL-DEVS ACOPLADOS GENÉRICOS. 66
3.5 MODELOS ACOPLADOS DEVS: REDEFINICIÓN PARA INCLUIR MODELOS CELL-DEVS 70
3.6 CLAUSURA BAJO ACOPLAMIENTO DE LOS MODELOS CELL-DEVS 71

4. SIMULACIÓN DE ESPACIOS CELL-DEVS 73

4.1 DEFINICIÓN DE UN MODELO DE IMPLEMENTACIÓN CELL-DEVS. 73
4.2 SIMULACIÓN ABSTRACTA DE ESPACIOS CELL-DEVS 77
4.3 SIMULACIÓN ACHATADA DE LOS ESPACIOS DE CELDA 78
4.4 SIMULACIÓN PARALELA DE UN ESPACIO CELL-DEVS. 81
4.4.1 COORDINADORES PESIMISTAS 82
4.4.2 COORDINADORES OPTIMISTAS 84
4.4.3 CUESTIONES DE DESEMPEÑO 85

5. UN EJEMPLO DE MODELIZACIÓN Y SIMULACIÓN CELL-DEVS 87

5.1 ESPECIFICACIÓN DEL COMPORTAMIENTO DE LAS CELDAS. 88
5.1.1 MODELO A (SMOG EN EL BARRIO RESIDENCIAL) 88
5.1.2 MODELO B (TRÁFICO EN EL BARRIO COMERCIAL) 88
5.1.3 MODELO C (AUTOPISTA) 89
5.2 ESPECIFICACIÓN FORMAL DE LOS MODELOS 90
5.3 MECANISMO DE SIMULACIÓN 98
5.3.1 SIMULACIÓN JERÁRQUICA DE LOS ESPACIOS DE CELDAS. 98
5.3.2 SIMULACIÓN DE ESPACIOS DE CELDAS ACHATADOS. 102

6. HERRAMIENTA DE SIMULACIÓN 104

6.1 GAD: UN ENTORNO GENÉRICO PARA SIMULACIONES DEVS 104
6.2 INTERACCIÓN DE MENSAJES EN GAD 109
6.3 LA HERRAMIENTA CD++ 112
6.4 INTERACCIÓN DE MENSAJES EN CD++ 115
6.5 INCORPORACIÓN DE NUEVOS MODELOS 117

7. RESULTADOS OBTENIDOS. DISCUSION. 120

7.1 EXPERIENCIAS CON AUTÓMATAS CELULARES SINCRÓNICOS Y ASINCRÓNICOS 120
7.2 EXPERIENCIAS CON MODELOS CLÁSICOS DEVS 126
7.3 EXPERIENCIAS CON MODELOS CELL-DEVS 131

8. CONCLUSION. TRABAJO FUTURO. 142

9. REFERENCIAS 144

10. BIBLIOGRAFIA ADICIONAL 147

TABLA DE FIGURAS

Figura 1. Pasos en el estudio de un sistema por medio de modelado [Gia96]. 10
Figura 2 - Relaciones de modelado y simulación [Zei90]. 13
Figura 3. Ejemplo de un modelo de comportamiento (a) y de estructura (b) para el mismo sistema real. 14
Figura 4 - Trayectorias en un modelo atómico DEVS [ChZ94]. 17
Figura 5 - Un modelo de buffering simple 18
Figura 6. Un modelo atómico de desempeño de un procesador. 19
Figura 7. Ejecución del modelo del Ejemplo 8. 19
Figura 8 - Un modelo acoplado 21
Figura 9 - Especificación del modelo acoplado EF y de sus componentes. 22
Figura 10 - Especificación del modelo acoplado EF-P. 23
Figura 11. Relación entre Modelos y Procesadores. 24
Figura 12 - Procesadores, variables usadas y sus mensajes. 25
Figura 13. Simulador abstracto para el modelo EF-P. 27
Figura 14. Detalle del modelo acoplado EF-P. 27
Figura 15. Un fragmento de autómata celular bidimensional. 28
Figura 16. Ejecución de una transición en el juego de la Vida. 30
Figura 17. Vecindad del modelo de tráfico en una dirección. 31
Figura 18. Mecanismos de descomposición. (a) Replicación independiente; (b) Compilador paralelizante; (c) funciones distribuídas; (d) eventos distribuídos; (e) descomposición del modelo. 35
Figura 19. Simulación paralela dirigida por tiempo: (a) Sincrónica; (b) Asincrónica. Dirigida por eventos: (c) Sincrónica; (d) Asincrónica. 36
Figura 20. Error de causalidad por sincronización entre procesos lógicos [Fuj90]. 37
Figura 21. Deadlock en aproximaciones conservadoras. 37
Figura 22. Evitando Deadlock con mensajes nulos. 38
Figura 23. Uso de mensajes nulos para la Figura 21. 39
Figura 24. Desempeño de algoritmos conservadores (a) Eficiencia; (b) Aceleración [Fuj90]. 39
Figura 25. Llegada de un antimensaje usando Time Warp. 41
Figura 26. Aceleración de simulación usando Time Warp (256 procesos lógicos) [Fuj90]. 42
Figura 27. Descripción informal de una celda atómica con demoras de transporte. 49
Figura 28. Definición de δ_{int}, δ_{ext} y λ para modelos TDC. 51
Figura 29. Función booleana con demora de transporte. 54
Figura 30. Trayectorias de entrada/salida con demoras de transporte para el modelo anterior. 54
Figura 31. Comportamiento de un modelo atómico de celdas con demora inercial. 56
Figura 32. Funciones de transición interna y externa para modelos con demoras inerciales. 57
Figura 33. Trayectorias de entrada y salida con demoras inerciales. 59
Figura 34. Trayectorias de entrada/salida para celdas de tres estados con demoras de transporte. 61
Figura 35. (a) Una celda, su vecindad y la lista de vecinos; (b) conexión de los ports de salida de la celda i,j (usando la lista de vecindad); (c) conexión de los ports de entrada de la celda i,j (usando la lista de vecindad inversa). 66
Figura 36. Ejemplo de acoplamiento usando la función Z_{ij} para espacios Cell-DEVS (a) Modelos básicos; (b) listas Xi e Yi para cada modelo; (c) Acoplamiento Z_{ij}. 71
Figura 37. Jerarquía de clases para el entorno de simulación Cell-DEVS. 74
Figura 38. Extensiones de operaciones booleanas para estados indefinidos [AAD97]. 75
Figura 39. Estructuras del espacio de celdas. (a) Espacio de celdas básico y su vecindad. (b) Acoplamiento usando coordinadores jerárquicos. (c) Definición usando modelos achatados. (d) Jerarquía de procesadores para (b). 79
Figura 40. Reacción de los coordinadores ante mensajes (a) Jerárquico (b) Achatado. 79
Figura 41. Simulación achatada para espacios Cell-DEVS. 81
Figura 42. Estructura de un proceso lógico para espacios de celdas. 82
Figura 43. Funciones de coordinación pesimista para entornos Cell-DEVS. 83
Figura 44. Funciones de coordinación optimista para entornos Cell-DEVS. 85
Figura 45 - Modelo de Frecuencia de Rollback. 86
Figura 46. Acoplamiento de modelos Cell-DEVS y DEVS [WG98]. 87
Figura 47. Especificación para el modelo A 88
Figura 48. Especificación para el modelo B. (a) Parámetros del espacio de celdas. (b) Descripción del movimiento de los automotores. 88
Figura 49. Especificación del modelo C. 89
Figura 50. Movimientos válidos para la regla 1 (distintas expresiones). 90
Figura 51. Movimientos válidos para la regla 2 (distintas expresiones). 90

Figura 52. Definición del modelo D. 91
Figura 53. Ejemplo de ejecución del modelo D. 92
Figura 54. Definición del modelo E. 93
Figura 55. Espacio de celdas A; celdas de entrada/salida del modelo. 93
Figura 56. Interconexión de ports para la función Z_A. 94
Figura 57. Espacio de celdas C; celdas de entrada/salida del modelo. 95
Figura 58. Espacio de celdas B; celdas de entrada/salida del modelo. 96
Figura 59. Definición de estructura del modelo M. 97
Figura 60. Jerarquía de procesadores de simulación para el modelo M. 98
Figura 61. Estado inicial del modelo M. 98
Figura 62. Contenidos iniciales de las estructuras de datos de los Simuladores/Coordinadores. 99
Figura 63. Actualización de los valores de la vecindad local. 100
Figura 64. Jerarquía de clases de la herramienta GAD [BBW98c]. 104
Figura 65. Jerarquía de clases para Messages. 106
Figura 66. Jerarquía de clases para MsgAdmin. 106
Figura 67. Jerarquía de clases para ModelAdm. 107
Figura 68. Jerarquía de clases para SimLoader. 108
Figura 69. Descripción de MainSimulator. 108
Figura 70. Métodos activados al iniciar la simulación [BBW98c]. 109
Figura 71. Mensajes y métodos usados durante la inicialización [BBW98c]. 110
Figura 72. Mensajes y métodos correspondientes a eventos internos [BBW98c]. 111
Figura 73. Mensajes y métodos correspondientes a eventos externos [BBW98c]. 111
Figura 74. Mensajes y métodos correspondientes a eventos externos[BBW98c]. 112
Figura 75 - (a) Jerarquía de clases de Modelos en la herramienta CD++. (b) Detalle de las clases para manejo de espacios de celdas. (c) Jerarquía de Simuladores en la misma herramienta. [BBW98c] 114
Figura 76 - Modelos achatados. 115
Figura 77 - Interacción de mensajes celulares [BBW98c]. 116
Figura 78. Configuración de un modelo atómico 117
Figura 79. Parámetros de un modelo acoplado celular 118
Figura 80. Sintaxis del lenguaje de especificación implementado. 118
Figura 81. Parámetros de un modelo acoplado celular 119
Figura 82. Comportamiento para autómatas celulares sincrónicos 120
Figura 83. Resultados de ejecución del Juego de la Vida. 121
Figura 84. Tiempos de ejecución del Juego de la Vida con respecto a la variación en la dimensión del espacio. 122
Figura 85. Resultados de ejecución de Wa-Tor. 122
Figura 86. Comparación entre autómatas celulares sincrónicos y asincrónicos. Modelo de circulación de múltiples vías en una sola mano. Demora entre ejecuciones: 50 ms. 123
Figura 87. Comparación entre autómatas celulares sincrónicos y asincrónicos con distintas demoras por celda. (a) 7 vehículos; (b) 49. 123
Figura 88. (a) Peores casos de uso de memoria para el Juego de la Vida. (b) Detalle. 124
Figura 89. (a) Casos promedio para 1000 celdas iniciales. (b) Detalle. 124
Figura 90. Modelo Atómico Queue 126
Figura 91. Definición del modelo atómico Queue. 126
Figura 92. Constructor para la clase Queue. 127
Figura 93. Funciones de transición para la clase Queue. 127
Figura 94. Archivo de definición de componentes y enlaces para la clase Queue. 128
Figura 95. Acoplamiento de los modelos [BBW98a]. 129
Figura 96. Definición del acoplamiento en la herramienta 129
Figura 97. Generación de trabajos con mayor frecuencia que la de consumo [BBW98a]. 130
Figura 98. Generación de trabajos con menor frecuencia que el consumo [BBW98a]. 131
Figura 99. Especificación del Juego de la Vida usando la herramienta. 132
Figura 100. Resultado de ejecución del Juego de la Vida usando la herramienta. 132
Figura 101. Mensajes producidos durante la ejecución del modelo achatado. 133
Figura 102. Mensajes producidos durante la ejecución del modelo achatado. 134
Figura 103. Resultados de ejecución del Juego de la Vida. 134
Figura 104. Resultados obtenidos variando la duración de la simulación. 135
Figura 105. Tiempos de inicialización (a) Jerárquico (b) Achatado. 135
Figura 106. Resultados de ejecución del Juego de la Vida. 136
Figura 107. Intercambio de mensajes en el modelo celular de tráfico urbano. 137

Figura 108. Intercambio de mensajes usando el coordinador achatado. ____ 137
Figura 109. Estado inicial para la simulación de tráfico. ____ 137
Figura 110. Comparación entre jerárquicos y achatados para la simulación de tráfico. ____ 138
Figura 111. Definición de un modelo celular de tráfico urbano usando la herramienta. ____ 139
Figura 112. Comparación entre jerárquicos y achatados para la simulación de tráfico. ____ 139
Figura 113. Comparación de tiempos de desarrollo para el Juego de la Vida. ____ 140
Figura 114. Comparación de tiempos de desarrollo para simulación de tráfico. ____ 141
Figura 115. Comparación de tiempos totales de desarrollo para ambas aplicaciones. ____ 141

Tabla 1. Secuencia de ejecución de un modelo celular con demoras de transporte. ____ 55
Tabla 2 - Secuencia de ejecución de la Figura 33. ____ 59
Tabla 3. Secuencia de ejecución para el modelo de celda de tres estados con demora de transporte. ____ 61
Tabla 4. Principales operadores y funciones para definir el comportamiento de una celda. ____ 76
Tabla 5. Formato de las tablas de cálculo local. ____ 76